Applied Post-Quantum Cryptography

Stephen Duan Wei Li

July 31, 2026

Standards snapshot: 29 July 2026. Standards and drafts cited in this book are identified by their exact publication or draft version.

Contents

Part I

Foundations

Chapter 1

The Quantum Threat

1.1 Why this chapter matters

Every other chapter in this book builds toward schemes labelled *post-quantum*. This chapter explains what that word is reacting to. Post-quantum cryptography exists because a sufficiently large quantum computer would break almost all of the public-key cryptography deployed today – the very algorithms protecting web traffic, software updates, and stored data right now. Understanding *which* problems fall, *which* survive, and *why* is the motivation for everything that follows.

1.2 Learning goals

After reading this chapter, you should be able to:

- name the hard problems that today's public-key cryptography relies on;
- explain, at a high level, what Shor's and Grover's algorithms do;
- distinguish what a quantum computer *breaks* from what it merely *weakens*;
- explain "harvest now, decrypt later" and why migration is urgent;
- place the published NIST standards and the main mathematical families in context.

1.3 What today's cryptography rests on

Classical public-key cryptography stands on two number-theoretic problems that are believed hard for ordinary computers:

- **Integer factorization** – given a large $N = p \cdot q$, recover the primes p and q. This is the foundation of **RSA**.
- **Discrete logarithm** – given g and g^x in a suitable group, recover x. This underlies **Diffie–Hellman** key exchange and elliptic-curve schemes such as **ECDSA** and **ECDH**.

For a classical computer, the best known algorithms for both problems run in super-polynomial (sub-exponential or worse) time, so the parameters can be chosen to make an attack take longer than the age of the universe. That safety margin is exactly what a quantum computer removes.

1.4 Shor's algorithm: the break

In 1994 Peter Shor showed that a quantum computer can factor integers and compute discrete logarithms in *polynomial* time [9]. The engine is *period finding*: both problems can be recast as finding the period of a cleverly chosen function, and a quantum computer extracts that period efficiently using the quantum Fourier transform – evaluating the function on a superposition of all inputs at once, then interfering the results so that the period stands out.

The consequence is stark: RSA, Diffie–Hellman, ECDSA, and ECDH are *all* broken by a large enough quantum computer, no matter how big the keys are. Doubling an RSA key only adds polynomial work for Shor's algorithm – it does not restore security. There is no parameter choice that saves them.

1.5 Grover's algorithm: the weakening

Symmetric cryptography (block ciphers like AES) and hash functions (like SHA-3) face a milder threat. Grover's algorithm [10] speeds up *generic search*: finding a marked item among N possibilities takes about $\sqrt{N}$ steps instead of N. Against a symmetric key of b bits, this cuts the effective security to roughly $b/2$ bits.

Crucially, this is a *quadratic* speedup, not an exponential one. For exhaustive key search, AES-256 therefore retains roughly 128 bits of generic quantum-search security. Hash properties must be treated separately: for an m-bit ideal hash, preimage and second-preimage search have a generic Grover-style cost of about $2^{m/2}$, whereas collision search has different quantum algorithms and cost models. In the quantum-query model, the Brassard–Høyer–Tapp algorithm finds collisions in about $2^{m/3}$ queries; memory, circuit cost, and concrete security estimates require separate analysis. Symmetric primitives *survive*, but their parameters and security goals must be chosen carefully.

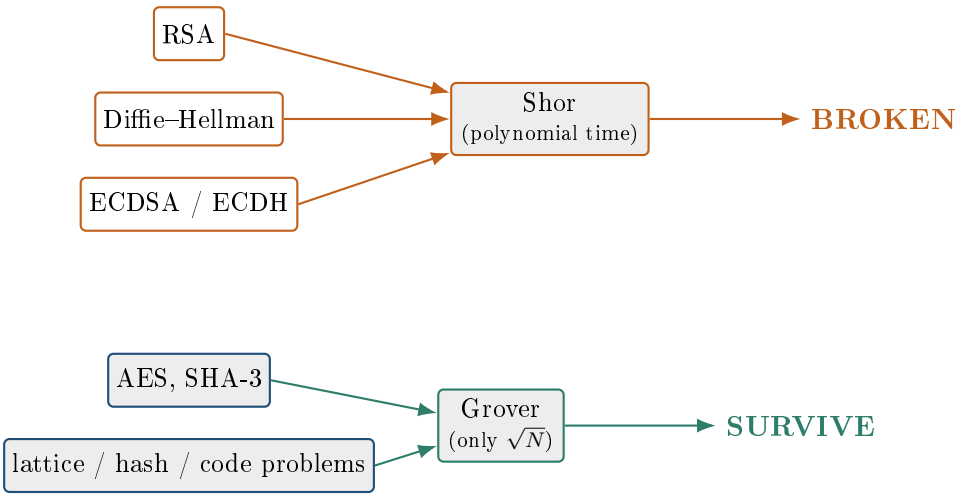


Figure 1.1: The quantum threat, sorted. Shor's algorithm breaks the number-theoretic public-key schemes outright (orange). Grover's algorithm gives generic quadratic speedups for search (teal); its implications for keys, preimages, and collisions are not identical.

Primitive	Quantum algorithm	Impact
RSA	Shor	broken (any key size)
Diffie–Hellman, ECDH	Shor	broken
ECDSA	Shor	broken
AES-128	Grover	$\approx$ 64-bit; use AES-256
SHA-256 preimage	Grover	generic cost $\approx 2^{128}$
SHA-256 collision	quantum collision algorithms	analyse separately
Lattice / code / hash problems	(none known)	survive

Table 1.1: How a large quantum computer affects common primitives. Only the number-theoretic public-key schemes are broken outright.

1.6 Harvest now, decrypt later

A natural objection is that large quantum computers do not yet exist, so why migrate today? The answer is *harvest now, decrypt later*. An adversary can record encrypted traffic today and decrypt it years later once a quantum computer is available. Any data that must remain confidential beyond the arrival of quantum computing is therefore already at risk.

Signatures create a different risk. A future quantum adversary may forge *new* signatures, including signatures on malicious firmware or certificates, and may undermine long-term verification. Whether a historically verified signature remains useful depends on its timestamp, archival evidence, and trust model; it is not a case of "record now, decrypt later." Both risks argue for migration before the threat materializes.

1.7 The response: post-quantum standards

Because Shor's algorithm targets the specific structure of factoring and discrete logarithms, the defense is to build public-key cryptography on entirely different hard problems – ones with no known efficient quantum algorithm. After a multi-year public competition, NIST selected the first post-quantum schemes:

- **FIPS 203 – ML-KEM** (Kyber): key encapsulation, based on module lattices [1];
- **FIPS 204 – ML-DSA** (Dilithium): signatures, based on module lattices [2];
- **FIPS 205 – SLH-DSA** (SPHINCS$^+$): signatures, based only on hash functions [3];
- a future FN-DSA standard based on Falcon is tracked separately; any implementation discussion must name the exact draft or specification version it follows.

Post-quantum cryptography is commonly organized into *five broad mathematical families* of hard problems, each offering a different route to quantum-resistant public-key cryptography [6]. This is a teaching classification, not a claim that the families are cryptographically independent.

- **Lattice-based** – short vectors and noisy linear systems in high-dimensional lattices [7, 8]. The workhorse of the standards, balancing speed and size; the main subject of this book.
- **Hash-based** – security reduced to nothing but a hash function; the most conservative choice, resting on the longest-understood assumption (Chapter 21).

- **Code-based** – decoding random-looking linear codes; the oldest family, with a decades-long track record (Chapter 22).
- **Multivariate** – solving systems of multivariate quadratic equations over a finite field [42]; very compact signatures and large keys, with a turbulent cryptanalytic history (Chapter 24).
- **Isogeny-based** – walking between elliptic curves along secret isogenies; the smallest keys of all, but the youngest and least settled family (Chapter 25).

The NIST standards are concrete *implementations* of the first two families: three are lattice schemes and one is hash-based. The code-based, multivariate, and isogeny families have no FIPS standard yet, but each offers active candidates and, crucially, a structurally different hedge should a future attack ever undermine lattices. Table 1.2 places every standard in its family.

Family	Hard problem	Standard / leading scheme	Role
Lattice	Module-LWE	FIPS 203 ML-KEM	KEM
Lattice	Module-LWE / SIS	FIPS 204 ML-DSA	signature
Lattice	NTRU	FN-DSA / Falcon specifications	signature
Hash	preimage / collision	FIPS 205 SLH-DSA	signature
Code	syndrome decoding	Classic McEliece, HQC	KEM
Multivariate	MQ (quadratic systems)	UOV, MAYO (candidates)	signature
Isogeny	supersingular isogeny path	SQIsign (candidate)	signature

Table 1.2: Five commonly discussed mathematical families of post-quantum cryptography. FIPS 203–205 cover lattice- and hash-based schemes; other families remain active areas of research and standardization.

1.8 Summary

- Today's public-key cryptography (RSA, Diffie–Hellman, ECC) rests on factoring and discrete logarithms.
- **Shor's algorithm** solves both in polynomial time on a quantum computer, breaking these schemes at *any* key size.
- **Grover's algorithm** gives a quadratic generic-search speedup; key search, hash preimages, and hash collisions need separate security analyses.
- Confidentiality has a "harvest now, decrypt later" risk; signature migration concerns future forgery and long-term verification.
- The rest of the book surveys five broad mathematical families – lattice, hash, code, multivariate, and isogeny – with FIPS 203–205 as published NIST standards at this snapshot.

The remaining chapters start from the ground up: the mathematical objects – finite fields, vectors, polynomials, and lattices – on which these new hard problems are built.

Chapter 2

SageMath Basics for Post-Quantum Cryptography

2.1 Why this chapter matters

This chapter is not meant to teach SageMath in isolation. Its purpose is to build a small experimental laboratory for post-quantum cryptography. Later chapters on LWE, RLWE, Kyber, Dilithium, McEliece, and lattice attacks all depend on the objects introduced here.

2.2 Learning goals

After finishing this chapter, you should be able to:

- create integers, rationals, reals, complex numbers, finite fields, vectors, matrices, and polynomials in SageMath;
- distinguish Sage objects from ordinary Python values;
- express algebraic structures such as $\mathbb{F}_q$, $R_q = \mathbb{Z}_q[x]/(f(x))$, and $Ax = b$ in Sage;
- run small experiments that foreshadow later cryptographic constructions.

Every algebraic object used later in the book is a specialization of a small handful of number systems. Figure 2.1 lays out that map before we start building each piece in Sage.

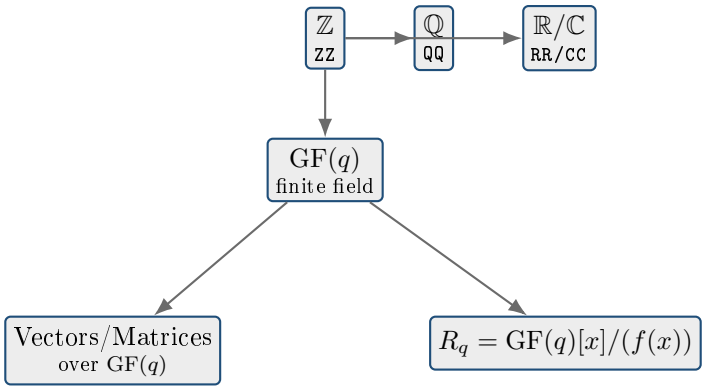


Figure 2.1: The map of algebraic objects this book builds on. Ordinary integers $\mathbb{Z}$ extend to fractions $\mathbb{Q}$ and to $\mathbb{R}/\mathbb{C}$ for numerical and Fourier-style reasoning, but the object cryptography actually computes with is the finite field $\mathrm{GF}(q)$, obtained by reducing $\mathbb{Z}$ modulo a prime q. Everything in later chapters – vectors and matrices for LWE, and the quotient ring R_q for RLWE and Kyber – is built on top of $\mathrm{GF}(q)$.

2.3 Installing SageMath

There are two practical ways to begin.

2.3.1 Option 1: Docker

A convenient approach is to start SageMath from a container:

```
docker run -it sagemath/sagemath sage
```

Once the shell starts, you can enter Sage commands directly.

2.3.2 Option 2: Local installation

On Ubuntu, a typical installation command is:

```
sudo apt install sagemath
```

On macOS, one may use:

```
brew install sage
```

2.3.3 Notebook workflow

For experiments and exploratory work, a notebook interface is often most convenient:

```
sage -n jupyter
```

Then open the displayed local URL in a browser.

2.4 SageMath and Python

SageMath is built on top of Python, but it adds richer mathematical types and objects. A plain Python integer and a Sage integer are not the same thing.

```
a = 3
b = 4
a + b
```

In Python, the result is an ordinary integer. In Sage, the corresponding object may carry algebraic meaning.

```
a = Integer(3)
b = Integer(4)
a + b
```

To inspect the type, use:

```
type(a)
```

This matters in cryptography because the symbol 3 may represent an ordinary integer in one context, but a field element in another. For example, $3 \in \mathrm{GF}(7)$ is not the same object as the integer 3 in $\mathbb{Z}$.

2.5 Basic number systems

SageMath provides several standard number domains that are useful in later cryptographic work.

2.5.1 Integers $\mathbb{Z}$

The integer ring is available as `ZZ`.

```
ZZ
```

You can create integers explicitly:

```
a = ZZ(10)
a
```

Basic arithmetic works as expected:

```
a = ZZ(10)
b = ZZ(3)
a + b
```

Division behaves more carefully than in Python because Sage keeps symbolic and exact arithmetic whenever possible:

```
10/3
```

The result is the rational number $\frac{10}{3}$. Integer division is available through:

```
10//3
```

and modular arithmetic through:

```
10 % 3
```

2.5.2 Rationals $\mathbb{Q}$

The rational field is `QQ`.

```
a = QQ(1)/3
a
```

This is useful because many lattice algorithms, such as LLL and Gram–Schmidt-style computations, work naturally over rational numbers.

2.5.3 Reals $\mathbb{R}$ and complexes $\mathbb{C}$

SageMath also supports real and complex domains.

```
RR
CC
```

For example:

```
sqrt(2)
RR(sqrt(2))
CC(1 + 2*I)
```

These domains become relevant later when discussing numerical methods, Fourier-style transforms, and related implementation ideas.

2.6 Finite fields

Finite fields are among the most important structures in modern cryptography. A finite field is usually written as $\mathrm{GF}(q)$ or $\mathbb{F}_q$. Before computing with them, it is worth stating precisely what they are, because the whole book rests on them.

Definition 2.1 (Field)**.** A *field* is a set $\mathbb{F}$ with addition and multiplication such that $(\mathbb{F}, +)$ and $(\mathbb{F} \setminus \{0\}, \cdot)$ are both abelian groups (with identities 0 and 1) and multiplication distributes over addition. Concretely: one can add, subtract, multiply, and divide by anything nonzero.

A *finite* field is simply a field with finitely many elements. Their existence is completely classified.

Theorem 2.2 (Classification of finite fields)**.** *A finite field of order q exists if and only if $q = p^m$ is a power of a prime p, and it is then unique up to isomorphism, denoted $\mathbb{F}_q$ or $\mathrm{GF}(q)$. When $m = 1$ it is simply the integers modulo p,*

$$\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z} = \{0, 1, \ldots, p-1\},$$

with arithmetic reduced modulo p.

Almost every modulus in this book is prime, so the working picture is the clean one: $\mathbb{F}_q$ is "clock arithmetic" modulo q in which division is also available. This field structure makes tools such as Gaussian elimination available when they are needed. It is not, however, why ML-KEM or ML-DSA key generation works: those algorithms sample secrets and public values rather than solving a random linear system. Their prime moduli are chosen for several interacting reasons, including algebraic structure, NTT compatibility, noise distributions, correctness, security, and implementation efficiency.

2.6.1 Creating a finite field

For example, the field with seven elements is:

```
F = GF(7)
F
```

Elements are created by coercing values into the field:

```
a = F(3)
a
```

Arithmetic is performed modulo the prime:

```
a = F(6)
a + F(3)
```

The result is 2, because $6 + 3 = 9 \equiv 2 \pmod 7$.

2.6.2 Inverses

In cryptography, inverses appear constantly. For a field element a, the inverse a^{-1} satisfies $aa^{-1} = 1$.

```
a = F(3)
a.inverse()
```

Since $3 \cdot 5 = 15 \equiv 1 \pmod 7$, the inverse is 5.

Under the hood, `a.inverse()` does not search through all field elements; it runs the extended Euclidean algorithm, which computes $\gcd(a, q)$ while simultaneously tracking the coefficients of a Bézout identity $as+tq = \gcd(a, q)$, so that when $\gcd(a, q) = 1$ the coefficient s is exactly $a^{-1} \bmod q$.

Algorithm 1 Extended Euclidean Algorithm for Modular Inverse

Input: Integer a with $1 \le a < q$; modulus q with $\gcd(a, q) = 1$
Output: $a^{-1} \bmod q$, the unique $x \in \{0, \dots, q-1\}$ with $ax \equiv 1 \pmod q$

$(r_0, r_1) \leftarrow (q, a)$
$(s_0, s_1) \leftarrow (0, 1)$ ▷ invariant: $a\, s_i \equiv r_i \pmod q$ at every step
while $r_1 \neq 0$ **do**
    $t \leftarrow \lfloor r_0 / r_1 \rfloor$
    $(r_0, r_1) \leftarrow (r_1,\ r_0 - t\, r_1)$
    $(s_0, s_1) \leftarrow (s_1,\ s_0 - t\, s_1)$
end while
return $s_0 \bmod q$ ▷ $r_0 = \gcd(a, q) = 1$, so $a\, s_0 \equiv 1 \pmod q$

2.7 Linear algebra over a finite field

Finite-field linear algebra is the starting point for many lattice-based constructions.

```
F = GF(7)
A = Matrix(F, [[1, 2], [3, 4]])
A
```

The inverse can be computed as:

```
A.inverse()
```

and one can verify the identity matrix:

```
A * A.inverse()
```

This is the algebraic setting behind equations such as $As + e = b$, which will appear later in the study of LWE.

2.8 Vectors and matrices

Vectors are convenient for representing secret values and noise terms.

```
v = vector(F, [1, 2])
v
```

Matrix-vector multiplication is then straightforward:

```
A * v
```

The output is `(5, 4)`. The vector has length 2, matching the two columns of A; dimensions must agree for matrix–vector multiplication.

2.9 Polynomial rings

Polynomial rings are essential for the ring-based versions of lattice problems, especially RLWE and related constructions.

2.9.1 Building a polynomial ring

```
F = GF(7)
R.<x> = PolynomialRing(F)
```

Now x is a formal variable in the ring.

```
f = x^3 + 2*x + 1
f
```

Addition and multiplication are performed in the usual algebraic way:

```
f + f
f * f
```

2.9.2 Quotient rings

A quotient ring is created by imposing a polynomial relation. For example:

```
S.<x> = QuotientRing(R, R.ideal(x^3 + 1))
```

In this quotient, the relation $x^3 = -1$ holds. This idea is closely related to the ring structure used in Kyber, where one often works in a ring of the form

$$R_q = \mathbb{Z}_q[x]/(x^{256} + 1).$$

2.10 A first cryptographic experiment

As a first experiment, we can verify the behavior of modular inverses.

```
F = GF(17)
for i in range(1, 17):
    a = F(i)
    print(i, a.inverse() * a)
```

Every value should evaluate to 1 in the field.

2.11 A first LWE-style example

To foreshadow later chapters, we can create a simple linear system over a finite field:

```
q = 17
F = GF(q)
A = random_matrix(F, 3, 3)
s = random_vector(F, 3)
b = A * s
A
s
b
```

This is the basic algebraic shape behind LWE before noise is introduced.

2.12 Exercises

Try to solve the following exercises on your own.

1. Create GF(13) and compute the inverse of every nonzero element.
2. Create $R = \mathrm{GF}(17)[x]$ and compute $(x^2 + 3x + 1)^2$.
3. Create $\mathrm{GF}(17)[x]/(x^4 + 1)$ and verify that $x^4 = -1$.
4. Repeatedly sample $A \in \mathrm{GF}(17)^{4\times 4}$ until `A.is_invertible()` is true, then verify that $A^{-1}A = I$. What happens for a singular sample?

2.13 Connection to the next chapters

The next chapter, *Linear Algebra Foundations*, will deepen the linear-algebra viewpoint needed for lattice problems. After that, the polynomial viewpoint will be developed more fully in *Polynomial Rings*. These two chapters prepare the ground for LWE, RLWE, and ultimately Kyber and Dilithium.

Chapter 3

Linear Algebra Foundations

3.1 Why this chapter matters

This chapter is not a general linear algebra course. Its purpose is to develop, with enough precision to be useful later, the specific structures on which post-quantum cryptography is built: vector spaces over a finite field, linear maps and their matrices, and the rank–nullity relationship that governs when a linear system can be solved. The central object of lattice cryptography, the equation

$$A\mathbf{s} + \mathbf{e} = \mathbf{b},$$

is a linear system deliberately perturbed by a small error $\mathbf{e}$. To understand why the unperturbed system is easy and the perturbed one is (conjecturally) hard, we first need to be precise about what "solving a linear system" means.

3.2 Learning goals

After reading this chapter, you should be able to:

- state the axioms of a vector space and recognise $\mathbb{F}_q^n$ as the working example;
- define linear independence, span, basis, and dimension, and know why dimension is well defined;
- describe a matrix as the coordinate form of a linear map, and define its rank, kernel, and image;
- state the rank–nullity theorem and use it to reason about solvability of $A\mathbf{x} = \mathbf{b}$;
- explain precisely why $A\mathbf{s} = \mathbf{b}$ is easy while $A\mathbf{s} + \mathbf{e} = \mathbf{b}$ is the hard LWE problem.

3.3 Scalars: the underlying field

Every vector space is built over a set of *scalars* that can be added, subtracted, multiplied, and divided. That set is a *field* (Chapter 2, where the finite field $\mathbb{F}_q = \mathrm{GF}(q)$ was introduced). We recall the definition because the rest of the chapter depends on it.

Definition 3.1 (Field)**.** A *field* is a set $\mathbb{F}$ with two operations $+$ and $\cdot$ such that $(\mathbb{F}, +)$ is an abelian group with identity 0, the nonzero elements $(\mathbb{F} \setminus \{0\}, \cdot)$ form an abelian group with identity 1, and multiplication distributes over addition. In particular every nonzero element has a multiplicative inverse.

The familiar fields are $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$. Cryptography instead works over the finite field $\mathbb{F}_q$; for a prime q this is simply the integers modulo q,

$$\mathbb{F}_q = \mathbb{Z}/q\mathbb{Z} = \{0, 1, \ldots, q-1\},$$

with arithmetic reduced modulo q. The property that makes it a field, rather than merely a ring, is that division is available: because q is prime, every nonzero residue has an inverse. This is exactly what lets Gaussian elimination run, and it is why cryptographic parameters such as $q = 3329$ (Kyber) are chosen prime.

3.4 Vectors and vector spaces

In school a vector is a coordinate tuple such as $(1, 2)$. For our purposes a vector is an element of a *vector space*: a structured container of field elements that can be added and scaled.

Definition 3.2 (Vector space)**.** A *vector space* over a field $\mathbb{F}$ is a set V with an addition $V \times V \to V$ and a scalar multiplication $\mathbb{F} \times V \to V$ such that $(V, +)$ is an abelian group and, for all $a, b \in \mathbb{F}$ and $\mathbf{u}, \mathbf{v} \in V$,

$$a(\mathbf{u} + \mathbf{v}) = a\mathbf{u} + a\mathbf{v}, \quad (a+b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}, \quad (ab)\mathbf{v} = a(b\mathbf{v}), \quad 1\mathbf{v} = \mathbf{v}.$$

Example 3.3 (The space $\mathbb{F}_q^n$)**.** The set of column vectors

$$\mathbb{F}_q^n = \left\{ (v_1, \ldots, v_n)^\top : v_i \in \mathbb{F}_q \right\},$$

with entrywise addition and scalar multiplication, is a vector space over $\mathbb{F}_q$. It is finite: it contains exactly q^n elements. A Kyber secret key lives in $\mathbb{F}_{3329}^{256}$, so it is one of 3329^{256} possible vectors – already far more than the number of atoms in the observable universe.

A typical secret is written

$$\mathbf{s} = \begin{bmatrix} 2 \\ 5 \\ 1 \\ 7 \end{bmatrix} \in \mathbb{F}_q^4, \qquad \text{or in practice} \qquad \mathbf{s} \in \mathbb{F}_{3329}^{256}.$$

3.4.1 Sage experiment: creating a vector

```
F = GF(17)
v = vector(F, [1, 2, 3])
v
```

To inspect its type, use `type(v)`. Sage reports it as a *free module element* rather than a plain list. This is not pedantry: a vector space over a field is the special case of a *module* over a ring in which the ring happens to be a field, and Module-LWE (Chapter 13) will use exactly the more general object.

3.4.2 Vector addition and scalar multiplication

```
a = vector(F, [1, 2, 3])
b = vector(F, [5, 6, 7])
a + b        # (6, 8, 10)
3 * a        # (3, 6, 9)
```

All arithmetic is modulo the field characteristic: over $\mathbb{F}_7$ the sum $6+5$ is 4, not 11. Over $\mathbb{R}$ or $\mathbb{Q}$ addition has the familiar parallelogram picture of Figure 3.1.

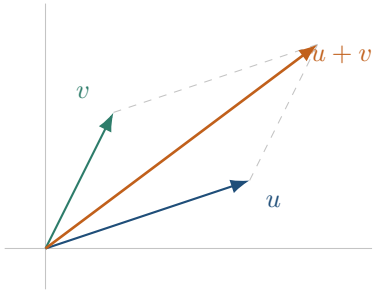


Figure 3.1: Vector addition by the parallelogram rule, for $u=(3,1)$ and $v=(1,2)$. Both vectors are drawn from the origin; completing the parallelogram (dashed sides) shows that the sum $u+v=(4,3)$ is exactly its diagonal. Coordinate-wise, this is the same rule the Sage experiment above computes entrywise: $(1,2,3)+(5,6,7)=(6,8,10)$.

3.5 Linear combinations, independence, and span

The single most important operation in the whole subject is forming a *linear combination*.

Definition 3.4 (Linear combination and span)**.** Given vectors $\mathbf{v}_1,\dots,\mathbf{v}_k \in V$ and scalars $c_1,\dots,c_k \in \mathbb{F}$, the vector $\sum_{i=1}^{k} c_i\mathbf{v}_i$ is a *linear combination* of the $\mathbf{v}_i$. The set of all such combinations is their *span*, written $\operatorname{span}(\mathbf{v}_1,\dots,\mathbf{v}_k)$; it is the smallest subspace of V containing every $\mathbf{v}_i$.

Definition 3.5 (Linear independence)**.** The vectors $\mathbf{v}_1,\dots,\mathbf{v}_k$ are *linearly independent* if the only linear combination equal to the zero vector is the trivial one:

$$\sum_{i=1}^{k} c_i\mathbf{v}_i = \mathbf{0} \quad\Longrightarrow\quad c_1 = \cdots = c_k = 0.$$

Otherwise they are *linearly dependent*, meaning one of them is a linear combination of the others.

For instance $(1,0)$ and $(0,1)$ are independent, whereas $(1,2)$ and $(2,4)$ are dependent, since the second is twice the first.

3.5.1 Sage experiment: detecting dependence through rank

Dependence is detected numerically by *rank*, defined precisely below: stacking the vectors as rows, the rank counts how many are genuinely independent.

```
A = Matrix(GF(17), [[1, 2], [2, 4]])
A.rank()      # 1: only one independent direction
```

3.6 Basis and dimension

Definition 3.6 (Basis)**.** A *basis* of a vector space V is a set of vectors that is both linearly independent and spans V. Equivalently, every $\mathbf{v} \in V$ has a *unique* expression as a linear combination of the basis vectors.

The standard basis of $\mathbb{F}_q^2$ is $\{(1,0),(0,1)\}$, since $(a,b) = a(1,0)+b(0,1)$. But $\{(2,1),(1,1)\}$ is an equally valid basis of the same space: the space is intrinsic, the basis is a chosen coordinate system for it. The number of vectors in a basis is not a matter of choice.

Theorem 3.7 (Dimension is well defined)**.** *Any two bases of a finite-dimensional vector space V have the same number of elements. This common number is the* dimension $\dim V$.

Thus $\dim \mathbb{F}_q^n = n$. The freedom to change basis without changing the space is the seed of the entire theory of lattice reduction (Chapter 8): a lattice, too, has many bases, and finding a *good* one is the central algorithmic problem.

3.6.1 Sage experiment: two bases for one space

```
B1 = Matrix(GF(17), [[1, 0], [0, 1]])
B2 = Matrix(GF(17), [[2, 1], [1, 1]])
# both have rank 2, so each is a basis of GF(17)^2
print(B1.rank(), B2.rank())
```

3.7 Matrices as linear maps

A matrix is best understood not as a grid of numbers but as the coordinate form of a *linear map*.

Definition 3.8 (Linear map)**.** A map $T\colon \mathbb{F}^n \to \mathbb{F}^m$ is *linear* if it respects both operations:

$$T(\mathbf{u}+\mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v}), \qquad T(a\mathbf{v}) = a\,T(\mathbf{v}).$$

Once bases are fixed, every linear map is given by an $m \times n$ matrix A via $T(\mathbf{v}) = A\mathbf{v}$, and every such matrix defines a linear map.

The product $A\mathbf{v}$ is itself a linear combination: it is the combination of the *columns* of A weighted by the entries of $\mathbf{v}$. Concretely, entry i of the result is the inner product of row i of A with $\mathbf{v}$, which is exactly the double loop of Algorithm 2.

```
F = GF(17)
A = Matrix(F, [[1, 2], [3, 4]])
v = vector(F, [5, 6])
A * v
```

Algorithm 2 Matrix–Vector Product

Input: Matrix $A \in \mathbb{F}^{m \times n}$ with entries $A_{i,j}$; vector $v \in \mathbb{F}^n$
Output: $w = Av \in \mathbb{F}^m$

```
for i = 1, ..., m do
    w_i ← 0
    for j = 1, ..., n do
        w_i ← w_i + A_{i,j} v_j
    end for
end for
return w
```

This same loop is foundational for the whole book: it computes $A\mathbf{s}$ in the LWE equation, and in ring form (Chapter 12) it reappears as the schoolbook polynomial multiplication used by RLWE and Kyber.

3.8 Rank, kernel, and image

Three subspaces measure what a linear map $A\colon \mathbb{F}^n \to \mathbb{F}^m$ does.

Definition 3.9 (Image, kernel, rank)**.** The *image* (or column space) is $\operatorname{im} A = \{A\mathbf{v} : \mathbf{v} \in \mathbb{F}^n\} \subseteq \mathbb{F}^m$; its dimension is the *rank* of A. The *kernel* (or null space) is $\ker A = \{\mathbf{v} \in \mathbb{F}^n : A\mathbf{v} = \mathbf{0}\} \subseteq \mathbb{F}^n$; its dimension is the *nullity*.

The image is exactly the set of right-hand sides $\mathbf{b}$ for which $A\mathbf{x} = \mathbf{b}$ has a solution. The kernel measures how far a solution is from unique: if $A\mathbf{x}_0 = \mathbf{b}$, then the full solution set is $\mathbf{x}_0 + \ker A$. These two dimensions are tied together.

Theorem 3.10 (Rank–nullity)**.** *For any linear map $A\colon \mathbb{F}^n \to \mathbb{F}^m$,*

$$\dim(\ker A) + \operatorname{rank}(A) = n.$$

In particular, a square matrix $A \in \mathbb{F}^{n \times n}$ is invertible if and only if its rank is n, equivalently $\ker A = \{\mathbf{0}\}$, equivalently its columns are a basis of $\mathbb{F}^n$. When A is invertible, $A\mathbf{x} = \mathbf{b}$ has exactly one solution for every $\mathbf{b}$.

3.8.1 Sage experiment: rank governs invertibility

```
F = GF(17)
for _ in range(10):
    A = random_matrix(F, 4, 4)
    print(A.rank())
```

For a uniformly random $n \times n$ matrix over $\mathbb{F}_q$, the invertibility probability is

$$\prod_{i=1}^{n} (1 - q^{-i}).$$

Thus the claim depends on q: for this 4×4 example over $\mathbb{F}_{17}$ it is high, while over $\mathbb{F}_2$ it is only about 30.8%.

3.9 Solving linear systems, and the leap to LWE

Given A and $\mathbf{b}$, the equation $A\mathbf{x} = \mathbf{b}$ is solved by *Gaussian elimination*: row operations reduce A to echelon form, and back-substitution recovers $\mathbf{x}$.

Proposition 3.11 (Linear systems are easy)**.** *Over a field, Gaussian elimination solves an $n \times n$ system $A\mathbf{x} = \mathbf{b}$ (or reports that no solution exists) using $O(n^3)$ field operations. If A is invertible the solution is unique.*

This efficiency is precisely why a plain linear system carries *no* cryptographic value: anyone who sees A and $\mathbf{b} = A\mathbf{s}$ can recover the secret $\mathbf{s}$ in cubic time. LWE changes the setting. Given a public, typically rectangular random matrix $A \in \mathbb{Z}_q^{m \times n}$ with $m > n$, a secret $\mathbf{s}$ drawn from a prescribed small distribution, and an independent small error vector $\mathbf{e}$, one observes samples

$$\mathbf{b} = A\mathbf{s} + \mathbf{e},$$

with arithmetic modulo q. Unlike an invertible square system, this overdetermined system will generally have no exact solution for a nonzero random error. Gaussian elimination can detect inconsistency, but it does not recover the planted secret from the noisy samples. Recovering $\mathbf{s}$ is the *Learning With Errors* problem (Chapter 11).

Remark. It is worth pausing on how little changes on the page and how much changes in difficulty. For an invertible square matrix, $\mathbf{s} \mapsto A\mathbf{s}$ is easy to invert. With many noisy samples, the observed value also depends on independently sampled error; cryptographic hardness lives in distinguishing and exploiting that structure.

3.10 Experiments

```
F = GF(17)
# 1. random vectors
V = VectorSpace(F, 4)
for _ in range(5): print(V.random_element())

# 2. a matrix acting on a vector
A = random_matrix(F, 6, 4); v = random_vector(F, 4)
print(A * v)

# 3. exact solve is easy; try recovering s from b = A*s
s = random_vector(F, 4); b = A * s
print(A.solve_right(b) == s)   # True when A has full column rank

# 4. perturb an overdetermined system: it normally becomes inconsistent
e = vector(F, [1, 0, 0, 16, 0, 0])   # 16 == -1 mod 17
try:
    A.solve_right(b + e)
except ValueError:
    print("no exact solution: noisy samples are inconsistent")
```

The experiment is a toy illustration only: real LWE specifies the dimensions and the secret and error distributions precisely.

3.11 Connection to the next chapter

The next chapter, *Polynomial Rings*, keeps the linear-algebraic viewpoint but changes the scalars-and-vectors picture into a single richer object. A structured (circulant) matrix will be encoded by one polynomial, matrix–vector multiplication will become polynomial multiplication, and the vector space $\mathbb{F}_q^n$ will be upgraded to the quotient ring

$$R_q = \mathbb{Z}_q[x]/(x^n + 1),$$

the natural home of RLWE and Kyber.

Chapter 4

Polynomial Rings

4.1 Why this chapter matters

The previous chapter expressed cryptographic objects as vectors and matrices over a field. This chapter changes the algebra: instead of a vector space $\mathbb{F}_q^n$ we work in a *quotient polynomial ring*

$$R_q = \mathbb{Z}_q[x]/(x^n + 1),$$

the setting of RLWE, Kyber, and Dilithium. The payoff is structural. A single element of R_q carries the same information as a whole structured $n \times n$ matrix, and multiplication in R_q is a form of convolution that admits a fast transform (the NTT of Chapter 6). To use this ring safely we need to be precise about what it is: what a ring is, what a quotient by a polynomial means, and why the particular modulus $x^n + 1$ is chosen.

4.2 Learning goals

After reading this chapter, you should be able to:

- state what a (commutative) ring is and recognise $\mathbb{F}_q[x]$ as one;
- perform Euclidean division of polynomials and explain why it makes quotient rings well defined;
- construct the quotient ring $\mathbb{F}_q[x]/(f)$ and count its elements;
- state when the quotient is a field, and why x^n+1 (for n a power of two) is the natural modulus;
- describe multiplication in R_q as negacyclic convolution and connect it to the NTT.

4.3 Rings and fields

A field, from Chapter 3, supports addition, subtraction, multiplication, *and* division. A ring is the weaker structure in which division may fail.

Definition 4.1 (Commutative ring)**.** A *commutative ring with identity* is a set R with operations $+$ and $\cdot$ such that $(R, +)$ is an abelian group with identity 0, multiplication is associative and commutative with an identity 1, and multiplication distributes over addition. A *field* is the special case in which every nonzero element has a multiplicative inverse.

The integers $\mathbb{Z}$ form a ring but not a field (2 has no integer inverse); the integers modulo a prime, $\mathbb{F}_q$, form a field. The distinction matters here because we will build new rings out of polynomials, and those are generally not fields.

4.4 The polynomial ring $\mathbb{F}_q[x]$

Definition 4.2 (Polynomial ring)**.** For a field $\mathbb{F}$, the *polynomial ring* $\mathbb{F}[x]$ consists of all finite expressions

$$f(x) = a_0 + a_1 x + \cdots + a_d x^d, \qquad a_i \in \mathbb{F},$$

with addition and multiplication of polynomials. The *degree* $\deg f$ is the largest i with $a_i \neq 0$. It is a commutative ring with identity 1.

A polynomial is, coefficient by coefficient, just a vector: $f(x) = 3x^4 + 2x^2 + 5$ is the coefficient list $(5, 0, 2, 0, 3)$. Addition of polynomials is addition of these coefficient vectors. The new ingredient is *multiplication*, and it is not entrywise.

Proposition 4.3 (Multiplication is convolution)**.** *If $f(x) = \sum_i a_i x^i$ and $g(x) = \sum_j b_j x^j$, then*

$$(fg)(x) = \sum_k c_k x^k, \qquad c_k = \sum_{i+j=k} a_i b_j.$$

The coefficient sequence (c_k) is the convolution *of (a_i) and (b_j).*

For example $(x+1)(x+2) = x^2 + 3x + 2$, where the middle coefficient $3 = 1 \cdot 2 + 1 \cdot 1$ is a sum of cross terms. This convolution is the operation that the NTT will later accelerate.

4.4.1 Sage experiment: the polynomial ring

```
F = GF(17)
R.<x> = PolynomialRing(F)
f = 3*x^4 + 2*x^2 + 5
print(f.degree())        # 4
print(f.list())          # [5, 0, 2, 0, 3]: the coefficient vector
print((x+1)*(x+2))       # x^2 + 3*x + 2
```

4.5 Euclidean division

The one structural fact that makes everything below work is that polynomials over a field can be divided with remainder, exactly like integers.

Theorem 4.4 (Division algorithm in $\mathbb{F}[x]$)**.** *For $f, g \in \mathbb{F}[x]$ with $g \neq 0$ there exist* unique *polynomials q (quotient) and r (remainder) with*

$$f = q\,g + r, \qquad \deg r < \deg g.$$

Uniqueness of the remainder is what will let us reduce every polynomial modulo a fixed g to a single canonical form of degree below $\deg g$. That canonical form is the coefficient vector we actually store.

4.6 Quotient rings

Left unbounded, degrees grow without limit under multiplication. We tame this by declaring a fixed polynomial to be zero.

Definition 4.5 (Ideal and quotient ring)**.** The *principal ideal* generated by $f \in \mathbb{F}[x]$ is the set of all multiples $(f) = \{f \cdot h : h \in \mathbb{F}[x]\}$. The *quotient ring* $\mathbb{F}[x]/(f)$ has as its elements the residue classes of polynomials modulo f; two polynomials are identified when they differ by a multiple of f. Arithmetic is done as in $\mathbb{F}[x]$ and then reduced modulo f.

By Euclidean division, every class has a unique representative of degree less than $\deg f$, so the quotient is not some abstract fog: it is a concrete, finite set of low-degree polynomials.

Proposition 4.6 (Size of the quotient)**.** *If $\mathbb{F} = \mathbb{F}_q$ and $\deg f = n$, then $\mathbb{F}_q[x]/(f)$ has exactly q^n elements, one for each coefficient vector $(c_0, \ldots, c_{n-1}) \in \mathbb{F}_q^n$.*

So as a set, $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ is just $\mathbb{F}_q^n$; what the ring structure adds on top of the vector space is a *multiplication*.

4.6.1 Sage experiment: building a quotient ring

```
R.<x> = PolynomialRing(GF(17))
S.<y> = R.quotient(x^4 + 1)
print(y^4)     # 16 == -1 mod 17: the relation x^4 = -1
print(y^5)     # 16*y == -y
```

4.7 When is the quotient a field?

Whether $\mathbb{F}[x]/(f)$ is a field is governed entirely by whether f can be factored.

Theorem 4.7 (Irreducibility and fields)**.** *$\mathbb{F}[x]/(f)$ is a field if and only if f is irreducible over $\mathbb{F}$. If f factors non-trivially, the quotient has zero divisors and is only a ring.*

This is the polynomial analogue of "$\mathbb{Z}/m\mathbb{Z}$ is a field iff m is prime." It will matter twice. Over $\mathbb{Q}$ the modulus $x^n + 1$ is irreducible (below), which is what gives the associated number field its clean structure; but over $\mathbb{Z}_q$ the *same* polynomial deliberately factors, and that factorisation is precisely what the NTT exploits.

4.8 The modulus $x^n + 1$ and cyclotomic polynomials

Modern lattice schemes fix n to be a power of two and work modulo $x^n + 1$. This choice is not cosmetic.

Definition 4.8 (Cyclotomic polynomial)**.** The m-th *cyclotomic polynomial* $\Phi_m(x)$ is the minimal polynomial over $\mathbb{Q}$ of a primitive m-th root of unity; its roots are exactly the primitive m-th roots of unity.

Proposition 4.9. *For $n = 2^k$, one has $x^n + 1 = \Phi_{2n}(x)$, and it is irreducible over $\mathbb{Q}$.*

Thus $\mathbb{Z}[x]/(x^n + 1)$ is the ring of integers of the cyclotomic field $\mathbb{Q}(\zeta_{2n})$ – a highly structured, well-studied object, which is the deeper reason it supports both strong security reductions and fast arithmetic. Working modulo a power-of-two degree also makes the reduction step trivial to implement, as the next section shows.

4.8.1 Negacyclic wrap-around

The relation $x^n = -1$ has a very concrete effect on the coefficient vector. Multiplying by x shifts every coefficient one slot to the right, and the coefficient that would fall off the top wraps back into slot 0 with its sign flipped.

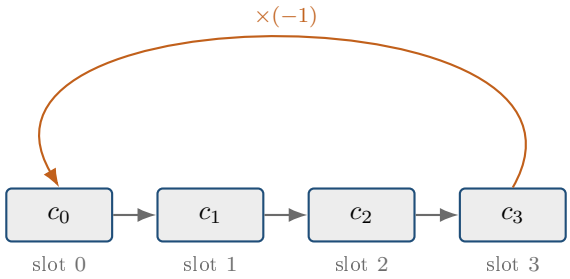


Figure 4.1: Multiplying $c(x) = c_0 + c_1x + c_2x^2 + c_3x^3$ by x inside $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ shifts every coefficient one slot to the right, shown here for $n = 4$ (blue arrows). The coefficient that falls off the top does not simply reappear: it comes back into slot 0 *negated* (orange arrow), because the defining relation of the ring is $x^n = -1$. Concretely, $x \cdot c(x) = -c_3 + c_0x + c_1x^2 + c_2x^3$. This negacyclic wrap-around, rather than a plain cyclic shift, is the single most important structural fact about ring-based lattice cryptography, and it is exactly the sign flip implemented in Algorithm 3.

Carrying the sign flip through a full product gives the *negacyclic convolution*: the coefficients of $c(x) = a(x)b(x) \bmod (x^n + 1)$ are

$$c_k = \sum_{i+j=k} a_i b_j \; - \sum_{i+j=k+n} a_i b_j,$$

the ordinary convolution minus the part that has wrapped past degree $n - 1$. Algorithm 3 is the direct schoolbook realisation.

Algorithm 3 Polynomial Multiplication modulo $x^n + 1$

Input: Coefficient arrays $a_0, \ldots, a_{n-1}$ and $b_0, \ldots, b_{n-1}$ of $a(x), b(x) \in R_q = \mathbb{Z}_q[x]/(x^n + 1)$
Output: Coefficients $c_0, \ldots, c_{n-1}$ of $c(x) = a(x)b(x) \bmod (x^n + 1)$

$c_0, \ldots, c_{n-1} \leftarrow 0, \ldots, 0$
for $i = 0, \ldots, n - 1$ **do**
  for $j = 0, \ldots, n - 1$ **do**
    $k \leftarrow i + j$
    if $k < n$ **then**
      $c_k \leftarrow c_k + a_i b_j$ ▷ ordinary term, degree stays below n
    else
      $c_{k-n} \leftarrow c_{k-n} - a_i b_j$ ▷ wraps past degree $n - 1$: negated, since $x^n = -1$
    end if
  end for
end for
return $(c_0, \ldots, c_{n-1}) \bmod q$

4.9 Splitting modulo q: the door to the NTT

Over $\mathbb{Q}$ the modulus is irreducible, but over $\mathbb{Z}_q$ it need not be, and we *want* it to factor.

Proposition 4.10 (Full splitting). *If $2n \mid q-1$, then $\mathbb{F}_q$ contains a primitive $2n$-th root of unity, x^n+1 splits into n distinct linear factors over $\mathbb{F}_q$, and the Chinese Remainder Theorem gives a ring isomorphism*

$$\mathbb{Z}_q[x]/(x^n+1) \;\cong\; \mathbb{F}_q \times \cdots \times \mathbb{F}_q \quad (n \text{ copies}).$$

Under this isomorphism, multiplication of polynomials becomes coordinatewise multiplication of n scalars – an $O(n)$ operation once the map is applied. The map itself, and its inverse, are computed in $O(n \log n)$ by the NTT (Chapter 6); that is the entire reason the NTT exists.

Remark (Kyber's partial split). Kyber uses $q = 3329$ and $n = 256$, and $q - 1 = 3328 = 2^8 \cdot 13$. Here $n = 2^8$ divides $q-1$ but $2n = 2^9$ does not, so $\mathbb{F}_q$ has a primitive 256-th root of unity but no primitive 512-th root. Consequently $x^{256}+1$ does not split completely; it factors into 128 irreducible *quadratics*, and Kyber's NTT maps R_q to 128 copies of $\mathbb{F}_q[x]/(\text{quadratic})$ rather than to 256 scalars. The principle is identical – CRT turns one multiplication into many independent small ones – only the block size differs.

4.10 RLWE as structured LWE

With the ring in hand, the transition from LWE to RLWE is a change of alphabet:

$$\underbrace{A\mathbf{s} + \mathbf{e} = \mathbf{b}}_{\text{LWE, over } \mathbb{F}_q^n} \qquad \longrightarrow \qquad \underbrace{a(x)\, s(x) + e(x) = b(x)}_{\text{RLWE, over } R_q}.$$

The random matrix A is replaced by structured polynomial multiplication, because one polynomial encodes an entire negacyclic linear map; the secret and error become polynomials. This can substantially reduce storage relative to an unstructured matrix, but there is no parameter-independent "factor of n" size theorem: a comparison depends on dimensions, sample counts, moduli, security targets, and encoding. The precise RLWE construction is the subject of Chapter 12.

4.11 Experiments

```
R.<x> = PolynomialRing(GF(17))
S.<y> = R.quotient(x^4 + 1)

# 1. the quotient relation and its powers
print(y^4, y^5, y^8)          # -1, -y, 1

# 2. negacyclic wrap-around by hand: multiply by y
c = 2 + 3*y + 4*y^2 + 5*y^3
print(y * c)                   # top coeff returns to slot 0 negated

# 3. a quotient element is a length-4 coefficient vector
print((3*y^3 + y + 2).lift().list())
```

4.12 Summary

This chapter built the ring

$$R_q = \mathbb{Z}_q[x]/(x^n + 1)$$

from the ground up: a polynomial ring $\mathbb{F}_q[x]$, made finite by quotienting out the ideal $(x^n + 1)$, whose elements are length-n coefficient vectors and whose multiplication is negacyclic convolution. Choosing n a power of two makes the modulus a cyclotomic polynomial with excellent structure, and choosing q so that the modulus splits mod q opens the door to the NTT. The next chapter, *Lattices*, turns from this algebraic view to the geometric one, where the hardness of the underlying problems actually lives.

Chapter 5

From Convolution to Negacyclic Ring Arithmetic

5.1 Why this chapter matters

Polynomial multiplication is the linear operation underlying RLWE and ML-KEM. The important distinction is the modulus: reduction modulo $x^n - 1$ gives *cyclic* convolution and ordinary circulant matrices, while reduction modulo $x^n + 1$ gives *negacyclic* convolution. ML-KEM uses the latter.

5.2 Two kinds of wrap-around

Let $a(x) = \sum_{i=0}^{n-1} a_i x^i$ and $s(x) = \sum_{j=0}^{n-1} s_j x^j$. Before reduction, the coefficient of x^k is

$$\sum_{i+j=k} a_i s_j.$$

For example,

$$(1 + 2x + 3x^2)(4 + 5x) = 4 + 13x + 22x^2 + 15x^3,$$

where $22 = 2 \cdot 5 + 3 \cdot 4$. This indexing condition, $i + j = k$, is the ordinary convolution rule.

Now choose a quotient modulus.

- In $\mathbb{F}_q[x]/(x^n - 1)$, $x^n = 1$. A term that wraps past degree $n - 1$ returns with the *same* sign. This is cyclic convolution.
- In $\mathbb{F}_q[x]/(x^n + 1)$, $x^n = -1$. A wrapped term returns with the *opposite* sign. This is negacyclic convolution.

Thus the two structures are related but not interchangeable. The quotient ring used by ML-KEM is

$$R_q = \mathbb{Z}_{3329}[x]/(x^{256} + 1),$$

so its polynomial products are negacyclic.

5.2.1 A four-coefficient hand calculation

Take $n = 4$ and multiply $a(x) = 1 + 2x + 3x^2 + 4x^3$ by $s(x) = 5 + 6x + 7x^2 + 8x^3$. Before reduction, the product has degree six. The low-degree coefficients are the same in both quotient rings:

$$[x^0](as) = 1 \cdot 5 = 5, \qquad [x^1](as) = 1 \cdot 6 + 2 \cdot 5 = 16,$$
$$[x^2](as) = 1 \cdot 7 + 2 \cdot 6 + 3 \cdot 5 = 34, \qquad [x^3](as) = 1 \cdot 8 + 2 \cdot 7 + 3 \cdot 6 + 4 \cdot 5 = 60.$$

The remaining terms are $53x^4 + 52x^5 + 32x^6$. Modulo $x^4 - 1$, they return as $+53, +52x, +32x^2$; modulo $x^4 + 1$, they return as $-53, -52x, -32x^2$. Hence

$$\begin{aligned} as \bmod (x^4 - 1) &= 58 + 68x + 66x^2 + 60x^3, \\ as \bmod (x^4 + 1) &= -48 - 36x + 2x^2 + 60x^3. \end{aligned}$$

The inputs are identical. The sign of the wrap-around is the only change, but it changes the ring operation.

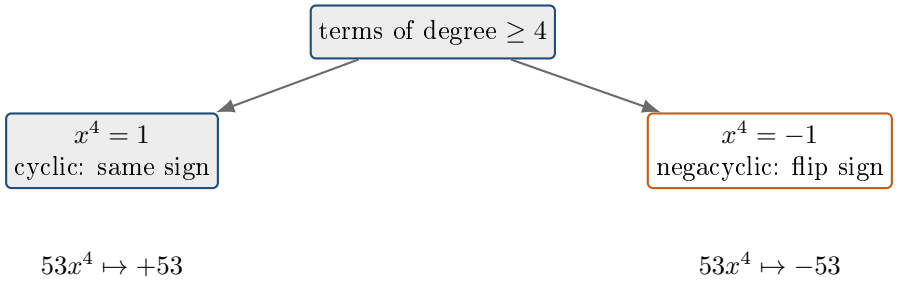


Figure 5.1: One wrapped term under the two quotient relations. ML-KEM follows the negacyclic branch.

5.3 Matrix representations

Multiplication by $a(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ is a linear map on coefficient vectors. In the cyclic case it is represented by the circulant matrix

$$C(a)_{i,j} = a_{(i-j) \bmod n}.$$

In the negacyclic case, the same wrapped coefficient receives a minus sign:

$$N(a)_{i,j} = \begin{cases} a_{i-j}, & i \geq j, \\ -a_{n+i-j}, & i < j. \end{cases}$$

For $n = 4$, multiplication by $a_0 + a_1x + a_2x^2 + a_3x^3$ modulo $x^4 + 1$ has matrix

$$N(a) = \begin{bmatrix} a_0 & -a_3 & -a_2 & -a_1 \\ a_1 & a_0 & -a_3 & -a_2 \\ a_2 & a_1 & a_0 & -a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix}.$$

It is therefore inaccurate to call ML-KEM multiplication ordinary circulant multiplication. A polynomial still compactly represents a structured linear map, which is the useful connection to linear algebra.

5.3.1 Reading the negacyclic matrix

The first column of $N(a)$ is the coefficient vector of $a(x)$. Moving one column right shifts the entries down, but the coefficient crossing the top boundary is negated. This is precisely $x^4 = -1$ written as a matrix. An ordinary circulant matrix has no minus signs in its upper-right corner and represents multiplication modulo $x^4 - 1$.

Quotient ring	Wrap-around rule	Matrix structure
$\mathbb{F}_q[x]/(x^n - 1)$	$x^n = 1$	circulant
$\mathbb{F}_q[x]/(x^n + 1)$	$x^n = -1$	negacyclic / signed circulant

Table 5.1: The quotient relation determines both convolution and matrix structure.

5.4 A reproducible Sage experiment

The following experiment uses one field, one input vector, and the negacyclic matrix on both sides of the comparison.

```
q = 17
F = GF(q)
a = vector(F, [1, 2, 3, 4])
s = vector(F, [5, 6, 7, 8])

# Matrix for multiplication by a(x) modulo x^4 + 1.
N = Matrix(F, [[a[(i-j) % 4] if i >= j else -a[(i-j) % 4]
                for j in range(4)] for i in range(4)])

R.<x> = PolynomialRing(F)
Rq = R.quotient(x^4 + 1, 'y')
y = Rq.gen()
ap = sum(a[i]*y^i for i in range(4))
sp = sum(s[i]*y^i for i in range(4))

matrix_product = N * s
ring_product = vector(F, (ap*sp).lift().list() + [0]*4)[:4]
print(matrix_product)
print(ring_product)
print(matrix_product == ring_product)  # True
```

The sign in the upper-right part of N is the whole difference between this experiment and a circulant one. Changing $x^4 + 1$ to $x^4 - 1$ and removing those signs produces the cyclic version.

5.4.1 A cyclic comparison

It is useful to see the conventional circulant case too, provided it is labelled correctly. This version changes both the matrix and the quotient modulus to $x^4 - 1$:

```
C = Matrix(F, [[a[(i-j) % 4] for j in range(4)] for i in range(4)])
Rc = R.quotient(x^4 - 1, 'z')
z = Rc.gen()
ac = sum(a[i]*z^i for i in range(4))
sc = sum(s[i]*z^i for i in range(4))
cyclic_ring_product = vector(F, (ac*sc).lift().list() + [0]*4)[:4]
print(C * s == cyclic_ring_product)  # True
```

The two experiments are deliberately parallel. Changing the field, input vector, or quotient modulus would no longer test a mathematical equivalence.

5.5 What the NTT accelerates

Transforms can convert a convolution into multiplication in a different representation, but the representation depends on the quotient ring and on the available roots of unity. A full cyclic NTT diagonalizes multiplication modulo $x^n - 1$ when an n-th root is available. A full negacyclic NTT that evaluates at n scalar points needs a primitive $2n$-th root. ML-KEM does not have such a root for $n = 256$ and $q = 3329$; its partial NTT instead produces 128 quadratic components. Chapter 6 develops that construction.

5.6 From one polynomial to a module

RLWE uses ring elements, whereas Module-LWE uses vectors and matrices whose entries lie in R_q. For a $k \times k$ polynomial matrix A and a polynomial vector $\mathbf{s}$,

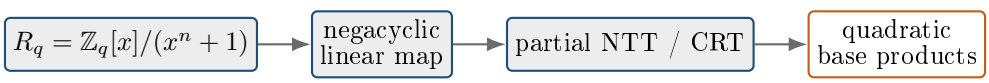
$$(A\mathbf{s})_i = \sum_{j=1}^{k} A_{i,j}(x)s_j(x) \quad \text{in } R_q.$$

Each scalar-looking product is still negacyclic polynomial multiplication. The module structure adds a small matrix layer; it does not turn the ring operation back into ordinary cyclic convolution.

5.7 Checks before moving on

1. Which quotient relation makes a coefficient wrap back with a minus sign?
2. Why is the upper-right corner of a negacyclic multiplication matrix signed?
3. Why is it invalid to compare a circulant integer-matrix product with a product in $\mathbb{F}_{17}[x]/(x^4 + 1)$?
4. What root-of-unity condition would be needed to represent negacyclic products by scalar evaluations?

5.8 Summary



The ordinary circulant/cyclic-convolution picture is useful background, but ML-KEM follows the negacyclic path shown above.

Chapter 6

Number Theoretic Transform (NTT)

6.1 Purpose and scope

The NTT is a finite-field analogue of a discrete Fourier transform. It changes representation so that structured polynomial multiplication becomes much cheaper. It is an integral part of ML-KEM's specified arithmetic, not merely an optional optimization [1].

This chapter distinguishes three settings that are often conflated:

1. a cyclic NTT for $\mathbb{F}_q[x]/(x^n - 1)$;
2. a full scalar negacyclic transform for $\mathbb{F}_q[x]/(x^n + 1)$, when a primitive $2n$-th root exists; and
3. ML-KEM's partial NTT, whose output consists of 128 degree-one residues modulo quadratic polynomials.

6.2 Roots of unity and the cyclic transform

If $\omega \in \mathbb{F}_q$ has multiplicative order n, then evaluation at

$$1, \omega, \omega^2, \ldots, \omega^{n-1}$$

identifies multiplication modulo $x^n - 1$ with n scalar products. This requires $n \mid (q - 1)$. The standard radix-2 butterfly network implements this map in $O(n \log n)$ operations when n is a power of two.

For the negacyclic quotient x^n+1, scalar evaluation points must be roots of x^n+1, so a primitive $2n$-th root is required. The difference is structural, not cosmetic: an algorithm written only for an n-th root and cyclic convolution is not automatically an ML-KEM algorithm.

6.2.1 Finding a root in a toy field

The nonzero elements of $\mathbb{F}_{17}$ form a group of order 16. If g is a generator, then g^2 has order 8 and can serve as ω for an 8-point cyclic NTT. SageMath can check that condition directly:

```
F = GF(17)
g = F.multiplicative_generator()
omega = g^2
print(omega.multiplicative_order())  # 8
```

The important test is the multiplicative order, not merely that an element happens to satisfy an isolated power equation. For a coefficient vector $(a_0, \ldots, a_{n-1})$, the cyclic transform records

$$\widehat{a}_i = a(\omega^i) = \sum_{j=0}^{n-1} a_j \omega^{ij}, \qquad 0 \le i < n.$$

The inverse uses ω^{-1} and multiplies by n^{-1}, so an NTT is a reversible change of representation, not a lossy sampling operation.

6.3 A complete toy cyclic NTT

The following SageMath program performs a complete forward transform, pointwise product, and inverse transform in the cyclic ring $\mathbb{F}_{17}[x]/(x^8 - 1)$. It is intentionally a toy model, not ML-KEM.

```
q = 17
F = GF(q)
n = 8
g = F.multiplicative_generator()
omega = g^((q - 1)//n)
assert omega.multiplicative_order() == n

def ntt(a):
    return [sum(a[j] * omega^(i*j) for j in range(n)) for i in range(n)]

def intt(A):
    n_inv = F(n)^(-1)
    return [n_inv * sum(A[i] * omega^(-i*j) for i in range(n))
            for j in range(n)]

a = vector(F, [1, 2, 3, 0, 0, 0, 0, 0])
b = vector(F, [4, 5, 0, 0, 0, 0, 0, 0])
A, B = ntt(a), ntt(b)
c = vector(F, intt([A[i]*B[i] for i in range(n)]))
print(c)                 # (4, 13, 5, 15, 0, 0, 0, 0)  (22 = 5 in GF(17))
print(intt(ntt(a)) == list(a))  # True
```

The code evaluates at powers of the actual primitive root `omega`, uses an inverse transform, and checks reconstruction. It therefore demonstrates an NTT, unlike evaluation at the unrelated points $0, 1, \ldots, n-1$.

6.3.1 Following one product through the transform

For the input polynomials $a(x) = 1 + 2x + 3x^2$ and $b(x) = 4 + 5x$, direct multiplication gives $4 + 13x + 22x^2 + 15x^3$. The program computes the same result in four conceptual steps:

1. evaluate a at $1, \omega, \ldots, \omega^7$;
2. evaluate b at the same points;
3. multiply the eight pairs of values; and
4. apply the inverse transform to obtain $(4, 13, 22, 15, 0, 0, 0, 0)$.

Because the toy ring is modulo $x^8 - 1$ and the product has degree below eight, no visible wrap-around occurs. Higher-degree terms would wrap cyclically, not negacyclically.

6.4 Butterflies and complexity

The direct code is intentionally transparent, but it evaluates every output from every input and therefore takes $O(n^2)$ field operations. A practical radix-2 NTT uses butterflies to reuse work. A butterfly combines two values u and v using a twiddle factor ω^k:

$$(u, v) \longmapsto (u + \omega^k v,\ u - \omega^k v).$$

There are $\log_2 n$ stages and $O(n)$ work per stage, giving $O(n \log n)$ operations.

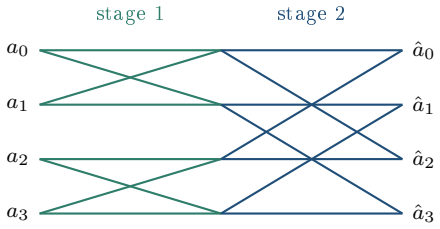


Figure 6.1: A conceptual four-point radix-2 butterfly network. The picture shows the reuse pattern, not ML-KEM's complete partial NTT.

6.5 Why ML-KEM is different

ML-KEM uses $q = 3329$ and $n = 256$. Since

$$q - 1 = 3328 = 2^8 \cdot 13,$$

there are primitive 256-th roots in $\mathbb{Z}_{3329}$, but no primitive 512-th roots. In particular, $x^{256} + 1$ does not split into 256 linear factors. With $\zeta = 17$, a primitive 256-th root, FIPS 203 gives the factorization

$$x^{256} + 1 = \prod_{i=0}^{127} \left(x^2 - \zeta^{2\,\mathrm{BitRev}_7(i)+1} \right).$$

Consequently, the ML-KEM NTT is an isomorphism from

$$R_q = \mathbb{Z}_{3329}[x]/(x^{256} + 1)$$

to a product of 128 quadratic residue rings, not to 256 scalar evaluation values [1, Sec. 4.3].

6.5.1 The three-level mental model

Setting	Required roots / factorisation	Multiplication after transform
Cyclic NTT, $x^n - 1$	primitive n-th root	n scalar products
Full negacyclic NTT, $x^n + 1$	primitive $2n$-th root	n scalar products at roots of x^n+1
ML-KEM, $x^{256} + 1$ over $\mathbb{Z}_{3329}$	128 quadratic factors	128 quadratic base products

Table 6.1: Related transforms with different algebraic targets. Only the third row describes ML-KEM.

6.6 Multiplication in the ML-KEM NTT domain

Each NTT-domain coordinate is represented by two coefficients, $a_0 + a_1X$, reduced modulo $X^2 - \gamma$. For two such residues, multiplication is

$$(a_0 + a_1X)(b_0 + b_1X) \bmod (X^2 - \gamma) = (a_0b_0 + a_1b_1\gamma) + (a_0b_1 + a_1b_0)X.$$

FIPS 203's `MultiplyNTTs` applies this base-case multiplication to each of the 128 coordinate pairs. Thus "pointwise multiplication" is accurate only if each point is understood as a quadratic component; it is not 256 independent scalar multiplications.

Algorithm 4 ML-KEM NTT-domain multiplication (conceptual form)

Input: NTT representations $\hat{f}, \hat{g} \in \mathbb{Z}_q^{256}$
Output: $\hat{h} = \hat{f} \times_{T_q} \hat{g}$

1: **for** $i = 0$ to 127 **do**
2: $\quad \gamma \leftarrow \zeta^{2\,\mathrm{BitRev}_7(i)+1}$
3: $\quad \hat{h}_{2i} \leftarrow \hat{f}_{2i}\hat{g}_{2i} + \gamma\hat{f}_{2i+1}\hat{g}_{2i+1} \pmod q$
4: $\quad \hat{h}_{2i+1} \leftarrow \hat{f}_{2i}\hat{g}_{2i+1} + \hat{f}_{2i+1}\hat{g}_{2i} \pmod q$
5: **end for**
6: **return** $\hat{h}$

The forward and inverse transforms themselves use butterfly networks and precomputed twiddle factors, as specified by FIPS 203. Their implementation details are deliberately not reconstructed here from a generic radix-2 routine: a conforming implementation must follow the standard's algorithms, bounds, and test vectors.

6.7 Practical implementation checklist

Before treating an NTT implementation as an ML-KEM implementation, verify all of the following.

- The coefficient ring is $\mathbb{Z}_{3329}[x]/(x^{256} + 1)$, not a cyclic substitute.
- The forward and inverse transforms follow the specified ordering and twiddle factors.
- NTT-domain multiplication operates on 128 pairs with the appropriate quadratic modulus.
- Modular reduction, serialization, sampling, and bounds follow FIPS 203 in addition to the transform itself.
- The implementation is checked against official known-answer tests and is engineered for its required side-channel model.

This checklist explains why a short classroom transform is valuable for intuition but must not be described as a complete implementation of ML-KEM.

6.8 Summary

An NTT is not simply "evaluate a polynomial at several integers." It is a structured transform at roots of unity, together with its inverse. A cyclic transform uses $x^n - 1$; a full scalar negacyclic transform needs a primitive $2n$-th root. ML-KEM uses neither simplified picture directly: its partial NTT maps R_q to 128 quadratic components and multiplies them with base-case quadratic products.

Part II

Lattice-Based Cryptography

Lattices and Hard Problems

Chapter 7

Lattices

7.1 Why this chapter matters

This chapter is one of the most important in the whole course. If the previous chapters introduced vectors, matrices, and polynomial rings, then this chapter explains the geometric object that makes post-quantum cryptography possible: the lattice. Lattices are the core structure behind LWE, RLWE, Kyber, Dilithium, and many other modern constructions.

7.2 Learning goals

After reading this chapter, you should be able to:

- explain what a lattice is;
- distinguish a lattice from a vector space;
- understand why a lattice can have many different bases;
- understand the difference between good and bad bases;
- appreciate why problems such as SVP and CVP are difficult;
- see the connection between lattice problems and LWE.

7.3 Starting from two dimensions

Instead of jumping immediately to high-dimensional theory, begin with a simple two-dimensional example. Let

$$b_1 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \qquad b_2 = \begin{pmatrix} 1 \\ 3 \end{pmatrix}.$$

Now allow only integer coefficients:

$$v = xb_1 + yb_2, \qquad x, y \in \mathbb{Z}.$$

If we take many such combinations, the resulting points form an infinite regular pattern of discrete points in the plane. This set is called a lattice.

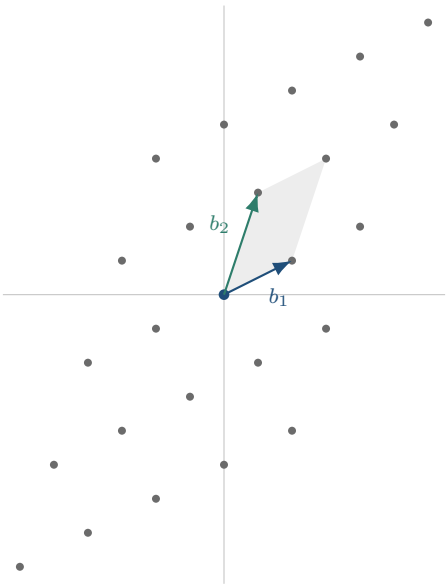


Figure 7.1: The lattice $L(b_1, b_2)$ generated by all integer combinations $xb_1 + yb_2$ of $b_1 = (2, 1)$ and $b_2 = (1, 3)$. The shaded parallelogram is the fundamental domain spanned by the basis.

7.3.1 Sage experiment: generate a two-dimensional lattice

```
b1 = vector(ZZ, [2, 1])
b2 = vector(ZZ, [1, 3])
points = []
for i in range(-5, 6):
    for j in range(-5, 6):
        points.append(i * b1 + j * b2)
points[:10]
```

The output is a list of lattice points. The important observation is that they are arranged in a very regular, discrete pattern.

7.4 Formal definition

Formally, if $b_1, \ldots, b_n$ are linearly independent vectors, then the set

$$L(B) = \left\{ \sum_{i=1}^{n} z_i b_i \ : \ z_i \in \mathbb{Z} \right\}$$

is called the lattice generated by the basis $B = \{b_1, \ldots, b_n\}$.

The key difference from a vector space is that the coefficients are restricted to integers rather than arbitrary real numbers.

This definition is directly executable: enumerating a lattice inside a bounded region is exactly a bounded search over integer coefficient vectors.

Algorithm 5 Bounded Lattice-Point Enumeration

Input: Basis $b_1, \ldots, b_n \in \mathbb{Z}^n$; coefficient bound $N \in \mathbb{Z}_{>0}$
Output: The set S of lattice points $\sum_i z_i b_i$ with every $|z_i| \le N$

1: $S \leftarrow \emptyset$
2: **for** each $(z_1, \ldots, z_n) \in \{-N, \ldots, N\}^n$ **do**
3: $\quad v \leftarrow \sum_{i=1}^{n} z_i\, b_i$
4: $\quad S \leftarrow S \cup \{v\}$
5: **end for**
6: **return** S

This is precisely what the Sage experiment above does for $n = 2$ and $N = 5$: two nested loops over integer coefficients, collecting every resulting point. The regular grid in Figure 7.1 is the output of this algorithm.

7.5 Why a lattice is not a vector space

A vector space allows arbitrary real or complex coefficients. For example,

$$0.13b_1 + 2.78b_2$$

is a perfectly valid element of the vector space spanned by b_1 and b_2. But in a lattice, the coefficients must be integers. That is why the lattice is a discrete structure rather than a continuous one.

This distinction is crucial. Cryptography needs discrete problems because continuous problems are often easy to solve.

7.6 Vector space versus lattice

A vector space is continuous and dense. A lattice is discrete and sparse. In other words:

- vector space: all linear combinations with real coefficients;
- lattice: only integer linear combinations.

This difference is exactly why lattice problems are hard and useful for cryptography.

7.7 A lattice can have many bases

The most important conceptual point is that a lattice can be described by many different bases. For example, the basis

$$\{(1, 0), (0, 1)\}$$

and the basis

$$\{(2, 1), (1, 1)\}$$

can generate the same lattice.

This is similar to how a city can be described using different coordinate systems. The city does not change, only the coordinate description does.

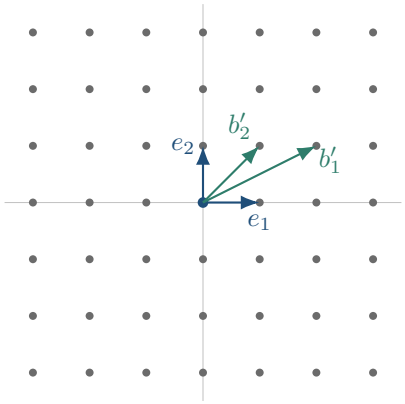


Figure 7.2: Two different bases for the same lattice $\mathbb{Z}^2$: the standard basis $\{e_1, e_2\}$ in blue and the skewed basis $\{b'_1, b'_2\}$ in teal. Both reach every point of the same underlying grid, because $\det \left(\begin{smallmatrix} 2 & 1 \\ 1 & 1 \end{smallmatrix}\right) = 1$.

7.7.1 Sage experiment: two different bases

```
B1 = matrix(ZZ, [[1, 0], [0, 1]])
B2 = matrix(ZZ, [[2, 1], [1, 1]])
```

Later, we will verify that these generate the same lattice.

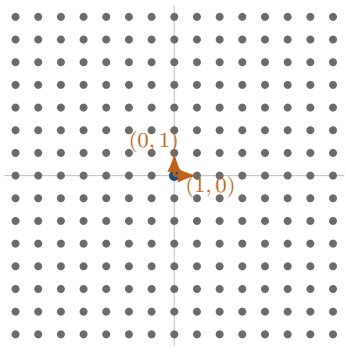
7.8 Good bases and bad bases

Now we arrive at the central idea behind lattice reduction. Some bases are good and some are bad.

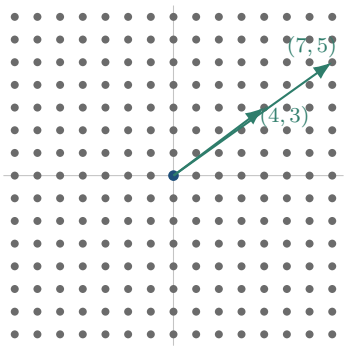
A bad basis is made of vectors that are nearly parallel and long. Such a basis makes it hard to find nearby lattice points.

A good basis consists of relatively short and nearly orthogonal vectors, which makes the lattice easier to navigate.

This is the reason that algorithms such as LLL and BKZ exist: they try to transform a bad basis into a better one.



(a) A good basis: short, nearly orthogonal.



(b) A bad basis for the *same* lattice $\mathbb{Z}^2$: long and nearly parallel.

Figure 7.3: Both bases generate exactly the same lattice ($\det = \pm 1$ in each case), but the bad basis on the right makes the lattice geometry almost invisible – the two vectors point in nearly the same direction. Recovering a good basis like the one on the left from a bad one like the one on the right is exactly the problem that lattice reduction solves.

7.9 Fundamental parallelepiped

Given a basis, we can form the fundamental parallelepiped, the basic cell of the lattice. Its volume is related to the determinant of the basis matrix.

For a two-dimensional lattice, the area of this cell is given by the absolute value of the determinant.

7.9.1 Sage experiment: determinant

```
B = matrix(ZZ, [[2, 1], [1, 3]])
B.det()
```

The determinant is a measure of how dense the lattice is. A larger determinant usually means a sparser lattice, while a smaller determinant means a denser one.

7.10 Lattice dimension

In two dimensions, a lattice is easy to visualize. In cryptography, however, the dimension is much larger. Kyber, Dilithium, and other lattice-based systems may work in dimensions of hundreds or even thousands. In these high-dimensional spaces, geometry becomes much harder to picture, but much of the same intuition still applies.

7.11 Why lattice problems become hard

In low dimensions, one can often solve lattice problems by hand or by simple geometric arguments. In higher dimensions, however, the search for short vectors or nearby vectors becomes extremely difficult. This explosion in complexity is the foundation of lattice-based security.

7.12 Sage experiment: shortest vector

Sage provides built-in lattice tools. A lattice object can be created directly from an integer matrix as follows:

```
from sage.modules.free_module_integer import IntegerLattice

B = matrix(ZZ, [[2, 1], [1, 3]])
L = IntegerLattice(B)
L
```

Once the lattice is defined, the shortest vector can be computed with a single command:

```
L.shortest_vector()
```

This is the basic object behind the shortest vector problem (SVP), one of the central hardness assumptions in lattice-based cryptography.

7.13 Experiments

The lattices underlying ML-KEM (FIPS 203 [1]) and ML-DSA (FIPS 204 [2]) live in dimensions of several hundred, where their geometry cannot be visualized. The following experiments isolate, in two dimensions where every object can be drawn and checked by hand, the four phenomena on which those standards ultimately depend: the discreteness of the point set, the basis-invariance of the determinant, the gap between good and bad bases, and the intractability of the shortest vector. Each recurs unchanged in principle at cryptographic scale.

7.13.1 Experiment 1: enumerating the lattice

Generate a finite window of the lattice $\Lambda(\mathbf{b}_1, \mathbf{b}_2)$ and confirm that it is a discrete grid of points, not a continuous span.

```
b1 = vector(ZZ, [2, 1])
b2 = vector(ZZ, [1, 3])
pts = [i * b1 + j * b2 for i in range(-4, 5) for j in range(-4, 5)]
pts
```

Plotting `pts` reveals the characteristic regular grid. This is the same object that, in dimension 256 and over $\mathbb{Z}_q$ rather than $\mathbb{Z}$, becomes the module lattice at the core of ML-KEM.

7.13.2 Experiment 2: the determinant is a basis invariant

Compute the lattice determinant $\det \Lambda = |\det B|$ for several bases of the *same* lattice, each obtained by left-multiplying B by a unimodular matrix $U \in \mathrm{GL}_n(\mathbb{Z})$ (an integer matrix with $\det U = \pm 1$).

```
B = matrix(ZZ, [[2, 1], [1, 3]])
U = matrix(ZZ, [[1, 0], [1, 1]])          # unimodular: det U = 1
print(abs(B.det()), abs((U * B).det()))   # identical
```

The determinant – the covolume of the fundamental parallelepiped, and hence the inverse density of the lattice – is unchanged, while the basis vectors themselves change completely. This invariance under $\mathrm{GL}_n(\mathbb{Z})$ is precisely what allows a scheme to publish one basis of a lattice while holding another in secret.

7.13.3 Experiment 3: good bases versus bad bases

A single lattice admits both near-orthogonal ("good") and highly skewed ("bad") bases. Contrast the two, and quantify the skew through the orthogonality defect $\prod_i \|\mathbf{b}_i\| / \det \Lambda$.

```
good = matrix(ZZ, [[1, 0], [0, 1]])       # orthonormal basis of Z^2
bad  = matrix(ZZ, [[1, 7], [0, 1]])       # a sheared basis of the same Z^2
```

This gap is the heart of trapdoor lattice cryptography. In the NTRU-based FN-DSA and, in spirit, throughout ML-DSA (FIPS 204 [2]), the secret key behaves as a good basis and the public key as a bad one; forging a signature without the good basis reduces to a hard lattice problem. Chapter 8 makes the good/bad distinction quantitative and shows how far a bad basis can be reduced toward a good one.

7.13.4 Experiment 4: the shortest vector is a lattice invariant

Compute the shortest nonzero vector for several distinct bases of the same lattice and confirm that its length is a property of the lattice, not of the chosen basis.

```
from sage.modules.free_module_integer import IntegerLattice
B1 = matrix(ZZ, [[2, 1], [1, 3]])
B2 = matrix(ZZ, [[3, 4], [4, 7]])          # = U*B1, U unimodular: same lattice
for B in [B1, B2]:
    print(IntegerLattice(B).shortest_vector())   # identical length
```

In two dimensions `shortest_vector` answers instantly. In the dimensions of FIPS 203 and 204 no known algorithm does so in feasible time, and that asymptotic intractability – formalized as the shortest vector problem (SVP), together with its approximate and decisional variants – is the security foundation on which both standards rest.

7.14 Summary

The main lessons of this chapter are threefold:

1. A lattice is not a vector space; it is a discrete structure formed by integer linear combinations.
2. A lattice can have many different bases, and the choice of basis strongly affects how easy or hard the lattice is to work with.
3. The hardness of lattice problems, especially SVP and CVP, is the foundation of many post-quantum cryptosystems.

The next chapter, *Gram–Schmidt Orthogonalization and the LLL Algorithm*, will introduce the first major algorithmic tool for turning a bad basis into a good one.

Chapter 8

Gram–Schmidt Orthogonalization and the LLL Algorithm

8.1 Why this chapter matters

This chapter is often described as the heart of lattice reduction. The reason is simple: once we understand what a lattice is and why some bases are better than others, we need a way to measure and improve that quality. Gram–Schmidt orthogonalization provides exactly that tool. It is the conceptual foundation for LLL, BKZ, Babai's nearest plane algorithm [22], and many other lattice techniques.

8.2 Learning goals

After reading this chapter, you should be able to:

- explain why some lattice bases are good and others are bad;
- understand the meaning of orthogonality;
- describe what Gram–Schmidt orthogonalization does conceptually;
- understand why LLL repeatedly uses Gram–Schmidt information;
- connect basis quality to later algorithms such as LLL and BKZ.

8.3 What makes a basis good?

We begin with intuition rather than formulas. Consider two bases in the plane. In one case, the vectors are short and nearly perpendicular. In the other case, the vectors are long and nearly parallel. Both may generate the same lattice, but the second is much harder to work with.

A good basis is one that makes the lattice easy to navigate. A bad basis is one that hides the short vectors and makes the lattice look much more complicated than it really is.

8.4 Why nearly parallel vectors are bad

Suppose we have a bad basis consisting of two vectors that are almost aligned. To express a simple vector such as $(1, 0)$, we may need to use very large integer combinations of the basis vectors. This means that the coordinate representation becomes unstable and large. In cryptography, this is exactly what we want to avoid.

This is why lattice reduction aims to find a better basis.

8.5 What does orthogonal mean?

Two vectors u and v are orthogonal if

$$u \cdot v = 0.$$

For example, $(1, 0)$ and $(0, 1)$ are orthogonal. In contrast, $(2, 1)$ and $(3, 2)$ are not.

8.5.1 Sage experiment: dot product

```
u = vector(QQ, [1, 0])
v = vector(QQ, [0, 1])
u.dot_product(v)
```

The result is 0, showing that the vectors are orthogonal.

8.6 What Gram–Schmidt does

The Gram–Schmidt process does not change the lattice. It does not even change the basis in a fundamental sense. Instead, it analyzes a basis by splitting each vector into two parts:

- a component along the previous directions;
- a new component that points in a genuinely new direction.

In this way, the process isolates the part of each vector that contributes new geometric information. The resulting vectors are orthogonal, and they reveal the true directions present in the basis.

This is why Gram–Schmidt is often described as a way of peeling off old directions and keeping only the new ones.

8.7 The projection formula

If u and v are vectors, then the projection of v onto u is

$$\mathrm{proj}_u(v) = \frac{u \cdot v}{u \cdot u} u.$$

The Gram–Schmidt process subtracts this projection from v, leaving the part of v that is orthogonal to u.

8.8 The Gram–Schmidt construction

The first vector is kept unchanged:

$$b_1^* = b_1.$$

Then each subsequent vector is adjusted by subtracting its projection onto earlier directions:

$$b_i^* = b_i - \sum_{j<i} \mu_{i,j} b_j^*,$$

where the coefficients $\mu_{i,j}$ are chosen so that the new vector is orthogonal to the previous ones.

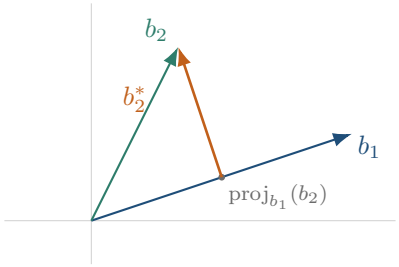


Figure 8.1: Gram–Schmidt decomposition of b_2 relative to b_1, for $b_1 = (3, 1)$ and $b_2 = (1, 2)$. The dot marks $\mathrm{proj}_{b_1}(b_2) = (1.5, 0.5)$, the component of b_2 along b_1. The orange vector $b_2^* = b_2 - \mathrm{proj}_{b_1}(b_2)$ is orthogonal to b_1: it is the genuinely new geometric information that b_2 contributes beyond b_1.

This formula generalizes directly to an algorithm: process the basis vectors one at a time, and for each one subtract off its projection onto every previously orthogonalized direction.

Algorithm 6 Gram–Schmidt Orthogonalization

Input: Basis $b_1, \ldots, b_n$
Output: Orthogonal vectors $b_1^*, \ldots, b_n^*$ and coefficients $\mu_{i,j}$

1: $b_1^* \leftarrow b_1$
2: **for** $i = 2, \ldots, n$ **do**
3: $\quad b_i^* \leftarrow b_i$
4: $\quad$ **for** $j = 1, \ldots, i-1$ **do**
5: $\quad\quad \mu_{i,j} \leftarrow \dfrac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$
6: $\quad\quad b_i^* \leftarrow b_i^* - \mu_{i,j}\, b_j^*$
7: $\quad$ **end for**
8: **end for**
9: **return** $(b_1^*, \ldots, b_n^*)$ and $(\mu_{i,j})$

8.9 Sage experiment: Gram–Schmidt

```
B = matrix(QQ, [[2, 1], [1, 3]])
B.gram_schmidt()
```

The output is a set of orthogonal vectors derived from the original basis. These are not necessarily lattice vectors themselves, but they are extremely useful for analysis.

8.9.1 Verification

```
u = vector(QQ, [2, 1])
v = vector(QQ, [-1, 2])
u.dot_product(v)
```

The dot product should be 0, showing that the two vectors are orthogonal.

8.10 Why Gram–Schmidt does not change the lattice

This is a subtle but important point. The Gram–Schmidt vectors are not usually integer vectors, and they may not lie in the lattice at all. They are analysis tools, not new lattice generators. Their role is to reveal the geometry of the lattice basis rather than to replace it.

8.11 Gram–Schmidt lengths

The lengths of the Gram–Schmidt vectors,

$$\|b_i^*\|,$$

are a key measure of basis quality. If the lengths decrease smoothly and remain reasonably balanced, then the basis is good. If one vector becomes extremely short and another extremely long, the basis is poor.

8.12 Orthogonality defect

A useful measure of quality is the orthogonality defect. One version is

$$\mathrm{OD}(B) = \frac{\prod_i \|b_i\|}{\det(B)}.$$

If the basis vectors are perfectly orthogonal, the defect is 1. The larger the defect, the worse the basis.

8.13 Hadamard ratio

Another standard quantity is the Hadamard ratio:

$$\mathrm{HR}(B) = \left(\frac{\det(B)}{\prod_i \|b_i\|}\right)^{1/n}.$$

It ranges between 0 and 1, and values closer to 1 indicate better basis quality.

8.14 Why LLL uses Gram–Schmidt repeatedly

The LLL algorithm [21] is built around the idea that a basis can be improved by repeatedly examining the Gram–Schmidt information. If the current basis is not sufficiently reduced, LLL swaps vectors, rescales, and recomputes the orthogonalized directions until the basis quality is acceptable.

This is why Gram–Schmidt is not just a side concept: it is the engine behind one of the most important lattice reduction algorithms. The algorithm alternates between two operations: *size reduction*, which uses the $\mu_{i,j}$ coefficients to make each b_i as close as possible to the span of the earlier vectors without changing the lattice, and the *Lovász condition*, a test on consecutive Gram–Schmidt lengths that decides whether two neighboring vectors should be swapped.

Algorithm 7 LLL Lattice Basis Reduction

Input: Basis $b_1, \dots, b_n$; reduction parameter $\delta \in (1/4, 1)$, typically $\delta = 3/4$
Output: An LLL-reduced basis generating the same lattice as $b_1, \dots, b_n$

Compute $b_1^*, \dots, b_n^*$ and $\mu_{i,j}$ via Gram–Schmidt (Algorithm 6)
$k \leftarrow 2$
while $k \le n$ **do**
  for $j = k-1$ down to 1 **do** ▷ size-reduce b_k against every earlier vector
    if $|\mu_{k,j}| > 1/2$ **then**
      $b_k \leftarrow b_k - \mathrm{round}(\mu_{k,j})\, b_j$
      recompute b_k^* and $\mu_{k,\cdot}$
    end if
  end for
  if $\|b_k^*\|^2 \ge (\delta - \mu_{k,k-1}^2)\, \|b_{k-1}^*\|^2$ **then** ▷ Lovász condition holds
    $k \leftarrow k + 1$
  else
    swap $b_k \leftrightarrow b_{k-1}$
    recompute $b_1^*, \dots, b_n^*$ and $\mu_{i,j}$
    $k \leftarrow \max(k-1,\, 2)$
  end if
end while
return $(b_1, \dots, b_n)$

Every swap strictly decreases a potential function built from the Gram–Schmidt lengths, so the loop terminates; in practice it converges in a small number of passes even in the hundreds of dimensions used by lattice-based cryptosystems. The output is not the shortest possible basis, but it is provably short enough – each $\|b_i\|$ is bounded in terms of the true shortest vector – to make LLL the workhorse preprocessing step before any serious lattice attack.

8.15 Connection to LWE

This chapter also helps explain why lattice problems are hard. In LWE, the secret and error are hidden in a high-dimensional structure. If an attacker could obtain a good basis for the associated lattice, many related tasks would become much easier. The security of lattice-based cryptography rests on the assumption that such a basis is not easily obtainable.

8.16 Experiments

8.16.1 Experiment 1: orthogonality

```
u = vector(QQ, [1, 0])
v = vector(QQ, [0, 1])
```

```
u.dot_product(v)
```

8.16.2 Experiment 2: compare two vector pairs

```
u = vector(QQ, [100, 99])
v = vector(QQ, [101, 100])
u.dot_product(v)
```

Compare the result with the previous experiment.

8.16.3 Experiment 3: Gram–Schmidt in Sage

```
B = random_matrix(QQ, 3)
B.gram_schmidt()
```

8.16.4 Experiment 4: orthogonality defect

Generate several random bases and compare their orthogonality defect values.

8.17 Summary

This chapter introduced the geometric notion of basis quality. Gram–Schmidt orthogonalization is not a new lattice construction; it is a way of analyzing a basis by separating old directions from genuinely new ones. A good basis is short and nearly orthogonal, while a bad basis is long and nearly parallel. These ideas are the direct precursor to LLL and other reduction algorithms.

The next chapter, *Shortest Vector Problem (SVP)*, will connect these geometric ideas to a concrete hard problem. From there, the book turns to noise distributions and the learning with errors problem, explaining why lattice problems are so closely tied to the security of modern post-quantum cryptography.

Chapter 9

Shortest Vector Problem (SVP)

9.1 What is the shortest vector?

Recall that a lattice is generated by a basis

$$B = (b_1, b_2, \ldots, b_n),$$

with all integer linear combinations

$$L(B) = \left\{ \sum_{i=1}^{n} z_i b_i : z_i \in \mathbb{Z} \right\}.$$

Among all nonzero lattice points, the shortest vector problem asks for the one whose Euclidean norm is minimal. Formally,

$$\lambda_1(L) = \min_{v \in L \setminus \{0\}} \|v\|.$$

The quantity $\lambda_1(L)$ is called the first successive minimum. In the literature, this quantity appears again and again in lattice cryptography.

In other words, if we imagine all integer combinations of the basis vectors, the shortest vector is the closest nonzero lattice point to the origin.

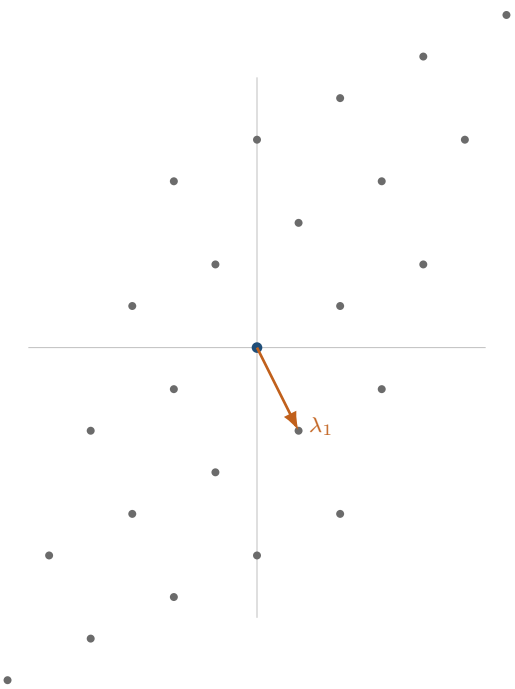


Figure 9.1: The same lattice $L(b_1, b_2)$ from Chapter 7, with $b_1 = (2, 1)$ and $b_2 = (1, 3)$. Among all the points shown, $(1, -2) = b_1 - b_2$ is a shortest nonzero vector, with $\lambda_1(L) = \sqrt{5}$. Notice that it is a *combination* of the basis vectors, not one of the basis vectors itself.

9.2 Why is two-dimensional SVP easy?

In low dimensions, the geometry is still visible. In two dimensions, one can often look at the lattice picture and identify the shortest vectors by eye. In three dimensions it is still possible to reason geometrically, but the picture becomes less intuitive. In four dimensions and above, the human intuition becomes weak. By the time we reach 256 dimensions, there is no meaningful way to visualize the geometry directly.

This is why lattice problems become computationally hard in high dimensions. The search space grows exponentially, and a brute-force approach is hopeless.

9.3 Why brute force is impossible

Suppose we try to search a lattice in dimension 100. If each coefficient is chosen from $-5, \ldots, 5$, then there are 11 possibilities per coordinate. The total number of candidate vectors is

$$11^{100},$$

which is roughly

$$1.37 \times 10^{104}.$$

That is far beyond the number of atoms in the observable universe. Exhaustive search is therefore infeasible.

Stated precisely, brute force is just Algorithm 5 from Chapter 7 with an extra step that tracks the minimum norm seen so far:

Algorithm 8 Brute-Force Shortest Vector Search

Input: Basis $b_1, \dots, b_n$; coefficient bound N
Output: A shortest vector among $\{\sum_i z_i b_i : |z_i| \le N,\ z \neq 0\}$

$v^\star \leftarrow$ NULL, $\quad m \leftarrow \infty$
for each $(z_1, \dots, z_n) \in \{-N, \dots, N\}^n \setminus \{0\}$ **do**
  $v \leftarrow \sum_{i=1}^{n} z_i\, b_i$
  if $\|v\| < m$ **then**
    $v^\star \leftarrow v, \quad m \leftarrow \|v\|$
  end if
end for
return $v^\star$

The loop body is cheap, but the loop itself visits $(2N+1)^n$ candidates. For $n = 100$ and $N = 5$ that is the 11^{100} figure above; this exponential blow-up in n, not any inefficiency in the inner computation, is what makes exact SVP hard.

9.4 A common beginner mistake

Many beginners assume that the shortest vector must be one of the basis vectors themselves. That is not true. A shortest vector is a lattice vector, but it need not be equal to any single basis vector. For example, the shortest vector can be a combination of basis vectors such as

$$b_2 - b_1,$$

which may be shorter than either basis vector individually.

This means that solving SVP requires looking at the entire lattice structure, not just the basis that happened to be given.

9.5 Sage experiment

We can build a small lattice and ask Sage for its shortest vector.

```
from sage.modules.free_module_integer import IntegerLattice

B = matrix(ZZ, [[2, 1], [1, 3]])
L = IntegerLattice(B)
print(L.shortest_vector())
print(L.shortest_vector().norm())
```

The output gives a shortest lattice vector and its Euclidean norm. This is the basic SVP instance.

9.6 Minkowski's theorem

Minkowski's theorem is one of the foundational results in lattice theory. It states that a lattice of dimension n must contain a nonzero vector whose norm is at most

$$\sqrt{n}\, \det(L)^{1/n}.$$

The important intuition is simple: if a lattice is dense, then the shortest vector cannot be too large. If the lattice is sparse, then the nearest lattice point is farther away. Minkowski's theorem

guarantees the existence of a vector that is not too long, although it does not tell us how to find it efficiently.

9.7 Exact SVP versus approximate SVP

There is an important distinction between exact SVP and approximate SVP.

- Exact SVP asks for the truly shortest vector.
- Approximate SVP only asks for a vector whose length is within a factor γ of the optimum.

In the notation of the literature, a γ-SVP solver returns a vector of length at most

$$\gamma \, \lambda_1(L).$$

This distinction matters because practical algorithms in lattice cryptanalysis usually do not solve exact SVP. Instead, they solve approximate variants, which are much more tractable.

9.8 Why LLL does not solve exact SVP

The LLL algorithm is extremely useful, but its goal is not to find the exact shortest vector. Rather, it transforms a basis into a more reduced basis, often producing a vector that is much shorter than the original one. However, the vector returned by LLL is typically not guaranteed to be the true shortest vector. In many cases, it is only a good approximation.

9.9 Why BKZ is stronger

BKZ improves on LLL by working with local blocks of the lattice. It repeatedly solves small-dimensional SVP instances inside a sliding window, then uses the resulting short vectors to improve the global basis. This makes BKZ stronger than LLL, but it also increases the computational cost substantially. In practice, BKZ can find very short vectors, but it still does not guarantee exact SVP in high dimensions.

9.10 Why SVP is hard

At the present time, the best known classical algorithms for exact SVP are still exponential in the dimension, and no known quantum algorithm provides the dramatic speedup that Shor's algorithm gives for factoring. As a result, lattice-based cryptography remains a practical and theoretically grounded approach to post-quantum security.

9.11 Connection with LWE

The connection to learning with errors is central. The LWE problem can be viewed as a noisy linear system over a lattice. If one could efficiently solve the relevant lattice problems, including SVP in the appropriate setting, then many LWE-based attacks would become much more powerful. Regev's reduction showed that LWE can be reduced to worst-case lattice problems, which is one of the deepest reasons that lattice-based cryptography is considered secure.

9.12 Main takeaways

- SVP asks for the shortest nonzero lattice vector, not merely the shortest basis vector.
- The hardness of SVP grows rapidly with dimension.
- LLL and BKZ solve approximate versions of SVP, not the exact problem in general.
- The connection between SVP and LWE is a core reason why lattice-based cryptography is believed to be secure.

Learning With Errors and Its Variants

Chapter 10

Probability and Error Distribution

10.1 What is a distribution?

Many textbooks write expressions such as

$$e \leftarrow \chi,$$

where the symbol χ is a probability distribution. In plain language, this means that the value e is sampled randomly from a certain distribution.

For example, when you roll a die, the outcome is one of

$$\{1, 2, 3, 4, 5, 6\},$$

with each value appearing with a certain probability. This is a distribution.

Formally, a distribution describes the probability of each possible outcome:

$$\Pr(X = x).$$

10.2 Uniform distribution

The simplest distribution is the uniform distribution. For example, suppose the value is chosen uniformly from

$$\{0, 1, 2, 3\}.$$

Then each element occurs with probability $1/4$.

In a histogram, all bars have the same height. This is the easiest notion of randomness, but it is usually not suitable for LWE noise.

If the noise were uniform over a very wide range, such as from -100 to 100, then the resulting error would be too large for decryption to work reliably. In short, uniform noise is too wild.

10.3 Gaussian distribution

Many natural phenomena are approximately Gaussian. Examples include human height, measurement error, and random fluctuations around a mean. A Gaussian distribution is centered at zero and has a bell shape. The standard notation is

$$N(0, \sigma^2),$$

meaning that the mean is 0 and the variance is σ^2.

The parameter σ controls the spread of the noise. A small σ means that most samples are close to zero; a large σ means that samples are more spread out.

10.4 Why the mean is often zero

In cryptographic noise models, the Gaussian is usually centered at zero. The reason is simple: noise should not systematically bias the system in one direction. If the error were biased, then the secret or the decrypted value might leak information.

10.5 Why we cannot use continuous Gaussian directly in LWE

The Gaussian distribution is a continuous distribution. It can produce real numbers such as

$$0.1352, \quad -1.2398, \quad 2.8888.$$

However, LWE is defined over a finite ring or finite field, typically modulo q. The coordinates of the error must be represented as integers modulo q.

Therefore, we use a discrete version of the Gaussian distribution.

10.6 Discrete Gaussian

A discrete Gaussian assigns probabilities to a discrete set of integers, such as

$$-3, -2, -1, 0, 1, 2, 3.$$

The probabilities are larger near zero and smaller away from zero. A typical definition is

$$\Pr(e = x) = \frac{e^{-x^2/(2\sigma^2)}}{Z},$$

where Z is a normalization constant ensuring that the total probability is 1.

This kind of distribution is important because it matches the mathematical structure of lattice-based cryptography and appears in many security proofs.

10.7 Why discrete Gaussian is so important

Discrete Gaussian sampling is central in Regev-style reductions and in the theoretical analysis of lattice problems. In many proofs, the Gaussian distribution is not just convenient; it is essential. If we replace it with something else, the reduction arguments may fail.

10.8 Toward CBD: the distribution Kyber ships

Elegant as the discrete Gaussian is in theory, it is awkward in practice: sampling it well needs exponentials, floating-point comparisons, or rejection loops – all slow, and all hard to make constant-time, so that a careless sampler leaks the secret through its timing. For this engineering reason Kyber abandons the Gaussian in favour of the *centered binomial distribution* (CBD): a bits-only

distribution, built from nothing more than sums of coin flips, whose bell shape approximates a Gaussian closely enough for security while remaining trivial to sample in constant time.

CBD is important enough – and specific enough to ML-KEM – to have a chapter of its own: Chapter 15 develops it in full, from the "count the ones in each half and subtract" recipe to the exact FIPS 203 sampling procedure and its constant-time properties. Here we only place it at the end of the progression of noise distributions.

10.9 What does σ mean?

In the Gaussian model, the parameter σ controls the magnitude of the error. If $\sigma = 0.5$, then most errors are very small. If $\sigma = 100$, then the noise is huge and the system becomes unstable. In practice, the design goal is to choose σ large enough to hide the secret, but small enough that decryption still succeeds with high probability.

10.10 Tail bound

Gaussian distributions are theoretically unbounded, but in practice the probability of very large errors is tiny. When cryptographers talk about the tail of a distribution, they mean the probability of values far away from the center. A short tail is useful because it means that extreme errors are rare, which helps with correctness.

10.11 The noise budget

Every scheme built on LWE and RLWE carries a small error term inside each ciphertext, and correct decryption depends on keeping that error below a fixed threshold – the *noise budget*. The tension is fundamental: choose the noise too small and the underlying problem becomes easy, so the scheme is insecure; choose it too large and honest decryption fails. Later chapters return to this balance when we fix concrete parameters for RLWE-based encryption and for ML-KEM, where the design drives the decryption-failure probability down to a negligible level. Managing the margin between correctness and security is one of the central practical challenges in lattice-based cryptography.

10.12 Why cryptography likes Gaussian noise

Cryptography likes Gaussian noise for three main reasons.

1. It is mathematically elegant and easy to reason about.
2. It fits naturally with lattice-based proofs and reductions.
3. It provides a strong theoretical connection to worst-case lattice hardness.

In short, noise is not there just to make things random. It is used to create a balance between correctness and security.

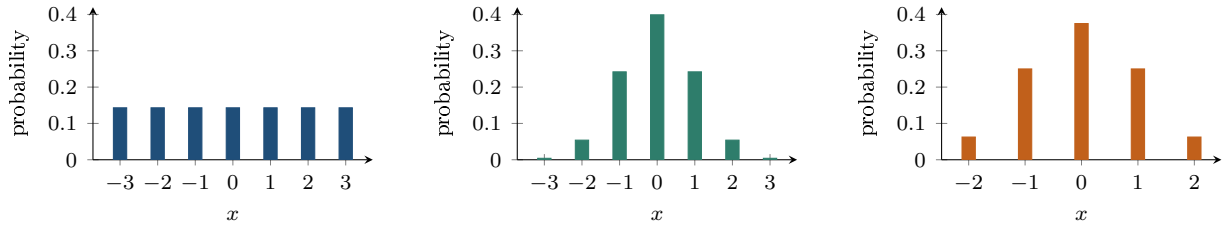
- If the noise is too small, the secret can be recovered more easily.
- If the noise is too large, legitimate users cannot decrypt correctly.
- The parameter choices in schemes such as q, n, η, and σ are designed to balance these trade-offs.

10.13 Summary

The chain of ideas in this chapter runs from the simplest possible randomness to the distribution that ML-KEM actually uses:

$$\text{Uniform} \longrightarrow \text{Gaussian} \longrightarrow \text{Discrete Gaussian} \longrightarrow \text{Centered Binomial Distribution (CBD)}.$$

Uniform noise is easy to sample but too wild for correct decryption; the (discrete) Gaussian is the theoretically convenient choice behind most security proofs; CBD is the practical compromise that Kyber ships, built from nothing more than sums of coin flips. Figure 10.1 compares the three shapes directly: the uniform histogram is flat, while both the discrete Gaussian and CBD($\eta = 2$) already show the familiar bell shape, concentrated near zero.



(a) Uniform (b) Discrete Gaussian (c) CBD($\eta = 2$)

Figure 10.1: Three noise shapes over the same range $\{-3, \ldots, 3\}$. Uniform (left, blue) is flat: every value is equally likely, which is too wild for reliable decryption. Discrete Gaussian (middle, teal) is the theoretically convenient bell curve. CBD with $\eta = 2$ (right, orange) is produced only by summing and subtracting a handful of random bits (Chapter 15), yet its shape already approximates the same bell curve.

The central idea is that noise is not merely a random artifact. It is one of the main mechanisms that makes LWE and related systems both secure and decryptable.

Chapter 11

Learning With Errors (LWE)

11.1 From exact linear algebra to a noisy problem

Solving a system of linear equations over a field is one of the oldest solved problems in mathematics. Given $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ and $\mathbf{b} \in \mathbb{Z}_q^m$, Gaussian elimination recovers $\mathbf{s}$ from $\mathbf{As} = \mathbf{b}$ when $\mathbf{A}$ has full column rank and the system is consistent. Its usual cost is $O(mn^2)$ field operations (or $O(n^3)$ for a square $n \times n$ system). Because the problem is this easy, an *exact* full-rank linear system carries no cryptographic hardness: an adversary can recover the secret directly.

The Learning With Errors (LWE) problem, introduced by Regev [12], makes a single, deliberate change to this picture: it perturbs each equation by a small additive error. The observed system is

$$\mathbf{b} = \mathbf{As} + \mathbf{e} \pmod{q},$$

where $\mathbf{e}$ is a short noise vector drawn from a prescribed distribution. Ordinary exact elimination no longer solves this noisy modular system: it can detect inconsistency, but it does not identify the planted secret. LWE hardness is a parameterized cryptographic assumption about recovering that secret from many finite-modulus, small-error samples; it is not a claim that every noisy regression problem is hard. This chapter develops LWE as a formal object: its distribution, its two decision problems, the geometry that explains its hardness, and the worst-case reduction that makes it a trustworthy foundation. It is the assumption on which ML-KEM (Chapter 16) and ML-DSA (Chapter 17) ultimately rest.

11.2 The LWE distribution

Throughout, fix a dimension n (the security parameter), a modulus $q = q(n)$, and an *error distribution* χ over $\mathbb{Z}_q$ concentrated on values of small absolute value. Vectors are column vectors, and $\langle \cdot, \cdot \rangle$ denotes the inner product modulo q.

Definition 11.1 (LWE distribution)**.** For a fixed secret $\mathbf{s} \in \mathbb{Z}_q^n$, the *LWE distribution* $A_{\mathbf{s},\chi}$ over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ is sampled by drawing $\mathbf{a} \leftarrow \mathbb{Z}_q^n$ uniformly and $e \leftarrow \chi$, and outputting

$$(\mathbf{a},\, b), \qquad b = \langle \mathbf{a}, \mathbf{s} \rangle + e \pmod{q}.$$

Collecting m independent samples and stacking the vectors $\mathbf{a}_i$ as the rows of a matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ gives the compact matrix form

$$(\mathbf{A},\, \mathbf{b}), \qquad \mathbf{b} = \mathbf{As} + \mathbf{e} \pmod{q}, \quad \mathbf{e} \leftarrow \chi^m.$$

The name is now transparent: the adversary is asked to *learn* the hidden $\mathbf{s}$ from many noisy linear measurements of it. The only obstruction is the error $\mathbf{e}$; without it the samples are an exactly solvable system.

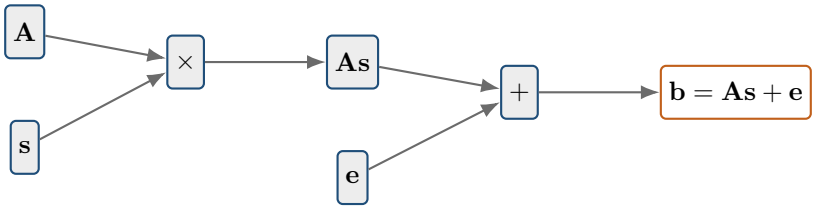


Figure 11.1: The LWE sample equation as a computation graph: the public matrix $\mathbf{A}$ and the secret $\mathbf{s}$ feed a multiply node, and the noise $\mathbf{e}$ is added to the product $\mathbf{As}$ to produce the public value $\mathbf{b}$. All arithmetic is modulo q. Recovering $\mathbf{s}$ from $(\mathbf{A}, \mathbf{b})$ alone is the LWE problem; the entire difficulty is contributed by the otherwise-negligible term $\mathbf{e}$.

Remark (On the secret distribution). The definition draws $\mathbf{s}$ uniformly from $\mathbb{Z}_q^n$, but a standard transformation puts LWE into *normal form*, in which the secret is drawn from the error distribution χ and is therefore itself short, without loss of hardness. Deployed schemes exploit this: in ML-KEM the secret is a small vector sampled from a centered binomial distribution (Chapter 15), which shortens keys and speeds arithmetic while preserving security.

11.3 The two LWE problems

LWE gives rise to a search problem and a decision problem, and the distinction is central: encryption schemes are proved secure against the *decision* version, while the *search* version is the more intuitive statement of "recover the key."

Definition 11.2 (Search-LWE)**.** Given m independent samples $(\mathbf{A}, \mathbf{b})$ drawn from $A_{\mathbf{s},\chi}$ for a fixed but unknown $\mathbf{s} \in \mathbb{Z}_q^n$, recover $\mathbf{s}$.

Definition 11.3 (Decision-LWE)**.** Distinguish, with non-negligible advantage, between the two distributions on $\mathbb{Z}_q^{m\times n} \times \mathbb{Z}_q^m$:

- the LWE distribution: $(\mathbf{A}, \mathbf{As} + \mathbf{e})$ with $\mathbf{s} \leftarrow \mathbb{Z}_q^n$ and $\mathbf{e} \leftarrow \chi^m$;
- the uniform distribution: $(\mathbf{A}, \mathbf{u})$ with $\mathbf{u} \leftarrow \mathbb{Z}_q^m$ uniform and independent of $\mathbf{A}$.

The decision formulation is what makes LWE a workhorse for cryptography: if LWE samples are computationally indistinguishable from uniform randomness, then a ciphertext that masks a message by adding it to an LWE sample is indistinguishable from a random string, which is exactly semantic security. Remarkably, the two problems are equivalent.

Proposition 11.4 (Search–decision equivalence)**.** *For a prime modulus $q = \mathrm{poly}(n)$, Search-LWE and Decision-LWE are equivalent up to a polynomial factor in running time and sample complexity: an efficient algorithm for either yields an efficient algorithm for the other.*

The forward direction (search solves decision) is immediate. The reverse – bootstrapping a mere distinguisher into a full secret-recovery algorithm – is the substantial part: one guesses each coordinate $s_j \in \mathbb{Z}_q$ in turn, uses the distinguisher to test the guess, and amplifies. Regev proved the equivalence for prime $q = \mathrm{poly}(n)$ [12]; later work extended it to composite and larger moduli. The practical consequence is that designers may reason about the clean search problem while security proofs consume the decision problem.

11.4 The error distribution and the modulus-to-noise ratio

The error distribution χ is the beating heart of LWE. In Regev's formulation it is a *discrete Gaussian* $D_{\mathbb{Z},\sigma}$ of width σ: the probability of an integer x is proportional to $\exp(-\pi x^2/\sigma^2)$, rounded to $\mathbb{Z}$. It is convenient to parameterize the width relative to the modulus by a factor $\alpha \in (0,1)$,

$$\sigma \;=\; \alpha q,$$

so that α measures the noise as a fraction of the modulus. Two competing constraints pin down the right regime.

- **Correctness needs small noise.** A legitimate decryptor removes the error by rounding, which succeeds only while $|e|$ stays below roughly $q/4$. Too much noise and honest decryption fails.
- **Security needs enough noise.** If αq is too small the error is negligible and the system is essentially exact, hence easy. The worst-case hardness theorem below requires $\alpha q \gtrsim \sqrt{n}$.

The single quantity that governs both is the *modulus-to-noise ratio* $q/\sigma = 1/\alpha$. A larger ratio makes decryption more robust but the underlying problem easier (it corresponds to a larger lattice approximation factor); a smaller ratio does the reverse. Parameter selection for a real scheme is, at bottom, the art of choosing this ratio. In practice the exact discrete Gaussian is replaced by the centered binomial distribution (Chapter 15), which has essentially the same shape but is far easier to sample in constant time; the foundational analysis, however, is cleanest for the Gaussian [14].

11.5 A geometric view: bounded-distance decoding

LWE has a precise lattice interpretation, which is the source of both its hardness and its name-recognition among lattice problems. Given $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$, define the *q-ary lattice*

$$\Lambda_q(\mathbf{A}) \;=\; \{\, \mathbf{A}\mathbf{s} \bmod q : \mathbf{s} \in \mathbb{Z}_q^n \,\} + q\mathbb{Z}^m \;\subseteq\; \mathbb{Z}^m.$$

The noiseless product $\mathbf{As}$ is exactly a point of this lattice, and the observed vector $\mathbf{b} = \mathbf{As} + \mathbf{e}$ is that lattice point displaced by the short offset $\mathbf{e}$. Recovering $\mathbf{s}$ is therefore the problem of finding the lattice point nearest to $\mathbf{b}$.

This is the closest vector problem (CVP) of Chapter 9, but with a crucial promise: the target $\mathbf{b}$ is guaranteed to lie *within a small, known distance* of the lattice, since $\|\mathbf{e}\|$ is bounded by the error distribution. A CVP instance carrying this promise is called *Bounded-Distance Decoding* (BDD).

Proposition 11.5 (LWE is bounded-distance decoding)**.** *Search-LWE with error norm bounded by r is an instance of* BDD *on the lattice $\Lambda_q(\mathbf{A})$ with decoding radius r; the secret $\mathbf{s}$ is recovered from the closest lattice point $\mathbf{As}$.*

The BDD promise is what an attacker tries to exploit and what the parameters are tuned to deny. When the decoding radius is a small fraction of the lattice's minimum distance, the nearest lattice point is unique but still hard to find in high dimension; this is the operating point of every deployed scheme.

11.6 Hardness: the worst-case to average-case reduction

The reason LWE is trusted is not merely that no efficient attack is known. It is that Regev proved a *worst-case to average-case reduction*: breaking LWE on random instances is at least as hard as solving standard lattice problems on *every* instance.

Theorem 11.6 (Worst-case hardness of LWE, informal [12]). *Let n be a security parameter, $q = q(n) \leq \mathrm{poly}(n)$ a modulus, and $\alpha = \alpha(n) \in (0,1)$ with $\alpha q > 2\sqrt{n}$. If there is an efficient (possibly randomized) algorithm that solves Decision-LWE$_{n,q,\alpha}$ on the average, then there is an efficient* quantum *algorithm that approximates the decision shortest-vector problem* (GapSVP) *and the shortest-independent-vectors problem* (SIVP) *to within $\widetilde{O}(n/\alpha)$ factors, in the worst case, on every n-dimensional lattice.*

Three points give this theorem its weight.

- **Worst-case to average-case.** A scheme's keys are random (average-case) LWE instances. The reduction guarantees that breaking a random key is as hard as the worst conceivable lattice, so there are no "weak keys" to stumble into – a guarantee that number-theoretic assumptions such as factoring do not provide.
- **The role of quantumness.** The reduction uses a quantum algorithm as a proof device only. LWE-based schemes run on entirely classical hardware; the quantum step is in the security argument, not the cryptosystem. Peikert later gave a *classical* reduction for exponential moduli [15], and Brakerski, Langlois, Peikert, Regev, and Stehlé established classical hardness for polynomial moduli [16], removing the quantum caveat.
- **The approximation factor.** The reduction yields hardness for $\widetilde{O}(n/\alpha)$-approximate lattice problems, which are believed to require time $2^{\Theta(n)}$ for the parameters used in practice. The factor n/α makes precise the tension of the previous section: less noise (smaller α) yields a weaker guarantee.

11.7 The dual problem: Short Integer Solution (SIS)

LWE has a close relative that becomes important once we reach signatures. Where LWE *hides* a secret inside noise and asks you to recover it, the **Short Integer Solution** (SIS) problem [13] asks you to *find* a short combination of public vectors. Given a random matrix

$$\mathbf{A} \in \mathbb{Z}_q^{n \times m}, \qquad m > n,$$

the task is to find a short nonzero integer vector $\mathbf{z}$ with

$$\mathbf{A}\mathbf{z} = \mathbf{0} \pmod{q}, \qquad 0 < \|\mathbf{z}\| \leq \beta.$$

Because there are more columns than rows, such vectors certainly exist; the difficulty is finding one that is *short*. A random solution would have large entries, and shrinking it is exactly the kind of short-vector search that is hard in a high-dimensional lattice. Geometrically, SIS lives in the *orthogonal* q-ary lattice

$$\Lambda_q^{\perp}(\mathbf{A}) \;=\; \{\, \mathbf{z} \in \mathbb{Z}^m : \mathbf{A}\mathbf{z} = \mathbf{0} \bmod q \,\},$$

and a solution is a short vector of $\Lambda_q^{\perp}(\mathbf{A})$ – the SVP counterpart to the BDD instance $\Lambda_q(\mathbf{A})$ that defines LWE.

The label *dual* is precise, not a loose analogy, and it holds in three linked senses. Write $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ for the SIS matrix; its transpose has the shape of an LWE sample matrix (Section 11.5), so the matching LWE sample is $\mathbf{b} = \mathbf{A}^\mathsf{T}\mathbf{s} + \mathbf{e}$, and

$$\Lambda_q(\mathbf{A}) \;=\; \{\mathbf{A}^\mathsf{T}\mathbf{s} \bmod q : \mathbf{s} \in \mathbb{Z}_q^n\} + q\mathbb{Z}^m$$

is exactly the LWE image lattice seen there.

- **Dual lattices.** The SIS lattice and the LWE lattice are dual to each other, up to the scale factor q:

$$\Lambda_q^\perp(\mathbf{A}) \;=\; q \cdot \Lambda_q(\mathbf{A})^*, \qquad L^* = \{\mathbf{x} \in \mathrm{span}(L) : \langle \mathbf{x}, \mathbf{y} \rangle \in \mathbb{Z} \;\; \forall\, \mathbf{y} \in L\}.$$

  SIS lives on the dual of the lattice on which LWE lives – this is the literal source of the name.

- **Dual problem types.** On that pair, the two are the archetypal lattice problems, themselves a dual pair: LWE is bounded-distance decoding (find the lattice point near $\mathbf{b}$), while SIS is a shortest-vector search (find a short $\mathbf{z} \in \Lambda_q^\perp(\mathbf{A})$). Short vectors in the dual lattice are precisely what let one decode in the primal.

- **A possible dual attack.** A sufficiently short SIS vector can support an LWE distinguisher. For a short $\mathbf{z}$ with $\mathbf{A}\mathbf{z} = \mathbf{0}$,

$$\langle \mathbf{z}, \mathbf{b} \rangle = \mathbf{z}^\mathsf{T}\Big(\mathbf{A}^\mathsf{T}\mathbf{s} + \mathbf{e}\Big) = (\mathbf{A}\mathbf{z})^\mathsf{T}\mathbf{s} + \langle \mathbf{z}, \mathbf{e} \rangle = \langle \mathbf{z}, \mathbf{e} \rangle \pmod q,$$

  which can be concentrated when $\mathbf{z}$ and $\mathbf{e}$ are sufficiently short, whereas $\langle \mathbf{z}, \mathbf{u} \rangle$ for a uniform $\mathbf{u}$ is spread over $\mathbb{Z}_q$. This observation motivates dual attacks, but it is not a general search-problem equivalence: the concentration depends on the parameters, and reductions between LWE and SIS require substantially more than this displayed identity.

Both problems have their own worst-case lattice reductions – SIS to the worst case of SIVP [13], and LWE through Theorem 11.6. They are closely related geometric assumptions, but should not be treated as interchangeable problems. We will meet SIS again directly: in ML-DSA / Dilithium (Chapter 17), forging a signature amounts to producing a short vector satisfying a public equation, which is precisely an SIS instance.

11.8 Concrete security and cryptanalysis

Theorem 11.6 is asymptotic; deploying LWE requires *concrete* parameters (n, q, α, m) for which the best known attack costs more than 2^{128} (or 2^{192}, 2^{256}) operations. Concrete security is estimated against lattice-reduction attacks rather than the reduction itself, since the reduction is not tight.

- **Primal attack.** Embed the BDD instance of Section 11.5 as a unique-shortest-vector instance in a lattice of dimension $\approx m + n + 1$ (Kannan's embedding) and run lattice reduction to find the short error vector, revealing $\mathbf{s}$.

- **Dual attack.** Find short vectors in the orthogonal lattice $\Lambda_q^\perp(\mathbf{A})$ and use them to distinguish LWE samples from uniform, solving Decision-LWE directly.

Both are driven by the **Block-Korkine-Zolotarev** (BKZ) algorithm with block size β, whose cost grows as $2^{\Theta(\beta)}$; the attacker's task reduces to finding the smallest β that achieves the root-Hermite factor needed to solve the instance. The community consolidates these estimates in the "LWE estimator" [18], building on the concrete-attack analysis of Lindner and Peikert [17]; ML-KEM's parameter sets are chosen to clear the target security levels under this estimator.

Remark (The number of samples is part of the assumption). Attack cost depends on how many samples m the adversary sees, not only on (n, q, α): more equations can make the underlying lattice easier to reduce. A well-designed scheme therefore *bounds* the number of LWE samples an adversary can obtain from public data. This is one reason a KEM publishes a fixed-size key rather than an unbounded stream of samples.

11.9 Choosing the modulus q

The modulus ties the preceding threads together. It must be large enough that the error rarely wraps around modulo q and destroys correctness, yet small enough to keep keys compact and arithmetic fast; and, as Section 11.4 showed, the ratio q/σ directly sets the security–correctness trade-off. Real schemes add a further constraint: q is chosen to admit an efficient number-theoretic transform (Chapter 6), which is why ML-KEM fixes the specific prime $q = 3329$. Modulus selection is thus the first place where the abstract assumption meets hard engineering.

11.10 Sage experiments

The quickest route to intuition is to build a small instance by hand and watch the error convert an easy system into a hard one. Fix a modulus and dimension.

```
from sage.all import *

q = 17
F = GF(q)
dimension = 3
```

Choose a secret vector and a batch of random rows.

```
s = random_vector(F, dimension)
A = random_matrix(F, 8, dimension)
print("secret =", s)
```

Without noise, $\mathbf{b} = \mathbf{A}\mathbf{s}$ is an exact system and $\mathbf{s}$ is recovered by solving it directly.

```
b = A * s
print(A.solve_right(b) == s)   # True: exact recovery
```

Now add a short error. In $\mathbb{F}_{17}$ the value 16 represents -1, so this vector has entries in $\{-1, 0, 1\}$.

```
noise   = vector(F, [1, 0, 16, 1, 0, 0, 16, 1])
b_noisy = A * s + noise
```

The system $\mathbf{A}\mathbf{s}' = \mathbf{b}_{\text{noisy}}$ now has no exact solution, and naive solving returns the wrong secret. Sweeping the error magnitude reveals the phase transition of Section 11.4: below a noise threshold the secret is still recoverable by more careful decoding, and above it recovery becomes infeasible. Algorithm 9 states the sampling procedure formally, using the same variable names as the code.

Algorithm 9 Generate an LWE Sample

Input: Modulus q; dimension n; number of rows m; error distribution χ over $\mathbb{Z}_q$
Output: Public sample $(\mathbf{A}, \mathbf{b})$ and secret $\mathbf{s}$

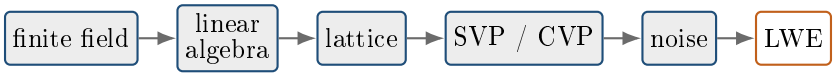
1: $\mathbf{s} \leftarrow$ uniformly random vector in $\mathbb{Z}_q^n$ ▷ `s = random_vector(F, n)`
2: $\mathbf{A} \leftarrow$ uniformly random matrix in $\mathbb{Z}_q^{m \times n}$ ▷ `A = random_matrix(F, m, n)`
3: $\mathbf{e} \leftarrow (e_1, \ldots, e_m)$ with each $e_i \leftarrow \chi$ ▷ `noise = vector(F, [...])`
4: $\mathbf{b} \leftarrow \mathbf{A}\mathbf{s} + \mathbf{e} \pmod q$ ▷ `b_noisy = A * s + noise`
5: **return** public sample $(\mathbf{A}, \mathbf{b})$, secret $\mathbf{s}$

11.11 Summary

LWE turns the easiest problem in linear algebra into one of the most robust hardness assumptions in cryptography by adding calibrated noise:

$$\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e} \pmod q.$$

Its structure rests on a few precise ideas: the search and decision formulations are equivalent (Proposition 11.4); the problem is bounded-distance decoding on the q-ary lattice $\Lambda_q(\mathbf{A})$; its average-case hardness follows from worst-case lattice problems (Theorem 11.6); its dual SIS lives in the orthogonal lattice $\Lambda_q^{\perp}(\mathbf{A})$; and concrete parameters are set against BKZ-based estimators rather than the asymptotic reduction. The conceptual chain built over the last several chapters is now complete.



The next chapter carries this assumption into a ring, replacing vectors of integers with polynomials to obtain the far more efficient Ring-LWE.

Chapter 12

Ring-LWE (RLWE)

12.1 A roadmap of the LWE family

This chapter is the middle step of a three-part evolution, and it helps to see the whole arc before diving into the details. Lattice-based key exchange did not arrive fully formed; it grew through a sequence of assumptions, each fixing a shortcoming of the last.

- **LWE** (Regev, 2005 [12]) is provably as hard as worst-case lattice problems, but its public key is a full $n \times n$ matrix – hundreds of kilobytes – which is impractical to deploy (Chapter 11).
- **Ring-LWE** (Lyubashevsky, Peikert, and Regev, 2010 [19]) replaces that matrix with a single polynomial in a ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$, shrinking the key from an entire matrix to one row's worth of coefficients and enabling near-linear-time multiplication. The price is extra algebraic structure, a potential attack surface. **This chapter builds RLWE.**
- **Module-LWE** (Langlois and Stehlé, 2015 [20]) is the deliberate compromise: a short *vector* of ring elements instead of a single one. It recovers most of RLWE's efficiency while diluting the ring structure, hedging against any future attack that exploits the full ring. The next chapter builds it.
- **ML-KEM** (Kyber [30], standardized as FIPS 203 in 2024 [1]) is the module-lattice key-encapsulation mechanism that the next part assembles in full – the destination of this whole progression.

One trade-off runs through the entire story: *more algebraic structure buys smaller keys and faster arithmetic, but widens the attack surface.* LWE sits at one extreme (no structure, largest keys); RLWE at the other (full ring structure, smallest keys); Module-LWE is the tunable middle that the standard ultimately chose. Note the arc is not monotone – the field advanced to RLWE's full structure, then deliberately *stepped back* to Module-LWE as a hedge. With that map in hand, we now build the ring step.

12.2 The fatal problem of LWE

LWE is elegant in theory, but it is not practical in the naive form. Recall the basic equation

$$b = As + e.$$

If the dimension is $n = 1024$, then the matrix A has roughly 1024×1024 entries, which is more than one million coefficients. If each coefficient is stored in a few bytes, then the public key already becomes large. In practice, a straightforward LWE public key is far too costly for real deployment.

This is why LWE is beautiful theoretically but unattractive in engineering. The algebra is simple, but the representation is bulky.

12.3 Observe the negacyclic matrix structure

RLWE uses the quotient $x^n + 1$, not $x^n - 1$. For $a(x) = a_0 + a_1x + a_2x^2$, multiplication modulo $x^3 + 1$ is represented by

$$N(a) = \begin{bmatrix} a_0 & -a_2 & -a_1 \\ a_1 & a_0 & -a_2 \\ a_2 & a_1 & a_0 \end{bmatrix}.$$

The shifts are structured, but a coefficient crossing the boundary changes sign because $x^3 = -1$. This is a *negacyclic* (signed-circulant) matrix, not an ordinary circulant matrix. One polynomial still determines the whole linear map, so the storage benefit remains.

12.4 Why negacyclic structure appears

For $a(x) = 1 + 2x + 3x^2$ and $s(x) = 4 + 5x + 6x^2$, ordinary multiplication creates terms through degree four. Reducing modulo $x^3 + 1$ maps x^3 to -1 and x^4 to $-x$. The sign-flipped wrap-around is the key structure behind RLWE; Chapter 5 develops the cyclic and negacyclic cases side by side.

12.5 The working ring R_q

RLWE lives not in an ordinary polynomial ring but in the quotient ring

$$R_q = \mathbb{Z}_q[x]/(x^n + 1),$$

whose construction – quotient rings, the reduction rule $x^n \equiv -1$, and the cyclotomic modulus x^n+1 – was developed in Chapter 4. We recall only what the rest of this chapter needs. Multiplying two polynomials can push the degree to n or beyond, but the defining relation $x^n \equiv -1$ (hence $x^{n+1} \equiv -x$, $x^{n+2} \equiv -x^2$, and so on) folds every product back to degree below n, so the representation stays compact and structured. Kyber instantiates this as $R_q = \mathbb{Z}_{3329}[x]/(x^{256} + 1)$.

12.6 Sage experiment

We can define a simple quotient ring in Sage.

```
q = 17
from random import choice
R.<x> = PolynomialRing(GF(q))
S = R.quotient(x^8 + 1)

a = S(x^3 + 2*x + 1)
b = S(x^2 + 3)
print(a*b)
```

The multiplication is automatically reduced modulo $x^8 + 1$.

12.7 From LWE to RLWE

In standard LWE, the secret is a vector

$$s = (s_1, \ldots, s_n).$$

In RLWE, the secret becomes a polynomial

$$s(x) = s_0 + s_1 x + \cdots + s_{255} x^{255}.$$

Likewise, the public value a becomes a polynomial. The noisy equation becomes

$$b(x) = a(x)s(x) + e(x).$$

The mathematics is not fundamentally different, but the representation changes. Instead of storing and multiplying large random matrices, we work with structured polynomials. The picture below makes this compression concrete.

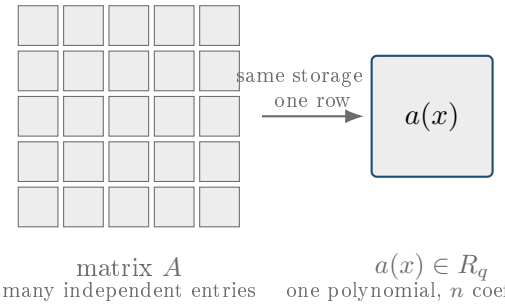


Figure 12.1: RLWE's storage saving in one picture. A general LWE matrix $A \in \mathbb{Z}_q^{n \times n}$ is replaced by a single polynomial $a(x) \in R_q$. Its coefficients determine a structured negacyclic linear map, so one polynomial replaces an entire unstructured matrix. Concrete size savings depend on the parameter choices and encodings.

12.8 Why RLWE is faster

In ordinary LWE, multiplication is matrix multiplication, which costs roughly $O(n^2)$ operations. For a dimension around 256, this means tens of thousands of operations.

In RLWE, the multiplication is polynomial multiplication. With FFT-like methods, this becomes roughly $O(n \log n)$. The speedup is substantial. In practice, the ring structure allows the system to run far faster.

This is one of the central reasons RLWE became attractive for real-world deployments.

12.9 Fast multiplication via the NTT

The NTT of Chapter 6 provides the fast representation. A conventional cyclic NTT diagonalizes cyclic convolution; the $x^n + 1$ ring instead needs negacyclic handling. For ML-KEM, the partial NTT maps to 128 quadratic components rather than to unrestricted scalar coordinates. It is this specified transform structure that makes the polynomial products efficient in practice.

12.10 Sage experiment: polynomial multiplication

We can see the basic idea without writing a full NTT implementation.

```
R.<x> = PolynomialRing(GF(17))
f = 1 + 2*x + x^2
g = 3 + x
print(f*g)
```

In a full implementation, the NTT (Chapter 6) is the technique used to compute this multiplication much faster than naive schoolbook multiplication.

12.11 Is RLWE secure?

At first glance, the additional algebraic structure might look dangerous. Maybe it makes the problem easier for an attacker. The security story is more subtle. RLWE is based on hardness in ideal lattices, not just general lattices [19]. This makes the security analysis more involved, but it is still believed to be strong enough for practical use.

12.12 Sage experiment: a toy RLWE-style setup

We can create a small ring and form the analog of an LWE sample.

```
q = 17
from random import choice
R.<x> = PolynomialRing(GF(q))
S = R.quotient(x^8 + 1)

a = S.random_element()  # uniform public ring element

def small_poly():
    return S([choice([-1, 0, 1]) for _ in range(8)])

s = small_poly()
e = small_poly()

b = a*s + e
print("a =", a)
print("s =", s)
print("e =", e)
print("b =", b)
```

This is a toy RLWE-style sample: a is uniform, while s and e have visibly small coefficients. Written independently of any particular computer-algebra system, the same construction is Algorithm 10.

Algorithm 10 Generate an RLWE Sample

Input: Ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$
Output: RLWE sample (a, b); secret s

1: $a(x) \leftarrow$ uniformly random element of R_q
2: $s(x) \leftarrow$ small (noise-sized) element of R_q ▷ the secret
3: $e(x) \leftarrow$ small (noise-sized) element of R_q ▷ the error
4: $b(x) \leftarrow a(x)\, s(x) + e(x) \pmod{x^n + 1}$
5: **return** (a, b) as the sample, s as the secret

12.13 Comparison: LWE vs RLWE

Property	LWE	RLWE
Data structure	vector and matrix	polynomial
Space	$O(n^2)$	$O(n)$
Computation	matrix multiplication	polynomial multiplication
Acceleration	limited	NTT-friendly
Security foundation	general lattice	ideal lattice
Practical deployment	uncommon	widespread

12.14 Why Kyber is not pure RLWE

A classic interview question is: why is Kyber not pure RLWE? The answer is that Kyber uses a vector of polynomials rather than a single polynomial. In other words, the secret is not just one ring element but a module element over the ring. That is the origin of Module-LWE, which we will study next.

12.15 A sibling assumption: NTRU

RLWE is not the only way to build a hard problem inside the ring R_q. An older construction, **NTRU** [23] (from 1996, predating LWE), lives in the same ring but hides its secret differently. Instead of the additive-noise equation $b = as+e$, NTRU picks *two* short secret polynomials $f, g \in R_q$ (with f invertible) and publishes a single quotient:

$$h = g \cdot f^{-1} \pmod{q}.$$

The **NTRU assumption** is that h looks like a uniformly random ring element, and that recovering the short pair (f, g) from h is hard.

```
q = 17
from random import choice
R.<x> = PolynomialRing(GF(q))
S = R.quotient(x^8 + 1)
def small_unit():
    f = S([choice([-1,0,1]) for _ in range(8)])
    while not f.is_unit():
        f = S([choice([-1,0,1]) for _ in range(8)])
    return f
```

```
f = small_unit()              # short and invertible
g = S([0,1,-1,0,1,0,0,-1])   # short
h = g * f^(-1)                # public key
print(h)
```

There is an equivalent lattice picture, and it is exactly the good-basis/bad-basis story of Part II. The *NTRU lattice*

$$\Lambda = \{(u, v) \in R_q^2 : u + v\, h \equiv 0 \pmod{q}\}$$

contains the short vector $(g, -f)$, because $g - f\, h = g - f\,(g/f) = 0$. From h alone one can only write down a *bad* (long) basis of Λ; the short pair (f, g) is a *good* basis. Recovering it is a shortest-vector problem in Λ.

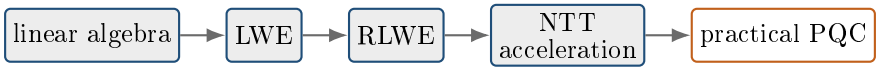
So RLWE and NTRU are two siblings in the same ring: RLWE hides its secret in additive noise, NTRU hides two short polynomials inside a quotient, and both reduce to finding short vectors in a structured lattice. We single NTRU out here because the Falcon signature (FIPS 206, Chapter 18) is built entirely on it.

12.16 Summary

The central idea is simple:

> RLWE replaces big random matrices by structured polynomials, which dramatically improves storage and speed while preserving the core noisy linear structure of LWE.

The evolution is:



where the RLWE step is precisely the move to a polynomial ring with negacyclic structure.

Chapter 13

Module-LWE (MLWE)

13.1 The three generations of lattice-based cryptography

We have already seen three major stages in the development of lattice-based cryptography.

- **First generation: LWE**. The equation is

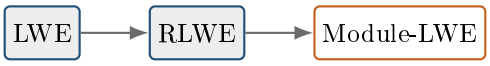
$$b = As + e,$$

  where A is a random matrix. The security proof is elegant, but the representation is large and the computation is costly.

- **Second generation: RLWE**. The matrix is replaced by a polynomial ring structure. Computation becomes much faster, but the algebraic structure becomes stronger and more specialized.

- **Third generation: Module-LWE**. This is a compromise between the large but simple LWE construction and the compact but highly structured RLWE construction.

The historical path is therefore:



13.2 What is a module?

The word "module" is not informal jargon; it is a precise algebraic structure, and Kyber's secrets live in one. It generalizes the vector space of Chapter 3 by letting the scalars come from a ring instead of a field.

Definition 13.1 (Module over a ring)**.** Let R be a commutative ring with identity 1. A (left) *R-module* is an abelian group $(M, +)$ together with a scalar multiplication $R \times M \to M$, written $(r, m) \mapsto rm$, such that for all $r, s \in R$ and $x, y \in M$:

1. $r(x + y) = rx + ry$;
2. $(r + s)x = rx + sx$;
3. $(rs)x = r(sx)$;

4. $1\,x = x$.

These are exactly the vector-space axioms with the field replaced by a ring. When R is a field, an R-module *is* a vector space; a module is the strictly more general object obtained by relaxing "field" to "ring." Several properties carry over, and one convenience is lost.

- **Free modules and rank.** The set $R^k = \{(m_1, \ldots, m_k) : m_i \in R\}$ with componentwise operations is an R-module, the *free module of rank k*. Like a vector space it has a basis – the standard unit vectors $e_1, \ldots, e_k$ – so every element is a unique R-combination of them. Kyber's secret space is exactly the free module R_q^k.
- **Matrices act as homomorphisms.** A matrix $A \in R^{k\times k}$ defines an R-linear map $R^k \to R^k$, $\mathbf{x} \mapsto A\mathbf{x}$, called an *$R$-module homomorphism*. The Module-LWE map $\mathbf{s} \mapsto A\mathbf{s}$ is one of these.
- **Submodules and ideals.** A submodule is the analogue of a subspace; the submodules of R regarded as a module over itself are precisely its *ideals* (Chapter 4) – which is why RLWE is said to rest on *ideal* lattices.
- **What is lost.** Over a general ring, not every module is free and not every submodule has a basis – a real departure from vector spaces. Kyber avoids this entirely by working in the free module R_q^k.

With the structure named precisely, the three-generation progression is a statement about *which module the secret lives in*:

$$\underbrace{s \in \mathbb{Z}_q^n}_{\text{LWE}} \qquad \underbrace{s(x) \in R_q}_{\text{RLWE}} \qquad \underbrace{\mathbf{s} \in R_q^k}_{\text{Module-LWE}} \quad .$$

LWE uses a vector of field elements; RLWE uses a single ring element; Module-LWE uses a length-k vector over the ring R_q – a free module of rank k.

13.3 Kyber's secret

In ML-KEM (Kyber), the secret is not a single polynomial but a vector of polynomials. The security level determines the dimension of that vector.

- Kyber512 uses $k = 2$.
- Kyber768 uses $k = 3$.
- Kyber1024 uses $k = 4$.

Thus, the security level is tied to the size of the module.

13.4 Why does the matrix come back?

At first glance, people are surprised to see matrices again in Kyber. But the matrix is not a large random matrix over integers. Instead, its entries are polynomials.

For example, for $k = 2$, we may have

$$A = \begin{bmatrix} a_{11}(x) & a_{12}(x) \\ a_{21}(x) & a_{22}(x) \end{bmatrix}.$$

Each entry is a polynomial in the ring R_q. The object is still a matrix, but the matrix entries are now ring elements rather than ordinary numbers.

13.5 Kyber public key generation

The public key generation is very similar to LWE in spirit. We choose a secret vector $\mathbf{s}$, a noise vector $\mathbf{e}$, and a polynomial matrix A, and then compute

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

This is exactly the same governing idea as LWE, except that the scalars are now ring elements. The figure below makes the shapes of A and $\mathbf{s}$ concrete for the smallest case $k = 2$ (indices run from 0 to $k - 1$, matching the Sage code below).

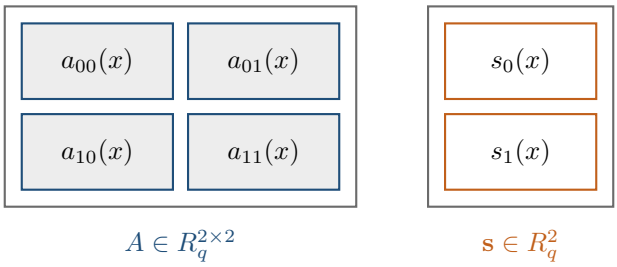


Figure 13.1: The module structure made concrete for $k = 2$: the public matrix $A \in R_q^{2\times 2}$ (blue, left) has four entries $a_{00}(x), a_{01}(x), a_{10}(x), a_{11}(x)$, each a full polynomial in R_q, and the secret $\mathbf{s} = (s_0(x), s_1(x))$ (orange, right) is a length-2 vector of such polynomials. Comparing this against the earlier "Secret object" column: plain LWE would put a single scalar in each box, and RLWE would collapse the whole picture to just one box.

13.6 Why is this better than plain LWE?

We can compare the three designs.

- **LWE**: the matrix is huge and the public key is large.
- **RLWE**: the representation is compact, but the algebraic structure is very strong.
- **Module-LWE**: the matrix is small and the elements are polynomials, giving a very good compromise between compactness and flexibility.

In other words, Module-LWE uses a small matrix of polynomials instead of a huge matrix of integers or a single large polynomial.

13.7 Sage construction of a toy module

We can create a very small toy example in Sage. First define the ring.

```
q = 17
R.<x> = PolynomialRing(GF(q))
S = R.quotient(x^8 + 1)
```

Now define a secret as a vector of two polynomials.

```
from random import choice
def small_poly():
    return S([choice([-1, 0, 1]) for _ in range(8)])

s = vector(S, [small_poly(), small_poly()])
print(s)
```

Create a small polynomial matrix.

```
A = Matrix(S, [[S.random_element(), S.random_element()],
               [S.random_element(), S.random_element()]])
print(A)
```

Create a noise vector.

```
e = vector(S, [small_poly(), small_poly()])
assert all(c in [0, 1, 16] for p in list(s) + list(e) for c in p.lift().list())
```

Then compute the public-like value.

```
t = A * s + e
print(t)
```

This has the algebraic shape of a Module-LWE sample. The public matrix is uniform, while the secret and error use small coefficients; it is still a toy example rather than the FIPS 203 sampler.

13.8 The Module-LWE problem

With the module structure in place, Module-LWE is the LWE distribution of Chapter 11 with the scalars promoted from $\mathbb{Z}_q$ to the ring R_q.

Definition 13.2 (Module-LWE)**.** Fix the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, a module rank k, and an error distribution χ over R_q concentrated on small elements. For a secret $\mathbf{s} \in R_q^k$, the *Module-LWE distribution* outputs

$$(\mathbf{a},\, b), \qquad \mathbf{a} \leftarrow R_q^k \text{ uniform}, \quad e \leftarrow \chi, \quad b = \langle \mathbf{a}, \mathbf{s} \rangle + e \in R_q.$$

Stacking m samples as the rows of $A \in R_q^{m \times k}$ gives the matrix form $(A,\ \mathbf{t} = A\mathbf{s} + \mathbf{e})$. *Search-MLWE* is to recover $\mathbf{s}$; *Decision-MLWE* is to distinguish $(A,\ A\mathbf{s} + \mathbf{e})$ from a uniform pair $(A,\ \mathbf{u})$.

Two special cases bracket the definition: $k = 1$ is RLWE (a single ring element), and $n = 1$ with $R_q = \mathbb{Z}_q$ recovers plain LWE. Module-LWE interpolates between them, and it inherits a worst-case hardness guarantee of the same flavour as LWE: solving it on average is at least as hard as standard problems on *module* lattices [20]. In Kyber the matrix is square ($m = k$), giving the compact form

$$\mathbf{t} = A\mathbf{s} + \mathbf{e}, \qquad A \in R_q^{k \times k}, \quad \mathbf{s}, \mathbf{e} \in R_q^k.$$

13.9 Why Module-LWE is considered a compromise

Many people wrongly think that Module-LWE is simply "better" because it uses more structure. That is not the correct understanding. The real point is that Module-LWE reduces the strongest algebraic structure of pure RLWE while retaining the efficiency advantages. In other words, it trades off between:

- the huge size of plain LWE,
- the strong structure of RLWE,
- and the practical middle ground provided by MLWE.

This balance is central to Kyber's design.

13.10 Why the Kyber parameters are $k = 2, 3, 4$

The values $k = 2, 3, 4$ correspond to different dimensions of the module, and are exactly the three security levels of Kyber [30, 20]. In practice:

- $k = 2$ gives Kyber512,
- $k = 3$ gives Kyber768,
- $k = 4$ gives Kyber1024.

Increasing k increases the size of the module and therefore increases the security level. The ring dimension and modulus stay fixed, but standardized ML-KEM parameter sets use different parameter tuples as well: in particular η_1 is $3, 2, 2$ and (d_u, d_v) is $(10, 4), (10, 4), (11, 5)$ for ML-KEM-512, -768, and -1024 respectively [1].

13.11 A complete toy MLWE experiment

A useful exercise is to implement a toy version of MLWE in full.

1. Define a ring R_q.
2. Create a small polynomial matrix A.
3. Create a polynomial vector $\mathbf{s}$.
4. Create a noise vector $\mathbf{e}$.
5. Compute $\mathbf{t} = A\mathbf{s} + \mathbf{e}$.
6. Print the result and inspect its structure.

This gives a practical understanding of why Kyber can be seen as a structured generalization of LWE. Written as pseudocode, using the same names A, $\mathbf{s}$, $\mathbf{e}$, $\mathbf{t}$ as the Sage code above, the same six steps are Algorithm 11.

Algorithm 11 Module-LWE Sample Generation

Input: Ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$; module dimension k
Output: MLWE sample $(A, \mathbf{t})$; secret $\mathbf{s}$

$A \leftarrow$ uniformly random $k \times k$ matrix over R_q
$\mathbf{s} \leftarrow$ length-k vector of small (noise-sized) elements of R_q ▷ the secret
$\mathbf{e} \leftarrow$ length-k vector of small (noise-sized) elements of R_q ▷ the error
$\mathbf{t} \leftarrow A\mathbf{s} + \mathbf{e}$ ▷ a vector of k ring elements, not one scalar
return $(A, \mathbf{t})$ as the sample, $\mathbf{s}$ as the secret

Real ML-KEM has this algorithmic shape with $n = 256$ and $k = 2, 3, 4$, but the standardized parameter sets differ in more than k; their sampling and compression parameters are specified in FIPS 203.

13.12 Summary

The key lesson is that Module-LWE is not merely "LWE with more polynomials". It is the middle ground between the two extremes:

Scheme	Secret object	Main idea
LWE	vector over $\mathbb{Z}_q$	simple and general, but large
RLWE	single polynomial over R_q	compact and fast, but highly structured
Module-LWE	vector of polynomials over R_q	balanced compromise, used by Kyber

The important insight is that Kyber is not simply RLWE with a different name. It is the module version of the same noisy linear structure, arranged in a way that gives practical efficiency while preserving strong security assumptions.

Building ML-KEM (FIPS 203)

Chapter 14

Introduction to Kyber (ML-KEM)

14.1 What is Kyber?

Kyber – standardized by NIST as **ML-KEM** in FIPS 203 [1, 30] – is the post-quantum replacement for the key exchange that secures most of the internet today. It is a *key-encapsulation mechanism* (KEM): a protocol by which two parties establish a shared secret over a public channel, taking the role that Diffie–Hellman and RSA key transport played in the pre-quantum world. Its hardness rests on Module-LWE (Chapter 13), the middle member of the LWE family developed in the previous chapters.

A KEM is defined by three algorithms:

- $\textsc{KeyGen} \to (ek, dk)$ – produce a public *encapsulation key* ek and a secret *decapsulation key* dk;
- $\textsc{Encaps}(ek) \to (K, c)$ – generate a fresh shared secret K together with a ciphertext c that carries it;
- $\textsc{Decaps}(dk, c) \to K$ – recover the same shared secret K from the ciphertext.

Crucially, a KEM transports a *random* secret rather than a chosen message. That single design choice is what later lets the Fujisaki–Okamoto transform upgrade Kyber to security against active, chosen-ciphertext attackers.

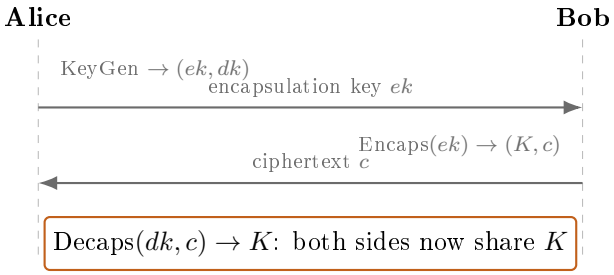


Figure 14.1: Kyber as a key-encapsulation mechanism. Alice publishes an encapsulation key; Bob encapsulates a fresh random secret K under it and sends only the ciphertext c; Alice decapsulates c with her secret key to recover the same K. The shared secret K itself never crosses the wire.

14.2 The pieces, and where this book builds them

Kyber is not a new primitive. It is a careful assembly of parts developed across this book, and this chapter is where they first come together – in miniature. Some pieces are already in hand (the Module-LWE relation, the ring, the NTT); the toy below still shortcuts the others, which the cited chapters supply in full.

Ingredient	Role in Kyber	Built in
Module-LWE relation $\mathbf{t} = A\mathbf{s} + \mathbf{e}$	the underlying hard problem	Chapter 13
Ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ and NTT	fast polynomial arithmetic	Chapters 12, 6
Centered binomial noise (CBD)	sampling small secrets/errors	Chapter 15
Fujisaki–Okamoto transform	lifts CPA security to CCA	Chapter 16
Full FIPS 203 pipeline	KeyGen / Encaps / Decaps + FO	Chapter 16

Table 14.1: Kyber is an assembly of pieces developed across this book; this introductory chapter shows the skeleton, and the cited chapters supply each real component.

The three standardized parameter sets ML-KEM-512, ML-KEM-768, and ML-KEM-1024 share the same ring but use different parameter tuples: $k = 2, 3, 4$, $\eta_1 = 3, 2, 2$, and $(d_u, d_v) = (10, 4), (10, 4), (11, 5)$ respectively [1]. ML-KEM-768 is a common default; this chapter retains only a stripped-down algebraic skeleton.

14.3 Building a toy Kyber

To see the skeleton before the machinery, we now build a *toy* Kyber in about a hundred lines of SageMath – small enough to inspect every object by hand. Using only mathematics already covered, it generates a public key, encrypts a message, and decrypts it:

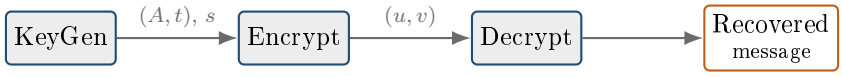


Figure 14.2: The toy pipeline built in this chapter: key generation produces a public key (A, t) and secret key s; encryption turns a message bit into a ciphertext (u, v); decryption recovers the bit using s. This is the public-key-encryption *core*; the full KEM of Figure 14.1 wraps exactly this core with the Fujisaki–Okamoto transform in Chapter 16.

The goal is not to reproduce the full FIPS 203 standard here, but to expose its mathematical skeleton clearly. Everything runs inside SageMath in around one hundred lines so that the structure stays visible.

14.4 Simplifications used in the toy version

The real ML-KEM standard uses many sophisticated mechanisms. In the toy version, we keep only the core idea:

Real ML-KEM	Toy Kyber
$q = 3329$	$q = 17$
$n = 256$	$n = 8$
CBD noise	randint$(-1, 1)$
SHAKE/XOF	simple random generation
Compression	omitted
NTT	ordinary polynomial multiplication
FO transform	omitted
Byte encoding	omitted

Although these simplifications are significant, the mathematical skeleton remains the same. For the module dimension we also shrink k from $2, 3, 4$ down to $k = 2$, so that the objects are easy to inspect by hand.

```
from random import randint

q = 17
n = 8
k = 2
```

14.5 Set up the ring

We define a polynomial ring modulo x^8+1. Every object we build from here on lives in this quotient ring R_q.

```
R.<x> = PolynomialRing(GF(q))
Rq = R.quotient(x^n + 1)
```

```
a = Rq.random_element()
b = Rq.random_element()
print(a)
print(b)
print(a*b)
```

The highest degree should remain below 8 because of the quotient relation.

14.6 Build the module: vectors and matrices of polynomials

For Kyber512, the module dimension is $k = 2$. That means the secret is not one polynomial but a vector of k polynomials, and the public matrix A is a $k \times k$ matrix of polynomials.

```
def rand_poly():
    return Rq.random_element()

def rand_vector():
    return vector(Rq, [rand_poly() for _ in range(k)])

def rand_matrix():
    return Matrix(Rq, k, k, [rand_poly() for _ in range(k*k)])
```

A vector of polynomials is already the first step toward the module viewpoint, and the matrix is the object that appears inside Kyber-like constructions.

14.7 Create small noise

In real Kyber, the noise is sampled from a centered binomial distribution (CBD), which is the subject of a later chapter. For the toy version, we simply use small coefficients in the range $\{-1, 0, 1\}$.

```
def small_poly():
    coeffs = []
    for _ in range(n):
        c = randint(-1, 1)
        coeffs.append(c % q)
    return Rq(coeffs)

def small_vector():
    return vector(Rq, [small_poly() for _ in range(k)])
```

The polynomial entries are therefore small values such as 0, 1, or 16 (which represents -1 in $\mathbb{F}_{17}$). This produces small errors, which are suitable for teaching.

14.8 Key generation

The module-LWE form is very close to the standard LWE form:

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

We implement it as follows.

```
def keygen():
    A = rand_matrix()
    s = small_vector()
    e = small_vector()
    t = A * s + e
    return (A, t), s
```

The public key is (A, t) and the secret key is s.

Algorithm 12 Toy KeyGen

Input: Public parameters q, n, k, R_q
Output: Public key (A, t), secret key s

1: $A \leftarrow$ uniformly random $k \times k$ matrix over R_q
2: $s \leftarrow$ small_vector() ▷ coefficients in $\{-1, 0, 1\}$
3: $e \leftarrow$ small_vector()
4: $t \leftarrow As + e$
5: **return** $\big((A, t),\ s\big)$

14.9 Verify the key generation process

We can print the matrix and the resulting public value to check that the shapes and types match expectations.

```
(pk, sk) = keygen()
A, t = pk
print("A =")
```

```
print(A)
print("\nSecret:")
print(sk)
print("\nPublic:")
print(t)
```

This already looks very close to the mathematical object used in Kyber.

14.10 Encoding a message bit

For the toy version, the message is reduced to a single bit. In the real scheme, message encoding is far more subtle; here we simply map a bit to either zero or a value near the middle of the modulus range.

```
def encode_bit(bit):
    if bit == 0:
        return Rq(0)
    else:
        return Rq(q // 2)
```

This is enough to demonstrate the basic idea of hiding a bit inside the noisy linear equation.

14.11 Encryption

To encrypt a message bit, we sample a random vector r and two noise terms e_1, e_2. The core equations are

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

$$v = t^T r + e_2 + m.$$

```
def encrypt(pk, bit):
    A, t = pk
    r = small_vector()
    e1 = small_vector()
    e2 = small_poly()
    u = A.transpose() * r + e1
    m = encode_bit(bit)
    v = t.dot_product(r) + e2 + m
    return (u, v)
```

The ciphertext is the pair (u, v).

Algorithm 13 Toy Encrypt

Input: Public key (A, t); message bit $m \in \{0, 1\}$
Output: Ciphertext (u, v)

1: $r \leftarrow \text{small\_vector}(), \quad e_1 \leftarrow \text{small\_vector}(), \quad e_2 \leftarrow \text{small\_poly}()$
2: $u \leftarrow A^T r + e_1$
3: $v \leftarrow t^T r + e_2 + \textsc{Encode}(m)$ ▷ $\textsc{Encode}(0) = 0,\ \textsc{Encode}(1) = \lfloor q/2 \rfloor$
4: **return** (u, v)

14.12 Decryption

The decryption step uses the secret key s. The main idea is that the terms involving A and s cancel out during the algebraic expansion, leaving only a small amount of noise and the encoded message.

```
def decrypt(sk, cipher):
    u, v = cipher
    value = v - sk.dot_product(u)
    return value
```

We recover the bit by comparing circular distances to the two encoding centres, 0 and $\lfloor q/2 \rfloor$. In particular, the field element 16 represents -1 and must be recognised as close to encoded zero, not as a message-one value.

```
def recover_bit(value):
    c = value.list()[0]
    d0 = min(c, q - c)
    d1 = min((c - q//2) % q, (q//2 - c) % q)
    return 0 if d0 < d1 else 1
```

Algorithm 14 Toy Decrypt

Input: Secret key s; ciphertext (u, v)
Output: Recovered bit $m' \in \{0, 1\}$

1: $w \leftarrow v - s^T u$ ▷ the A, s terms cancel, leaving noise + message
2: $c \leftarrow$ constant coefficient of w in $\{0, \ldots, q-1\}$
3: $d_0 \leftarrow \min(c, q-c)$; $\quad d_1 \leftarrow \min((c - \lfloor q/2 \rfloor) \bmod q, (\lfloor q/2 \rfloor - c) \bmod q)$
4: **return** 0 if $d_0 < d_1$, otherwise 1

14.13 Putting everything together

A complete toy run looks like this.

```
# Key generation
(pk, sk) = keygen()

# Encryption
bit = 1
cipher = encrypt(pk, bit)

# Decryption
value = decrypt(sk, cipher)
print("decrypted value =", value)
print("recovered bit =", recover_bit(value))
```

In many runs this returns the correct bit because the noise is deliberately kept small.

14.14 Why the cancellation works

The important algebraic idea is that the public key satisfies

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

so during decryption one computes

$$\mathbf{v} - \mathbf{s}^T\mathbf{u}.$$

After expansion, the terms involving A and s cancel, leaving only a small error term and the message. This is the core of the Kyber-like construction.

14.15 Why the scheme can fail sometimes

This toy construction can still fail frequently because $q = 17$ is tiny and the accumulated error is not bounded as in ML-KEM. It illustrates cancellation and circular decoding, not a reliable encryption scheme. In real ML-KEM, the parameters are tuned so that the decryption failure probability is extremely small.

A simple experiment is to change the noise generator from

```
randint(-1, 1)
```

to something larger, such as

```
randint(-2, 2)
```

or even

```
randint(-5, 5)
```

and observe that decryption errors become much more frequent. This directly illustrates the role of the noise parameter, and foreshadows why real ML-KEM controls its noise distribution so carefully.

14.16 What is still missing compared with real Kyber?

This toy implementation already captures the mathematical skeleton of ML-KEM. However, it is still far from the full standard. The following components are not yet present:

- CBD sampling,
- SHAKE128 / SHAKE256 and XOF-based expansion,
- NTT-based polynomial multiplication,
- compression and encoding,
- Fujisaki–Okamoto transformation,
- CCA security behavior,
- constant-time implementation details.

These will be introduced gradually in later chapters.

14.17 Summary

This toy implementation captures the core idea of Kyber in a compact form. Starting from the Module-LWE relation, key generation forms $\mathbf{t} = A\mathbf{s} + \mathbf{e}$; encryption forms $\mathbf{u} = A^T\mathbf{r} + \mathbf{e}_1$ and $v = \mathbf{t}^T\mathbf{r} + e_2 + m$; and decryption computes $v - \mathbf{s}^T\mathbf{u}$, recovering the message from the resulting small, noisy value.

The most important lesson is that Kyber is not a completely new primitive. It is a carefully engineered version of the same noisy linear structure that we have been studying throughout the book. The next chapters replace the remaining shortcuts: real CBD noise sampling (Chapter 15), and then the complete FIPS 203 implementation – key generation, encapsulation, decapsulation, and the Fujisaki–Okamoto transform – in Chapter 16.

Chapter 15

Centered Binomial Distribution (CBD) – Why Kyber does not use Gaussian

15.1 A very important chapter

This chapter is one of the most underestimated chapters in the whole Kyber story. Many introductions to Kyber say only that *CBD is just counting a few bits*. That is true, but it misses the deeper point. The real story is that CBD is not merely a convenient trick. It is a design choice that connects lattice cryptography, implementation security, hardware efficiency, and standardization history.

The central question is simple:

> Why did NIST ultimately choose CBD instead of the more traditional discrete Gaussian for ML-KEM?

By the end of this chapter, you should be able to explain what CBD is, why it is close to a Gaussian, what the parameter η means, how FIPS 203 Algorithm 2 works, and why CBD is so attractive for constant-time implementations.

15.2 Recall the LWE viewpoint

In LWE, the core object is noise:

$$\mathbf{b} = A\mathbf{s} + \mathbf{e}.$$

The security does not come from the matrix A alone, nor from the secret $\mathbf{s}$ alone. The crucial ingredient is the noise vector $\mathbf{e}$.

In other words, the whole cryptographic hardness is tied to the fact that the error term hides the secret in a way that is hard to recover. A good sampling distribution for that error is therefore essential.

15.3 The main problem with Gaussian sampling

In the original theoretical LWE constructions, one often uses a discrete Gaussian distribution. A sample is drawn from a distribution of the form

$$\Pr[X = x] \propto e^{-x^2/(2\sigma^2)}.$$

This is mathematically elegant, but from an engineering point of view it is painful.

The practical issue is that Gaussian sampling usually requires expensive operations:

- computing exponentials;
- comparing floating-point values;
- rejection sampling;
- repeated loops until a sample is accepted.

This is slow, especially in software. It is also more difficult to make constant-time.

A very important side effect appears in implementation security. If a sampler uses rejection loops and branching, the running time may depend on the sampled value. This opens the door to timing attacks, where an attacker observes the timing pattern and tries to infer information about the hidden noise.

For a cryptosystem designed for real-world deployment, this is a serious drawback.

15.4 Kyber's key idea

Kyber does not try to use the idealized Gaussian in the naive way. Instead, it asks a more practical question:

> Can we construct a distribution that looks like a Gaussian, but is made only from small integers, bits, and simple arithmetic?

The answer is yes, and that answer is CBD.

CBD is simple, integer-only, and very friendly to constant-time implementation. It is much easier to deploy on hardware and software than a Gaussian sampler.

15.5 The simplest example

Let us begin with the smallest nontrivial case: $\eta = 2$.

Suppose we sample four random bits:

```
1010
```

Split them into two groups of two bits:

```
10 10
```

Count the number of ones in each half:

- first half: $10 \Rightarrow 1$
- second half: $10 \Rightarrow 1$

Now subtract:

$$1 - 1 = 0.$$

This gives one sampled value.
Now consider another example:

```
1110
```

Then

- first half: $11 \Rightarrow 2$
- second half: $10 \Rightarrow 1$

and therefore

$$2 - 1 = 1.$$

Finally, take

```
0011
```

Then

- first half: $00 \Rightarrow 0$
- second half: $11 \Rightarrow 2$

so

$$0 - 2 = -2.$$

Thus the possible values are

$$-2, -1, 0, 1, 2.$$

15.6 Why it is called centered

The key word in the name is *centered*.

In a large number of samples, the positive outcomes and negative outcomes balance out. For example, values such as -2 and 2 are rare, while 0 is the most common. The distribution is centered around zero.

In fact, the mean is exactly zero because the two halves contribute equally in expectation.

So the distribution is centered around zero, and that is exactly what we need for LWE noise.

15.7 Why it becomes increasingly Gaussian-like

The beautiful mathematical reason is that each bit behaves like a Bernoulli random variable:

$$B_i \in \{0,1\}, \qquad \Pr(B_i = 0) = \Pr(B_i = 1) = \frac{1}{2}.$$

Now consider the sum of many such Bernoulli variables. By the central limit theorem, the sum of many independent Bernoulli variables becomes approximately Gaussian.

CBD is built from exactly this phenomenon. It is not a crude heuristic. It is a discrete distribution that is mathematically close to a Gaussian because it is formed from the difference of two binomial counts.

So the real logic is:

> CBD is not merely *trying* to look like Gaussian. It is a discrete construction whose shape naturally approaches a Gaussian-like bell curve.

15.8 What does η mean?

The parameter η controls the size of the sample.

If $\eta = 2$, we use two bits in the first half and two bits in the second half.

If $\eta = 3$, we use three bits in each half.

In general, we define

$$X = \sum_{i=1}^{\eta} B_i - \sum_{i=\eta+1}^{2\eta} B_i,$$

where each B_i is a random bit.

The resulting range is

$$-\eta, -\eta + 1, \ldots, \eta.$$

So larger η means larger noise magnitude and therefore a broader distribution.

15.9 Real ML-KEM parameter choices

In the actual ML-KEM parameter sets, the values of η_1 and η_2 are not arbitrary. They are chosen according to security and decryption failure requirements.

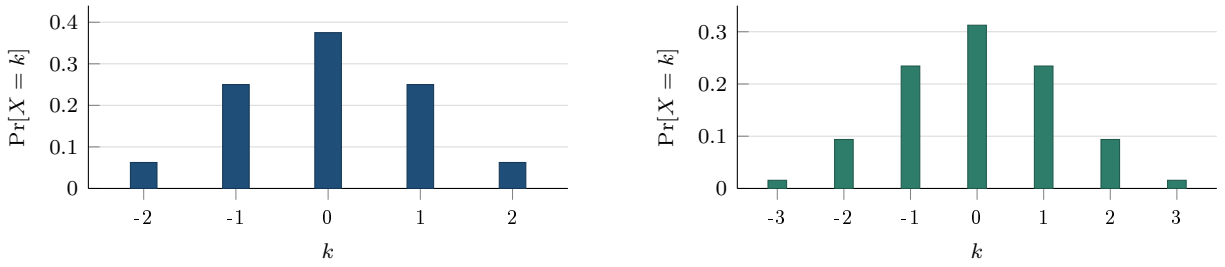
Parameter set	η_1	η_2
ML-KEM-512	3	2
ML-KEM-768	2	2
ML-KEM-1024	2	2

Table 15.1: Typical CBD parameters used in ML-KEM.

Here η_1 is typically used for the secret distribution, while η_2 is used for the encryption noise.

The important point is that these choices are not simply *make it larger for more security.* They are carefully tuned to balance security, correctness, and implementation efficiency.

Figure 15.1 plots the exact probability mass function for both values of η that actually appear above, computed directly from $\Pr[\mathrm{CBD}(\eta) = k] = \binom{2\eta}{\eta+k}/4^{\eta}$.



(a) $\mathrm{CBD}(\eta = 2)$ (b) $\mathrm{CBD}(\eta = 3)$

Figure 15.1: Exact probability mass functions of the centered binomial distribution for the two η values that appear in ML-KEM-512/768/1024 (table above), computed from $\Pr[\mathrm{CBD}(\eta) = k] = \binom{2\eta}{\eta+k}/4^{\eta}$: $\frac{1}{16}, \frac{4}{16}, \frac{6}{16}, \frac{4}{16}, \frac{1}{16}$ for $\eta = 2$, and $\frac{1}{64}, \frac{6}{64}, \frac{15}{64}, \frac{20}{64}, \frac{15}{64}, \frac{6}{64}, \frac{1}{64}$ for $\eta = 3$. Increasing η from 2 to 3 widens and flattens the bell shape – more of the probability mass moves away from 0 – at the cost of drawing twice as many random bits per coefficient.

15.10 FIPS 203 Algorithm 2: what is happening?

In FIPS 203, the standard describes a sampling procedure that is essentially the CBD sampling operation.

The high-level story is very simple:

- input a bit string of length 2η;
- split it into two groups of length η;
- count the number of ones in each group;
- subtract the two counts.

So the procedure really is:

$$\mathrm{CBD}_{\eta}(x) = \sum_{i=0}^{\eta-1} x_i - \sum_{i=\eta}^{2\eta-1} x_i.$$

In plain words: a short input bit string is converted into a small signed integer noise sample.

That is exactly the operation that Kyber relies on in its practical implementation.

Written out precisely, with exact bit indices instead of the informal "split it into two groups" description above, this is FIPS 203's Algorithm 2, SamplePolyCBD$_{\eta}$. It does not sample one coefficient at a time from a fresh bit string; it consumes one long byte array B of 64η bytes and slices it into 256 non-overlapping windows of 2η bits each, one window per output coefficient.

Algorithm 15 SamplePolyCBD$_\eta$ (FIPS 203, Algorithm 2)

Input: Byte array $B \in \{0, \ldots, 255\}^{64\eta}$
Output: Polynomial $f \in \mathbb{Z}_q^{256}$ with coefficients $f_0, \ldots, f_{255}$

$b \leftarrow \text{BytesToBits}(B)$ ▷ $b[0], \ldots, b[8 \cdot 64\eta - 1]$; only the first $2\eta \cdot 256$ bits are used
for $i = 0$ **to** 255 **do**
  $x \leftarrow \sum_{j=0}^{\eta-1} b[2i\eta + j]$ ▷ ones in the first η-bit half of window i
  $y \leftarrow \sum_{j=0}^{\eta-1} b[2i\eta + \eta + j]$ ▷ ones in the second η-bit half of window i
  $f_i \leftarrow (x - y) \bmod q$
end for
return $f = (f_0, \ldots, f_{255})$

Every coefficient f_i therefore reads its own private 2η-bit slice of b, starting at bit offset $2i\eta$: the first η bits of the slice are counted into x, the next η bits into y, and $f_i = x - y \bmod q$. For $i = 0$ this is exactly the "first half, second half, subtract" recipe worked out by hand above; the only new content is the explicit offset $2i\eta$ that tells you, for every one of the 256 coefficients, precisely which bits of the 64η-byte array belong to it.

15.11 A simple Sage implementation

Here is a very small version in Python.

```
from random import randint


def cbd(eta):
    a = 0
    b = 0

    for _ in range(eta):
        a += randint(0, 1)

    for _ in range(eta):
        b += randint(0, 1)

    return a - b
```

A quick test is:

```
for _ in range(20):
    print(cbd(2))
```

Typical output might be

```
0
1
-1
0
2
-2
```

15.12 Histogram view

To understand CBD properly, one should look at the histogram.

```
samples = [cbd(2) for _ in range(100000)]
```

The distribution tends to concentrate around zero and becomes increasingly bell-shaped. This is exactly why the distribution is often described as a discrete approximation to a Gaussian.

A very simple visualization can be done with `matplotlib`:

```
from collections import Counter
import matplotlib.pyplot as plt

samples = [cbd(2) for _ in range(100000)]
cnt = Counter(samples)

plt.bar(cnt.keys(), cnt.values())
plt.show()
```

If we try $\eta = 2$, $\eta = 3$, and $\eta = 4$, we can see the shape become smoother and more symmetric.

15.13 Why the standard does not use `randint`

A subtle but important point is that the real FIPS implementations do not literally use Python's `randint(0, 1)` style calls.

Instead, the standard uses a deterministic expansion from a seed through an extendable-output function such as SHAKE.

Instead, the seed is expanded deterministically: SHAKE turns it into a pseudorandom bit stream, and CBD turns those bits into the noise sample. The full pipeline is shown in Section 15.16. This design makes the noise reproducible from the seed and straightforward to integrate into a larger protocol.

15.14 Why CBD is constant-time friendly

One of the largest practical advantages of CBD is that it is easy to implement in constant time.

The operations are only:

- bit counts;
- additions;
- subtractions.

There is no expensive rejection loop and no data-dependent `if` branching in the basic procedure.

This makes CBD much easier to protect against timing attacks than a Gaussian sampler with rejection steps.

15.15 A small upgrade to the toy Kyber code

If we had earlier written a toy sampler that generated small values from a simple routine, then we can now replace it with CBD-style sampling.

For example, we might write something like:

```
def small_poly(n):
    coeff = []
    for _ in range(n):
        coeff.append(cbd(2))
    return coeff
```

In this way, the toy Kyber code becomes much closer to the actual ML-KEM design.

15.16 The complete noise pipeline in FIPS 203

We can now assemble the full path that noise takes in FIPS 203, from the initial seed to a Module-LWE sample.

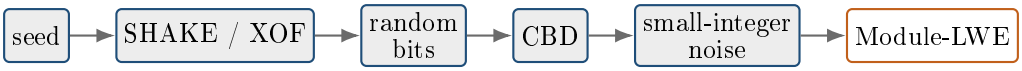


Figure 15.2: The full noise pipeline in FIPS 203. A short seed is expanded by SHAKE into a pseudorandom bit stream; CBD converts those bits into small signed integers; and those integers become the secret and error of a Module-LWE sample. Every step is deterministic and constant-time.

The noise is therefore not a mysterious floating-point object. It is a small integer, derived deterministically from a seed through a secure expansion procedure, and every step along the way is reproducible and constant-time – exactly what a standard requires.

15.17 Summary

The important lesson of this chapter is that CBD is not just a convenience trick. It is a highly practical sampling method that combines four useful properties:

1. It is approximately Gaussian-shaped.
2. It is simple to implement.
3. It is easy to make constant-time.
4. It is well suited to software and hardware deployment.

This is why CBD became the standard choice in ML-KEM rather than the more theoretically traditional Gaussian sampler.

In one sentence:

> CBD delivers a noise distribution that is good enough for lattice-based security, easy to implement, and much easier to protect against side-channel attacks than Gaussian sampling.

15.18 Suggested next step

The next natural topic is the internal hash and expansion layer: why SHAKE and Keccak are so powerful, and how a short seed can generate a huge amount of pseudorandom material. That is the bridge from CBD to the actual FIPS 203 and FIPS 204 implementations.

Chapter 16

Complete Implementation of ML-KEM (FIPS 203)

16.1 Learning objectives

This chapter assembles the pieces built earlier in the book – the negacyclic ring and the NTT (Chapter 6), the toy scheme (Chapter 14), and centered binomial sampling (Chapter 15) – into the real standard. FIPS 203, **ML-KEM** (from the submission CRYSTALS-Kyber) [1, 30], is the module-lattice-based key-encapsulation standard, and the first of NIST's post-quantum standards to be finalized (2024). All algorithm and data-flow details in this chapter follow FIPS 203 [1].

After finishing this chapter, you should be able to:

1. follow the full ML-KEM key generation workflow and its data types at every stage;
2. explain why key generation is almost entirely deterministic once the initial seed is fixed;
3. follow encapsulation and explain why a KEM encapsulates a *random* secret rather than encrypting chosen data;
4. follow decapsulation and explain the Fujisaki–Okamoto "re-encrypt and compare" transform with implicit rejection;
5. distinguish the CPA-secure inner scheme (K-PKE) from the CCA-secure KEM (ML-KEM) built on top of it.

16.2 Review: Toy Kyber versus real ML-KEM

In the previous toy implementation, we used the following informal key-generation pattern:

$$A \xleftarrow{\$} R_q^{k\times k}, \qquad s, e \xleftarrow{\$} \chi, \qquad t = A \cdot s + e.$$

The output was then a key pair (pk, sk).

The most important difference is that all random objects were generated directly and independently.

In real ML-KEM, this is not the case. The algorithm starts from a short random seed and then deterministically expands the rest of the randomness from it.

16.3 The real input in FIPS 203

The standard key generation procedure does not begin by generating a matrix at random. It begins by generating a short random seed:

$$d \in \{0,1\}^{256}.$$

In other words, the input to the standard algorithm is a 256-bit seed. We can think of the whole key generation algorithm as a deterministic function

$$\mathrm{KeyGen}(d)$$

rather than as a random process that samples all objects independently.
This is the first major conceptual shift from toy Kyber to the real standard.

16.4 Phase 1: expand the seed

The first step in the standard is to compute

$$(\rho, \sigma) = G(d),$$

where:

- ρ is used to generate the matrix A;
- σ is used to generate the secret and error polynomials.

In the standard, G is built from SHA3-512. In a teaching implementation, we can first abstract it as a black box:

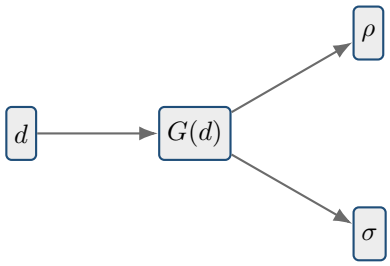


Figure 16.1: G expands the single seed d into two independent-looking seeds: ρ drives the matrix, σ drives the noise.

The pedagogical point is that the entire randomness of the later steps is derived from the initial seed d.

16.5 Phase 2: generate the matrix

The matrix is not stored as a random object in the public key. Instead, it is reconstructed from a seed:

$$A = \mathrm{ExpandA}(\rho).$$

This is a deterministic map:

$$\text{ExpandA} : \{0,1\}^{256} \to R_q^{k \times k}.$$

The same seed always yields the same matrix. This is a crucial compression trick in the standard.

SageMath experiment For a teaching implementation, we can first simulate the expansion by using a simple deterministic pseudo-random generator.

```
from random import Random
from sage.all import GF, Matrix, PolynomialRing

q = 17
n = 8
k = 2
R.<x> = PolynomialRing(GF(q))
Rq = R.quotient(x^n + 1)


def expand_A(seed):
    rng = Random(str(seed))
    return Matrix(
        Rq,
        k,
        k,
        [Rq([rng.randrange(q) for _ in range(n)])
         for _ in range(k * k)]
    )
```

The important thing is not the exact random source at this stage. The important thing is the interface:

- same seed $\Rightarrow$ same matrix;
- different seed $\Rightarrow$ different matrix.

16.6 Phase 3: generate the secret vector

The secret vector is derived from the CBD distribution. In other words,

$$s \leftarrow \text{CBD}(\sigma).$$

This is not uniform sampling. The coefficients are small integers sampled from a centered binomial distribution. Each coefficient of the secret polynomial is therefore a small signed value rather than an arbitrary element of the ring.

The secret vector is therefore a vector of small polynomials:

$$s = (s_0, \ldots, s_{k-1}).$$

16.7 Phase 4: generate the error vector

The error vector is generated in a very similar way, again using CBD-based sampling:

$$e \leftarrow \mathrm{CBD}(\sigma).$$

The exact detail depends on the parameter set, but the conceptual point is the same: the noise is derived from a short seed through a deterministic sampling procedure.

16.8 Phase 5: compute the public key in the NTT domain

FIPS 203 samples the secret and error in coefficient form, then transforms them. ExpandA already produces the matrix in the NTT representation. The public vector is computed as

$$\hat{t} = \hat{A} \circ \hat{s} + \hat{e},$$

where $\circ$ uses the specified MultiplyNTTs operation: 128 quadratic base products, not unrestricted coordinate-wise scalar multiplication. The PKE public key encodes $(\hat{t}, \rho)$ and the PKE secret key encodes $\hat{s}$; an inverse NTT is used later where coefficient-domain values are required. This is the data flow of FIPS 203, Sections 4.3.1 and 5.1.

16.9 Phase 7: encode the public key

The real public key is not the pair (A, t). Instead, it is essentially the pair

$$(\rho, t).$$

The reason is that the matrix A can be regenerated from the seed ρ. Therefore the public key stores only what is truly necessary:

- the seed ρ;
- the vector t.

16.10 KeyGen data flow

The whole process can be summarized as follows:

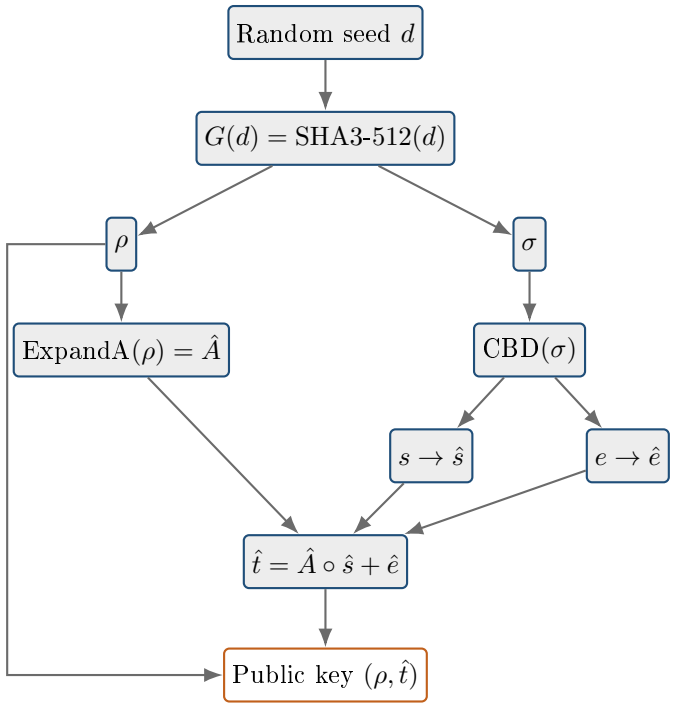


Figure 16.2: The K-PKE KeyGen data flow. The seed d expands into ρ and the sampling seed; ExpandA produces $\hat{A}$, while sampled coefficient-domain vectors are transformed before the NTT-domain multiplication. The public key stores ρ and $\hat{t}$, never A itself.

This flow should be remembered because it reappears inside encapsulation and decapsulation below. Written as pseudocode, the seven phases above collapse into a single short algorithm:

Algorithm 16 K-PKE.KeyGen (FIPS 203 [1], teaching form)

Input: Random seed $d \in \{0,1\}^{256}$
Output: Encoded public key $(\rho, \hat{t})$; encoded secret key $\hat{s}$

1: $(\rho, \sigma) \leftarrow G(d)$ ▷ Phase 1: seed expansion, $G = \text{SHA3-512}$
2: $\hat{A} \leftarrow \text{ExpandA}(\rho)$ ▷ NTT representation of the deterministic matrix
3: $s \leftarrow \text{CBD}_{\eta_1}(\sigma)$ ▷ Phase 3: secret vector
4: $e \leftarrow \text{CBD}_{\eta_1}(\sigma)$ ▷ Phase 4: error vector
5: $\hat{s}, \hat{e} \leftarrow \text{NTT}(s), \text{NTT}(e)$
6: $\hat{t} \leftarrow \hat{A} \circ \hat{s} + \hat{e}$ ▷ use MultiplyNTTs for every ring product
7: **return** $\big((\rho, \hat{t}),\ \hat{s}\big)$ ▷ encode the NTT-domain values as specified

Every line here is deterministic given d; the only place fresh randomness enters the entire computation is the sampling of d itself.

16.11 From K-PKE to ML-KEM: the two layers

The algorithm above is not quite ML-KEM. It is the key generation of **K-PKE**, the underlying *public-key encryption* scheme, which is only *CPA-secure* – safe against a passive eavesdropper, but not against an attacker who submits chosen ciphertexts and watches how decryption responds.

ML-KEM is K-PKE wrapped in one more layer. K-PKE provides three routines,

$$\text{K-PKE.KeyGen}, \qquad \text{K-PKE.Encrypt}(ek, m, r), \qquad \text{K-PKE.Decrypt}(dk, c),$$

and the **Fujisaki–Okamoto (FO) transform** turns them into a *CCA-secure* key-encapsulation mechanism. ML-KEM key generation simply calls K-PKE key generation and stores a little extra in the secret key:

Algorithm 17 ML-KEM.KeyGen (teaching form)

Input: Independent random values $d, z \in \{0,1\}^{256}$
Output: Encapsulation key ek; decapsulation key dk

1: $\big((\rho, t),\ dk_{\text{PKE}}\big) \leftarrow \text{K-PKE.KeyGen}(d)$ ▷ Algorithm 16
2: $ek \leftarrow (\rho, t)$
3: **return** ek, $dk \leftarrow \big(dk_{\text{PKE}},\ ek,\ H(ek),\ z\big)$

The encapsulation key $ek = (\rho, t)$ is what a KEM publishes. The decapsulation key carries the PKE secret $dk_{\text{PKE}} = s$, a copy of ek and its hash $H(ek)$ (both needed to re-encrypt during decapsulation), and the random value z whose role becomes clear below.

16.12 Encapsulation

Encapsulation produces a fresh *shared secret* K and a ciphertext c from which the holder of dk can recover the same K. The design choice that defines a KEM – and the answer to "why encapsulate a random secret instead of encrypting a chosen message?" – is that the encrypted message is *not chosen by the sender at all.* Encapsulation samples a random 32-byte value m, and derives both the shared key and the encryption randomness by *hashing* it. That the randomness is a deterministic function of the message is exactly what the FO transform will exploit.

Algorithm 18 ML-KEM.Encaps (teaching form)

Input: Encapsulation key ek
Output: Shared secret K; ciphertext c

1: $m \leftarrow \{0,1\}^{256}$ ▷ fresh random message, not chosen data
2: $(K, r) \leftarrow G\big(m \,\|\, H(ek)\big)$ ▷ G = SHA3-512: shared key K and encryption seed r
3: $c \leftarrow \text{K-PKE.Encrypt}(ek, m, r)$
4: **return** (K, c)

Inside K-PKE.Encrypt the message is hidden in the same noisy-linear way as the toy scheme of Chapter 14, now driven by the seed r: the matrix is regenerated as $A = \text{ExpandA}(\rho)$, small vectors y, e_1, e_2 are sampled from CBD using r, and the ciphertext is the compressed pair

$$u = A^T y + e_1, \qquad v = t^T y + e_2 + \text{Encode}(m).$$

The shared secret K itself never travels on the wire; only c does, and the recipient reconstructs K from it.

16.13 Decapsulation and the Fujisaki–Okamoto transform

Decapsulation must recover K from c using dk, and – this is the entire point of the FO transform – it must do so in a way that gives an attacker *no* usable signal from submitting malformed ciphertexts. The mechanism is *re-encrypt and compare*: decrypt to a candidate message, deterministically re-run encapsulation from it, and check that the ciphertext that *would* have been produced matches the one received.

Algorithm 19 ML-KEM.Decaps (teaching form)

Input: Decapsulation key $dk = (dk_{\text{PKE}}, ek, h, z)$; ciphertext c
Output: Shared secret K

$m' \leftarrow \text{K-PKE.}\textsc{Decrypt}(dk_{\text{PKE}}, c)$
$(K', r') \leftarrow G(m' \,\|\, h)$
$c' \leftarrow \text{K-PKE.}\textsc{Encrypt}(ek, m', r')$ ▷ re-encrypt from the recovered message
if $c' = c$ **then**
    return K' ▷ ciphertext was honestly formed
else
    return $J(z \,\|\, c)$ ▷ *implicit rejection*: a pseudorandom key from the secret z
end if

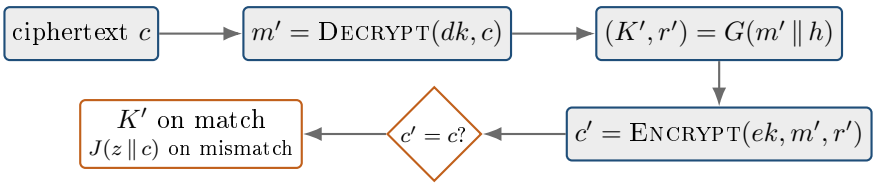


Figure 16.3: ML-KEM decapsulation and the Fujisaki–Okamoto transform. The candidate message is decrypted, encapsulation is re-run deterministically from it, and the recomputed ciphertext is compared with the received one in constant time. A match returns the real shared key; a mismatch returns a pseudorandom key derived from the secret z, so success and failure are indistinguishable.

The subtlety is the `else` branch. A naive KEM would return an *error* on a bad ciphertext – but that error is an oracle: by observing whether decapsulation succeeds, an attacker can mount a chosen-ciphertext attack and, over many queries, extract the secret. **Implicit rejection** closes this door: on a mismatch, ML-KEM returns a *pseudorandom* key $J(z \,\|\, c)$ derived from the secret z, of exactly the same form as a real key. Success and failure become indistinguishable to the caller, so there is no oracle. This is what lifts K-PKE from CPA to *IND-CCA2* security; it is why the comparison $c' = c$ must run in constant time (Chapter 26); and it is the same mechanism the code-based KEM HQC uses (Chapter 23).

16.14 Parameter sets

Scheme	k	η_1	(d_u, d_v)	NIST level	Comparable to
ML-KEM-512	2	3	$(10, 4)$	1	AES-128
ML-KEM-768	3	2	$(10, 4)$	3	AES-192
ML-KEM-1024	4	2	$(11, 5)$	5	AES-256

Table 16.1: The three ML-KEM parameter sets share $n = 256$ and $q = 3329$, but differ in their full parameter tuples. ML-KEM-768 is a common default.

16.15 SageMath experiments

The most useful way to study this chapter is through three short experiments.

Experiment 1: verify ExpandA Use the same seed twice:

```
A1 = expand_A(1234)
A2 = expand_A(1234)
```

Then verify that $A_1 = A_2$. Now change the seed and observe that the matrix changes as well.

Experiment 2: inspect CBD statistics Generate many samples from the CBD distribution and check that the average is close to zero.

```
from random import randint

def cbd(eta):
    return sum(randint(0, 1) for _ in range(eta)) - \
           sum(randint(0, 1) for _ in range(eta))

samples = [cbd(2) for _ in range(10000)]
print(sum(samples) / len(samples))
```

Observe how the variance changes when η changes.

Experiment 3: implement a teaching version of key generation Finally, write a function

```
pk, sk = keygen(seed)
```

and test the following properties:

- same seed gives the same key pair;
- different seeds give different key pairs.

This gives a first impression of how modern public-key cryptography is built from deterministic pseudo-random procedures.

Experiment 4: one MultiplyNTTs base product The NTT-domain multiplication in FIPS 203 is not 256 independent scalar products. Each coordinate pair is multiplied modulo a quadratic relation. The following check implements one such pair; it tests the algebraic base case, not the full standardized transform or its serialization.

```
q = 3329

def multiply_ntt_pair(a, b, gamma):
    a0, a1 = a
    b0, b1 = b
    return ((a0*b0 + gamma*a1*b1) % q,
            (a0*b1 + a1*b0) % q)

# (1 + 2X)(3 + 4X) modulo X^2 - gamma, with gamma = 17.
got = multiply_ntt_pair((1, 2), (3, 4), 17)
assert got == ((3 + 17*8) % q, (4 + 6) % q)
print("one ML-KEM quadratic base product:", got)
```

This is precisely the local multiplication used 128 times by MultiplyNTTs; a conforming ML-KEM implementation must additionally use FIPS 203's transform ordering, twiddle factors, reductions, encodings, and test vectors.

16.16 A complete SageMath implementation

The experiments above illustrate individual pieces. The companion file `sage/fips203_mlkem.sage` assembles all of them into a complete, byte-exact implementation of FIPS 203: every one of the standard's twenty-one numbered algorithms appears as its own function, named after the standard and annotated with its algorithm number, for all three parameter sets.

The algebraic objects are genuine Sage objects rather than opaque byte arrays. The coefficient domain is the quotient ring itself,

$$R_q = \mathbb{Z}_q[X]/(X^{256}+1),$$

and the NTT domain T_q is carried as a Sage vector over $\mathbb{F}_q$:

```
MLKEM_n, MLKEM_q, MLKEM_zeta = 256, 3329, 17

Fq = GF(MLKEM_q)
Rx = PolynomialRing(Fq, 'X')
X  = Rx.gen()
Rq = Rx.quotient(X^MLKEM_n + 1, 'x')   # coefficient domain
Tq = VectorSpace(Fq, MLKEM_n)          # NTT domain

ZETA  = [Fq(MLKEM_zeta)^BitRev7(i) for i in range(128)]
GAMMA = [Fq(MLKEM_zeta)^(2*BitRev7(i) + 1) for i in range(128)]
```

With that in place, the transform is the standard's butterfly network written directly over $\mathbb{F}_q$:

```
def NTT(f):
    """Algorithm 9. Map f in Rq to its NTT representation in Tq."""
    fh = _coeffs(f)
    i, length = 1, 128
    while length >= 2:
        for start in range(0, MLKEM_n, 2*length):
            z = ZETA[i]; i += 1
            for j in range(start, start + length):
```

```
                t = z * fh[j + length]
                fh[j + length] = fh[j] - t
                fh[j]          = fh[j] + t
        length //= 2
    return vector(Fq, fh)
```

Key generation is then the seven phases of this chapter, line for line:

```
def KPKE_KeyGen(self, d):
    """Algorithm 13. K-PKE.KeyGen(d) -> (ek_PKE, dk_PKE)."""
    k = self.k
    rho, sigma = _G(bytes(d) + bytes([k]))   # domain separator k
    N = 0
    Ah = self._ExpandA(rho)
    s = []
    for _ in range(k):
        s.append(SamplePolyCBD(self.eta1, _PRF(self.eta1, sigma, N))); N += 1
    e = []
    for _ in range(k):
        e.append(SamplePolyCBD(self.eta1, _PRF(self.eta1, sigma, N))); N += 1
    sh = [NTT(si) for si in s]
    eh = [NTT(ei) for ei in e]
    th = [ntt_vec_dot(Ah[i], sh) + eh[i] for i in range(k)]
    ek = b"".join(ByteEncode(12, _ints(t)) for t in th) + rho
    dk = b"".join(ByteEncode(12, _ints(si)) for si in sh)
    return ek, dk
```

Note the byte `k` appended to the seed in the very first line: FIPS 203 domain-separates the three parameter sets so that a seed expanded under the wrong one yields an unrelated key.

Why a computer-algebra system earns its place here. Because R_q is a real Sage ring, the hand-rolled transform can be checked against the algebra it claims to implement rather than merely against itself. The file's `verify_ntt_is_a_ring_isomorphism()` confirms three things using Sage's own arithmetic as the oracle: that NTT^{-1} inverts NTT; that the i-th output coefficient pair really is $f \bmod (X^2 - \gamma_i)$, so the transform is the Chinese-remainder map onto the 128 quadratic factors of $X^{256}+1$; and that

$$\mathrm{NTT}(f \cdot g) \;=\; \textsc{MultiplyNTTs}\big(\mathrm{NTT}(f), \mathrm{NTT}(g)\big),$$

where the left-hand product is computed by Sage in R_q. That last identity is precisely the statement that the NTT is a ring isomorphism $R_q \to T_q$ – the fact Experiment 4 could only gesture at with a single base product.

Validation. Correctness is not argued, it is measured. The test runner `sage/test_kat.sage` checks the implementation against NIST's ACVP test vectors [5], which carry both the inputs and the expected outputs of every case: key generation, encapsulation, decapsulation, and the encapsulation- and decapsulation-key validity checks of Sections 7.2 and 7.3. All outputs match byte for byte across ML-KEM-512, 768, and 1024. From the book's `latex/` directory, `make kat` runs it.

16.17 Pitfalls

Pitfall 1: thinking that the matrix A is stored explicitly This is not true in the standard. The matrix is derived from ρ and therefore regenerated when needed.

Pitfall 2: thinking that CBD is truly random sampling CBD is not random sampling in the naive sense. It is a deterministic sampling procedure driven by a seed and an expansion function.

Pitfall 3: thinking that KeyGen calls the random generator at every step In the real algorithm, the initial seed d is the only place where fresh randomness is introduced. Everything after that is a deterministic transform.

Pitfall 4: returning an error on decryption failure An implementation that signals "bad ciphertext" instead of returning the implicit-rejection key $J(z \,\|\, c)$ reintroduces a decryption-failure oracle and breaks CCA security. Both the comparison $c' = c$ and the choice between K' and $J(z \,\|\, c)$ must be constant-time.

16.18 Summary

ML-KEM is a two-layer construction, and this chapter built it end to end:

- the inner **K-PKE** is a CPA-secure public-key encryption whose entire key generation is a deterministic expansion of one seed,

$$d \longrightarrow (\rho, \sigma) \longrightarrow (A, s, e) \longrightarrow t \longrightarrow (\rho, t),$$

  publishing only (ρ, t) and never A;

- **encapsulation** hides a *random* message m, deriving both the shared key K and the encryption randomness r by hashing m, so the ciphertext is a deterministic function of m;

- **decapsulation** decrypts, re-encrypts, and compares – returning the real key on a match and a pseudorandom key $J(z \,\|\, c)$ on a mismatch. This is the **Fujisaki–Okamoto** transform with implicit rejection, and it is what lifts CPA security to *IND-CCA2*.

That completes the first NIST post-quantum standard, FIPS 203, from a single random seed to a CCA-secure shared key. The next part turns from key establishment to the other half of public-key cryptography – digital signatures – beginning with the lattice-based standards ML-DSA and FN-DSA, which reuse much of the machinery (module lattices, CBD sampling, the NTT) built here.

Signatures: ML-DSA and FN-DSA

Chapter 17

Complete Implementation of ML-DSA (FIPS 204)

17.1 Learning objectives

The previous part of the book built up to FIPS 203, the real ML-KEM standard for *encryption and key exchange*. We now cross to the other half of public-key cryptography, *digital signatures*, and to the first of three signature standards: FIPS 204, **ML-DSA**, better known by its submission name Dilithium [2, 31].

After finishing this chapter, you should be able to:

1. explain what a signature scheme does and how it differs from a KEM;
2. follow the ML-DSA key generation flow and see that it reuses the module-lattice machinery of Kyber almost verbatim;
3. explain the two hard problems, Module-LWE and Module-SIS, and which part of the scheme each one protects;
4. follow the signing procedure phase by phase, and understand why it contains a rejection ("abort") loop;
5. connect the rejection idea back to the role of noise studied in the LWE chapters.

17.2 Review: how signing differs from ML-KEM

ML-KEM answered the question *how do two parties agree on a shared secret?* A signature answers a different question: *how does a verifier know that a message really came from the holder of a secret key, unchanged?* The three algorithms mirror a KEM but serve this new goal:

- **KeyGen** $\rightarrow$ a public verification key pk and a secret signing key sk;
- **Sign**$(sk, M) \rightarrow$ a signature σ on a message M;
- **Verify**$(pk, M, \sigma) \rightarrow$ *accept* or *reject*.

The security goal is *unforgeability*: even after seeing many valid signatures, nobody without sk can produce a valid signature on a new message. The remarkable fact is that ML-DSA achieves this on *exactly* the same ring $R_q = \mathbb{Z}_q[x]/(x^{256}+1)$ and module structure we already built for Kyber. Only the modulus changes ($q = 8380417$), and the signing procedure is genuinely new.

17.3 The two hard problems

ML-DSA leans on two lattice problems in the module setting of Chapter 13.

- **Module-LWE**, already familiar: from $(A,\ \mathbf{t} = A\mathbf{s}_1 + \mathbf{s}_2)$ with short $\mathbf{s}_1, \mathbf{s}_2$, recovering the secret is hard. This hides the signing key inside the public key.
- **Module-SIS** (Short Integer Solution): the module-lattice form of the SIS problem introduced in Section 11.7. Given a random A, finding a short nonzero $\mathbf{z}$ with $A\mathbf{z} = \mathbf{0} \pmod q$ is hard. This blocks forgery: a forger would have to produce a short vector satisfying a public equation.

SIS is the mirror image of LWE. LWE hides a secret inside noise; SIS asks you to *find* a short solution to a random system. Both are hard for the same geometric reason – short vectors in a high-dimensional lattice are hard to find.

17.4 The real input in FIPS 204

Exactly as in FIPS 203, key generation does not begin by sampling a matrix. It begins with a single short seed:

$$\xi \in \{0,1\}^{256}.$$

Everything else – the public matrix, both secret vectors, and the internal PRF keys – is expanded deterministically from ξ. Key generation is therefore a deterministic function KeyGen(ξ), just like ML-KEM.

17.5 Phase 1: expand the seed

The seed is hashed by H (built on SHAKE-256) and split into three parts:

$$(\rho,\ \rho',\ K) = H(\xi).$$

Their roles are:

- ρ – public, expands into the matrix A (it will be stored in the public key);
- ρ' – secret, seeds the sampling of $\mathbf{s}_1$ and $\mathbf{s}_2$;
- K – secret, a PRF key used later to derive per-signature randomness.

17.6 Phase 2: generate the matrix

As in Kyber, the matrix is never stored; it is regenerated on demand from its seed:

$$A = \textsc{ExpandA}(\rho) \in R_q^{k\times\ell}.$$

The dimensions (k, ℓ) are what distinguish the three security levels.

17.7 Phase 3: sample the secret vectors

The two short secrets are expanded from ρ':

$$(\mathbf{s}_1,\ \mathbf{s}_2) = \textsc{ExpandS}(\rho'), \qquad \mathbf{s}_1 \in R_q^{\ell},\ \mathbf{s}_2 \in R_q^{k},$$

with coefficients in the small range $[-\eta, \eta]$. These play the role of the LWE secret and error.

17.8 Phase 4: compute and split the public value

The public value is the familiar Module-LWE equation:

$$\mathbf{t} = A\mathbf{s}_1 + \mathbf{s}_2.$$

To shrink the public key, $\mathbf{t}$ is split into a high part $\mathbf{t}_1$ (published) and a low part $\mathbf{t}_0$ (kept in the secret key):

$$\mathbf{t} = \mathbf{t}_1 \cdot 2^d + \mathbf{t}_0.$$

The public key is $(\rho, \mathbf{t}_1)$; the secret key bundles $(\rho, K, \mathbf{t}_0, \mathbf{s}_1, \mathbf{s}_2)$ together with a hash $tr = H(pk)$ used during signing.

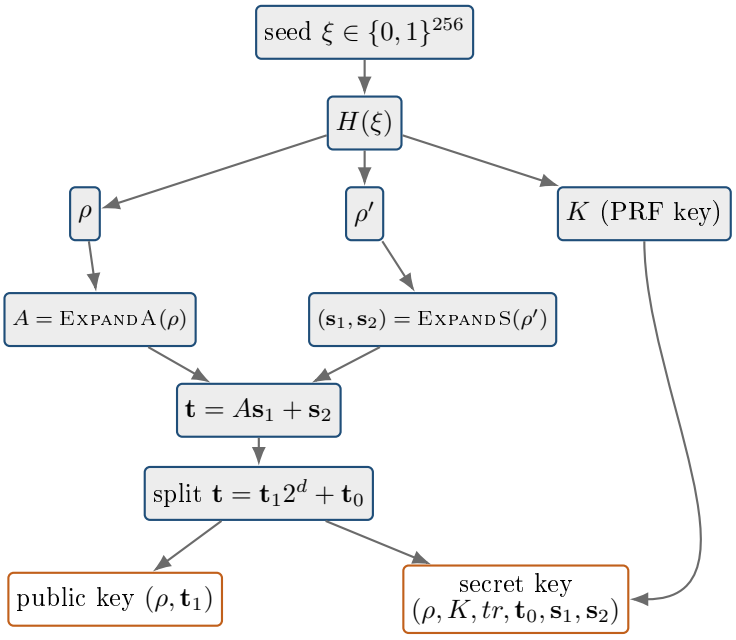


Figure 17.1: ML-DSA key generation, in the same deterministic-expansion style as FIPS 203. A single seed ξ fans out into the matrix, both secrets, and the PRF key; the public key stores only $(\rho, \mathbf{t}_1)$, from which A can always be regenerated.

Algorithm 20 ML-DSA.KeyGen (teaching form)

Input: Seed $\xi \in \{0,1\}^{256}$; parameters (k, ℓ, η, d)
Output: Public key $(\rho, \mathbf{t}_1)$; secret key $(\rho, K, tr, \mathbf{t}_0, \mathbf{s}_1, \mathbf{s}_2)$

1: $(\rho, \rho', K) \leftarrow H(\xi)$ ▷ Phase 1
2: $A \leftarrow \textsc{ExpandA}(\rho)$ ▷ Phase 2
3: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \textsc{ExpandS}(\rho')$ ▷ Phase 3
4: $\mathbf{t} \leftarrow A\mathbf{s}_1 + \mathbf{s}_2$ ▷ Phase 4
5: $(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \textsc{Power2Round}(\mathbf{t}, d)$
6: $tr \leftarrow H(\rho \,\|\, \mathbf{t}_1)$
7: **return** $\big((\rho, \mathbf{t}_1),\ (\rho, K, tr, \mathbf{t}_0, \mathbf{s}_1, \mathbf{s}_2)\big)$

17.9 Interlude: the Fiat–Shamir transform

Before diving into ML-DSA's signing, it is worth isolating the general technique it uses, because it is a basic building block that appears far beyond this one scheme.

An **identification scheme** lets a prover convince a verifier that it knows a secret s (whose public image is t), without revealing s. It is a three-move conversation: the prover sends a *commitment* w; the verifier replies with a random *challenge* c; the prover sends a *response* z that only someone knowing s could produce, and the verifier checks a relation among (w, c, z, t). If the response is properly randomized, the transcript reveals nothing about s.

The **Fiat–Shamir transform** [25] turns this interactive protocol into a signature by *removing the verifier*. Instead of waiting for a random challenge, the prover computes the challenge itself as a hash of the commitment together with the message:

$$c = H(w \,\|\, m).$$

Because H behaves like an unpredictable random function, the prover cannot steer c to its advantage; and because c depends on m, the whole transcript is bound to that message – it *is* a signature on m.

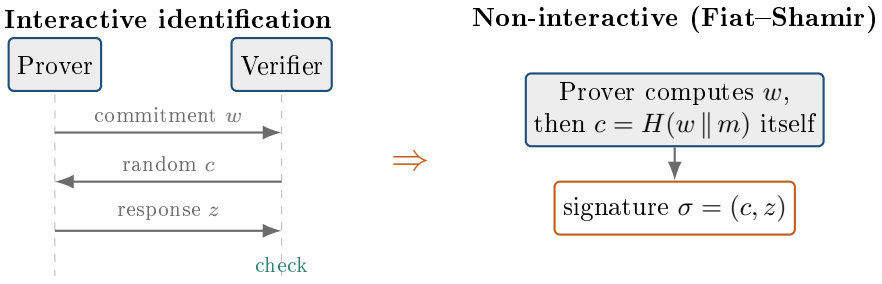


Figure 17.2: The Fiat–Shamir transform. On the left, an interactive identification protocol: three messages, with the verifier supplying a fresh random challenge c. On the right, the same transcript made non-interactive by having the prover compute $c = H(w \,\|\, m)$ itself – the hash plays the role of the verifier, and binding it to the message m turns the transcript into a signature. This recipe underlies many signature schemes; ML-DSA instantiates it with module lattices.

17.10 The signing idea: commitment, challenge, response

Signing instantiates exactly that identification transcript, with the module-lattice objects playing the roles of commitment, challenge, and response:

1. **Commit.** Sample a random short masking vector $\mathbf{y}$ and form $\mathbf{w} = A\mathbf{y}$.
2. **Challenge.** First form the message representative $\mu = H(tr \,\|\, M')$, where M' includes any context or pre-hash formatting. Then compute $\tilde{c} = H(\mu \,\|\, \mathbf{w}_1)$ with $\mathbf{w}_1 = \textsc{HighBits}(\mathbf{w})$, and derive $c = \textsc{SampleInBall}(\tilde{c})$.
3. **Respond.** Reply with $\mathbf{z} = \mathbf{y} + c\,\mathbf{s}_1$.

A verifier can then recompute $A\mathbf{z} - c\,\mathbf{t}$ and check it matches $\mathbf{w}$ in the high bits, because

$$A\mathbf{z} - c\,\mathbf{t} = A(\mathbf{y} + c\,\mathbf{s}_1) - c(A\mathbf{s}_1 + \mathbf{s}_2) = A\mathbf{y} - c\,\mathbf{s}_2 \approx \mathbf{w}.$$

The leftover $c\,\mathbf{s}_2$ is small, and the HighBits rounding absorbs it. Replacing the interactive verifier by the hash H is precisely the Fiat–Shamir transform: the hash acts as an unpredictable, unmanipulable challenger.

17.11 Why signing must sometimes abort

The response $\mathbf{z} = \mathbf{y} + c\,\mathbf{s}_1$ carries the secret $\mathbf{s}_1$ directly. If the mask $\mathbf{y}$ is not large enough to swamp $c\,\mathbf{s}_1$, then $\mathbf{z}$ leaks information – across many signatures the mask averages out and the secret emerges. This is the LWE lesson in a new costume: *the mask must completely hide the secret.*

ML-DSA's answer is **rejection sampling**, also called Fiat–Shamir *with aborts*. After forming $\mathbf{z}$, the signer checks whether it lands in a safe region where its distribution is independent of $\mathbf{s}_1$. If not, it discards everything, draws a fresh $\mathbf{y}$, and retries. Only a leak-free $\mathbf{z}$ is ever released.

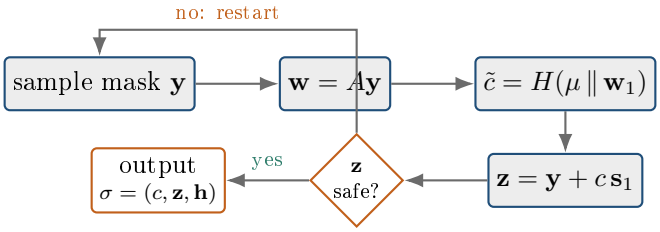


Figure 17.3: The ML-DSA signing loop. A fresh mask $\mathbf{y}$ is drawn each round; the response $\mathbf{z}$ is released only when the safety checks confirm its distribution is independent of $\mathbf{s}_1$. On average only a few iterations are needed. The hint $\mathbf{h}$ lets the verifier recover the high bits despite the discarded low part $\mathbf{t}_0$.

17.12 Signing and verification

Bringing the phases together gives the signing algorithm. Two bounds mark the safe region: $\|\mathbf{z}\|_\infty$ must stay below $\gamma_1 - \beta$, and the low bits of $\mathbf{w} - c\mathbf{s}_2$ must stay below $\gamma_2 - \beta$. A hint $\mathbf{h}$ compensates for the omitted $\mathbf{t}_0$.

Algorithm 21 ML-DSA.Sign (teaching form)

Input: Secret key $(\rho, K, tr, \mathbf{t}_0, \mathbf{s}_1, \mathbf{s}_2)$, message M, 32-byte value rnd
Output: Signature $\sigma = (c, \mathbf{z}, \mathbf{h})$

$A \leftarrow \textsc{ExpandA}(\rho)$; $\mu \leftarrow H(tr \,\|\, M')$
$\kappa \leftarrow 0$; $\rho'' \leftarrow H(K \,\|\, rnd \,\|\, \mu)$ ▷ $rnd = 0$ gives the deterministic option
loop
  $\mathbf{y} \leftarrow \textsc{ExpandMask}(\rho'', \kappa)$; $\kappa \leftarrow \kappa + \ell$ ▷ fresh mask each round
  $\mathbf{w} \leftarrow A\mathbf{y}$; $\mathbf{w}_1 \leftarrow \textsc{HighBits}(\mathbf{w})$
  $\tilde{c} \leftarrow H(\mu \,\|\, \mathbf{w}_1)$; $c \leftarrow \textsc{SampleInBall}(\tilde{c})$
  $\mathbf{z} \leftarrow \mathbf{y} + c\,\mathbf{s}_1$
  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **or** $\|\textsc{LowBits}(\mathbf{w} - c\mathbf{s}_2)\|_\infty \geq \gamma_2 - \beta$ **then**
    continue ▷ unsafe: discard and resample
  end if
  $\mathbf{h} \leftarrow \textsc{MakeHint}(c\mathbf{t}_0,\ \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0)$
  return $(c, \mathbf{z}, \mathbf{h})$
end loop

Verification recomputes the commitment from public data and checks the challenge hash matches, while also confirming $\mathbf{z}$ is genuinely short – the short-vector check is what confronts a forger with Module-SIS.

Algorithm 22 ML-DSA.Verify (teaching form)

Input: Public key $(\rho, \mathbf{t}_1)$, message M, signature $(c, \mathbf{z}, \mathbf{h})$
Output: *accept* or *reject*

$A \leftarrow \textsc{ExpandA}(\rho)$; $\mu \leftarrow H(H(\rho \,\|\, \mathbf{t}_1) \,\|\, M)$
$\mathbf{w}'_1 \leftarrow \textsc{UseHint}\big(\mathbf{h},\ A\mathbf{z} - c\,\mathbf{t}_1 \cdot 2^d\big)$
$c' \leftarrow H(\mu \,\|\, \mathbf{w}'_1)$
if $c' = c$ **and** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ **then**
  return *accept*
else
  return *reject*
end if

Because $A\mathbf{z} - c\,\mathbf{t} = A\mathbf{y} - c\mathbf{s}_2$ agrees with $\mathbf{w} = A\mathbf{y}$ in its high bits, and the hint repairs the gap left by publishing only $\mathbf{t}_1$, both sides hash to the same challenge exactly when the signature is genuine.

17.13 Why a forger is stuck

An attacker without *sk* may freely choose $\mathbf{z}$ and c, but to pass verification they must satisfy $c = H(\mu \,\|\, \textsc{HighBits}(A\mathbf{z} - c\mathbf{t}))$ *and* keep $\mathbf{z}$ short. Because H behaves like a random function, its output cannot be steered; the only route is to find a genuinely short $\mathbf{z}$ consistent with the equation – a Module-SIS instance. And they cannot instead learn $\mathbf{s}_1$ from public data, because that is Module-LWE. The two hard problems shut both doors at once.

17.14 SageMath experiment

The core cancellation is easy to verify in miniature. Using small parameters, sample a key, form a masked response, and confirm the verification identity holds before any signature is even released.

```
# Toy sizes; real ML-DSA uses n=256, q=8380417.
from random import Random
from sage.all import GF, Matrix, PolynomialRing, vector

q = 8380417
R.<x> = PolynomialRing(GF(q))
Rq = R.quotient(x^8 + 1)  # toy ring
rng = Random("mldsa-cancellation")

def small_poly():
    return Rq([rng.choice([-1, 0, 1]) for _ in range(8)])

def uniform_poly():
    return Rq([rng.randrange(q) for _ in range(8)])

A = Matrix(Rq, 2, 2, [uniform_poly() for _ in range(4)])
s1 = vector(Rq, [small_poly(), small_poly()])
s2 = vector(Rq, [small_poly(), small_poly()])
t = A * s1 + s2
y = vector(Rq, [uniform_poly(), uniform_poly()])
c = small_poly()
# For a mask y and challenge c, check the verifier's identity:
#   A*z - c*t  ==  A*y - c*s2   (should hold exactly)
z = y + c*s1
assert A*z - c*t == A*y - c*s2
```

Then repeat with several masks and count how often $\|\mathbf{z}\|_\infty$ exceeds the bound: this is exactly the rejection rate, and it shows why a signature takes a small, variable number of attempts.

17.15 A complete SageMath implementation

The cancellation check above is a fragment. The companion file `sage/fips204_mldsa.sage` is the whole standard: all forty-nine numbered algorithms of FIPS 204, each as its own function annotated with its algorithm number, for ML-DSA-44, 65, and 87. That includes the parts this chapter has treated as black boxes – ExpandA, ExpandS, ExpandMask, SampleInBall, Power2Round, Decompose, MakeHint, UseHint – as well as the bit packing, the hint encoding, the pre-hash variants HashML-DSA, and Montgomery reduction.

As in FIPS 203 the algebra is carried by Sage. Here, though, the standard is careful to distinguish the ring R_q from the ring R of integer polynomials with a restricted coefficient range, and so is the code: quantities reduced modulo q (the secrets $\mathbf{s}_1, \mathbf{s}_2$, the mask $\mathbf{y}$, the response $\mathbf{z}$) live in the Sage quotient ring $R_q = \mathbb{F}_{8380417}[X]/(X^{256}+1)$, while $\mathbf{t}_1$, $\mathbf{w}_1$, $\mathbf{r}_0$, and the hint $\mathbf{h}$ stay integer lists. The NTT domain T_q is the direct product $(\mathbb{Z}_q)^{256}$, so MultiplyNTT (Algorithm 45) is literally a coordinatewise product.

The rejection loop of Algorithm 21 then reads almost exactly as the standard writes it:

```
kappa = 0
while True:
    y  = self.ExpandMask(rhopp, kappa)
    yh = [NTT(Rq(p)) for p in y]
```

```
    w  = [NTTInverse(a) for a in MatrixVectorNTT(Ah, yh)]
    w1 = HighBitsVec(w, self.gamma2)

    ctilde = H(mu + self.w1Encode(w1), self.lam // 4)
    c      = self.SampleInBall(ctilde)
    ch     = NTT(Rq(c))

    cs1  = [NTTInverse(MultiplyNTT(ch, a)) for a in s1h]
    cs2  = [NTTInverse(MultiplyNTT(ch, a)) for a in s2h]
    z    = [Rq(y[i]) + cs1[i] for i in range(self.ell)]
    wcs2 = [w[i] - cs2[i] for i in range(self.k)]
    r0   = LowBitsVec(wcs2, self.gamma2)

    kappa += self.ell
    if inf_norm_vec(z) >= self.gamma1 - self.beta:      # unsafe: retry
        continue
    if inf_norm_ints(r0) >= self.gamma2 - self.beta:
        continue

    ct0 = [NTTInverse(MultiplyNTT(ch, a)) for a in t0h]
    h   = MakeHintVec([-p for p in ct0],
                      [wcs2[i] + ct0[i] for i in range(self.k)],
                      self.gamma2)
    if inf_norm_vec(ct0) >= self.gamma2:
        continue
    if sum(sum(p) for p in h) > self.omega:
        continue

    return self.sigEncode(ctilde, [centered(zi) for zi in z], h)
```

Every `continue` is one of the discarded attempts of Figure 17.3; counting them over many signatures reproduces the "repetitions" row of the parameter table, roughly four for ML-DSA-44. Note also that κ advances before the checks, so each retry draws a genuinely fresh mask.

Checking the transform against the algebra. The file's `verify_ntt_is_a_ring_isomorphism()` uses Sage to confirm that NTT^{-1} inverts NTT; that the j-th NTT coefficient is the evaluation $w(\zeta^{2\,\mathrm{BitRev8}(j)+1})$, which is what makes T_q a direct product; and that MultiplyNTT transports multiplication in R_q as Sage computes it. It also checks that MontgomeryReduce really returns $a \cdot 2^{-32} \bmod q$.

Validation. The implementation is checked against NIST's ACVP vectors [5] for FIPS 204 – key generation, signature generation, and signature verification – and reproduces them byte for byte. The signature-generation vectors are the demanding ones, because they cover every combination the standard permits: deterministic and hedged signing, the pure and pre-hash message encodings across all twelve approved pre-hash functions, the external and internal signing interfaces, and the external-μ mode in which the message representative is computed in a separate cryptographic module. Run `make kat` from the `latex/` directory.

17.16 Pitfalls

Pitfall 1: thinking the mask can be reused or made small. The mask $\mathbf{y}$ must be fresh every attempt and large enough to hide $c\mathbf{s}_1$. Reusing a mask, or shrinking it to avoid rejections, leaks the secret. This is the whole reason for the abort loop.

Pitfall 2: thinking the abort loop is a timing leak about the secret. The number of iterations depends on the mask, which is fresh randomness, not on the secret in an exploitable way. FIPS 204 derives the mask seed from $K \,\|\, rnd \,\|\, \mu$: choosing a zero rnd gives deterministic signing, while a fresh rnd supports randomized or hedged signing.

Pitfall 3: thinking A is stored in the key. As in ML-KEM, A is regenerated from ρ every time. The public key is only $(\rho, \mathbf{t}_1)$.

17.17 Parameter sets

Scheme	(k, ℓ)	NIST level	Comparable to
ML-DSA-44	$(4, 4)$	2	AES-128
ML-DSA-65	$(6, 5)$	3	AES-192
ML-DSA-87	$(8, 7)$	5	AES-256

Table 17.1: The three standardized ML-DSA parameter sets. All share $n = 256$ and $q = 8380417$.

17.18 Summary

ML-DSA reuses the module-lattice world of Kyber and adds one genuinely new ingredient, Fiat–Shamir with aborts:

- key generation is a deterministic expansion from one seed ξ, exactly like FIPS 203;
- a signature is a commitment–challenge–response transcript made non-interactive by hashing;
- the response $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ carries the secret, so it is released only when rejection sampling confirms it hides the secret completely;
- unforgeability rests on Module-LWE (hiding the key) and Module-SIS (blocking short-vector forgeries).

The next chapter stays with lattices but takes a very different route to a signature: FN-DSA (Falcon), which abandons Fiat–Shamir for a hash-and-sign construction over NTRU lattices and, in exchange, produces the most compact signatures of the three standards.

Chapter 18

FN-DSA (Falcon): Draft-Oriented Walkthrough

18.1 Learning objectives

This chapter is a draft-oriented walkthrough of FN-DSA, NIST's planned standardization of Falcon [4, 32]. It is not a claim of conformance to a final FIPS: readers must consult the exact draft version and date used by an implementation. Unlike ML-KEM, ML-DSA, and SLH-DSA, FIPS 206 remains under standardization at this book's standards snapshot. Where ML-DSA uses Fiat–Shamir with aborts, Falcon takes a different route on the same lattice foundation: *hash-and-sign* over NTRU lattices, using a Gaussian trapdoor sampler. The payoff is remarkably compact signatures and keys; the price is the most delicate implementation of the three.

After finishing this chapter, you should be able to:

1. contrast hash-and-sign with the Fiat–Shamir approach of ML-DSA;
2. describe the NTRU lattice and why a short secret basis is a trapdoor;
3. explain how signing is a nearest-lattice-point problem solved with a Gaussian sampler;
4. connect the sampler back to discrete Gaussians, good vs. bad bases, and CVP from Part II;
5. understand why Falcon's floating-point arithmetic makes constant-time implementation hard.

18.2 Review: hash-and-sign versus Fiat–Shamir

ML-DSA built a signature from an interactive identification protocol collapsed by a hash. Falcon uses the older, more direct *hash-and-sign* paradigm, the same shape as classical RSA signatures:

1. hash the message to a target point c;
2. use the secret trapdoor to produce a short vector that "explains" c;
3. the verifier checks the vector is short and consistent with c.

The difficulty that stalled lattice hash-and-sign for years was that a naive short vector *leaks the secret basis*: every signature is a hint about the trapdoor. Falcon's core contribution is a sampler

whose output distribution is provably independent of the secret – a discrete Gaussian over the lattice, in the hash-and-sign framework of Gentry, Peikert, and Vaikuntanathan [24]. This is the same "the mask must hide the secret" principle as ML-DSA's aborts, achieved by a different mechanism.

18.3 The NTRU lattice and its trapdoor

Falcon rests on the NTRU assumption introduced in Section 12.15. It works in the by-now-familiar ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ with $n \in \{512, 1024\}$ and $q = 12289$. Key generation picks two *short* secret polynomials $f, g \in R_q$ and publishes

$$h = g \cdot f^{-1} \pmod{q}.$$

The public key is just the single polynomial h. It defines the **NTRU lattice**

$$\Lambda = \{(u, v) \in R_q^2 : u + v\,h = 0 \pmod{q}\},$$

a $2n$-dimensional lattice exactly of the kind studied in Part II. From h alone you can write down a basis for Λ, but it is a *bad* basis – long and skewed – so finding short vectors in it is hard.

The secret is a *good* basis for the very same lattice. Key generation completes (f, g) into a full short basis by solving the **NTRU equation**

$$f\,G - g\,F = q$$

for two more short polynomials F, G. The pair of rows $(g, -f)$ and $(G, -F)$ is a short, nearly orthogonal basis of Λ – precisely the "good basis" that Chapters 7–9 told us is the key to solving lattice problems.

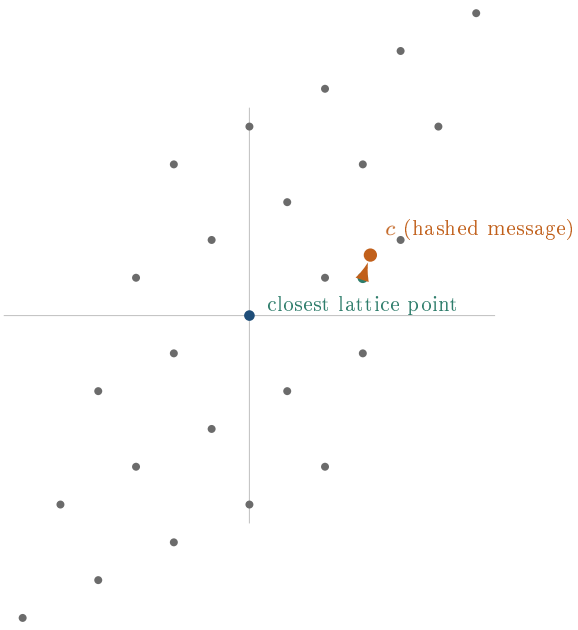


Figure 18.1: Signing as a closest-vector problem. Hashing the message gives a target point c (orange). The signer uses the secret *good* basis of the NTRU lattice to find a lattice point very close to c; the short difference vector is the signature. With only the public *bad* basis (from h) this closest point cannot be found – that gap is the trapdoor.

18.4 Phase 1: key generation

Algorithm 23 FN-DSA.KeyGen (teaching form)

Input: Ring R_q, $q = 12289$, dimension n
Output: Public key h; secret key (short basis) (f, g, F, G)

1: **repeat**
2: sample short $f, g \in R_q$ (small Gaussian coefficients)
3: **until** f is invertible mod q **and** (f, g) is short enough
4: solve the NTRU equation $fG - gF = q$ for short F, G
5: $h \leftarrow g \cdot f^{-1} \bmod q$
6: **return** public h, secret (f, g, F, G)

The secret basis and the public h describe the *same* lattice; they differ only in quality. That is the entire trapdoor.

18.5 Phase 2: hash the message to a point

Signing begins by mapping the message to a random-looking target in the ring. A fresh salt r is prepended so that signing the same message twice gives independent targets:

$$c = \textsc{HashToPoint}(r \,\|\, M) \in R_q.$$

18.6 Phase 3: sample a short solution (the trapdoor sampler)

The signer must find a short pair (s_1, s_2) satisfying

$$s_1 + s_2\, h = c \pmod q,$$

which is the same as finding a lattice point of Λ close to $(c, 0)$. Using the good basis, the signer runs **fast Fourier sampling**: a discrete Gaussian sampler over the lattice coset, implemented efficiently over the FFT representation of the basis (organized as an "LDL tree"). Two properties matter:

- the result is *short*, because the good basis makes nearby lattice points reachable;
- its distribution is a *discrete Gaussian* centered at the target, which – crucially – is independent of *which* good basis produced it. This is what stops each signature from leaking the secret.

This is where the whole book converges: the discrete Gaussian of Chapter 10, the good-versus-bad-basis geometry of Chapters 7–8, and the closest-vector view of the LWE chapter all appear at once, in service of one signature.

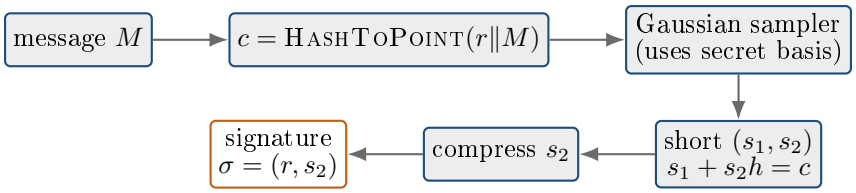


Figure 18.2: The Falcon signing flow. The message is hashed to a target c; the secret basis drives a Gaussian sampler that returns a short solution of $s_1 + s_2 h = c$; only the salt r and the compressed second component s_2 are output, since the verifier can recover s_1.

18.7 Phase 4: compress and output

Only s_2 needs to be sent: the verifier can recover $s_1 = c - s_2 h \bmod q$. Because s_2 has small Gaussian coefficients, it compresses well, giving Falcon its signature-size advantage. The signature is $\sigma = (r, \text{compress}(s_2))$.

Algorithm 24 FN-DSA.Sign / Verify (teaching form)

Sign$(sk = (f, g, F, G),\ M)$:
1: sample salt r; $c \leftarrow \text{HashToPoint}(r \| M)$
2: $(s_1, s_2) \leftarrow$ FFSampling of a short solution to $s_1 + s_2 h = c$ using the secret basis
3: **return** $\sigma = (r,\ \text{Compress}(s_2))$

Verify$(pk = h,\ M,\ \sigma = (r, \hat{s}_2))$:
4: $s_2 \leftarrow \text{Decompress}(\hat{s}_2)$; $c \leftarrow \text{HashToPoint}(r \| M)$
5: $s_1 \leftarrow c - s_2\, h \bmod q$ ▷ recover the first component
6: **return** *accept* if $\|(s_1, s_2)\|^2 \le \beta^2$, else *reject* ▷ must be short

Verification is cheap: one multiplication to recover s_1, then a norm check. A forger without the good basis would have to solve the closest-vector problem in the NTRU lattice using only h – exactly the hardness Part II established.

18.8 Why Falcon is the hardest to implement

The Gaussian sampler needs high-precision *floating-point* arithmetic (FFT over the reals) to get the output distribution exactly right. Floating-point operations are notoriously variable-time and platform-dependent, which makes a *constant-time* implementation – one that leaks nothing through timing – genuinely difficult. If the sampler's timing or rounding depends on the secret basis, an attacker can gradually reconstruct it. This is the direct motivation for the final chapter on side-channel security, and it is the main reason Falcon, despite its elegance and compactness, is the most demanding of the three standards to deploy safely.

18.9 SageMath experiment: the public NTRU relation

Sage can verify the public-key equation underlying the NTRU lattice. The example deliberately stops before the Gaussian trapdoor sampler: finding a correctly distributed short signature is the hard, specification-sensitive part of FN-DSA and is not reproduced by this toy calculation.

```
from random import Random
from sage.all import GF, PolynomialRing

q = 17
n = 8
rng = Random("fndsa-ntru-relation")
R.<x> = PolynomialRing(GF(q))
Rq = R.quotient(x^n + 1)

def small_poly():
    return Rq([rng.choice([-1, 0, 1]) for _ in range(n)])

def small_unit():
    f = small_poly()
    while not f.is_unit():
        f = small_poly()
    return f

f = small_unit()
g = small_poly()
h = g * f^(-1)                  # public NTRU value
assert f * h == g                # f h = g in Rq

c = Rq([rng.randrange(q) for _ in range(n)])
s2 = small_poly()
s1 = c - s2*h                   # verifier's reconstruction rule
assert s1 + s2*h == c
print("NTRU public relation and verification equation hold")
```

The last equality is only an algebra check. In a real signature, FFSampling must choose (s_1, s_2) with the required short Gaussian distribution; choosing an arbitrary small s_2 as above is not an FN-DSA signing algorithm.

18.10 A complete SageMath implementation

The experiment above stops exactly where the difficulty starts. The companion file `sage/fips206_fndsa.sage` goes the rest of the way: all eighteen numbered algorithms of the Falcon specification, at $n = 512$ and $n = 1024$ – the FFT over the complex tower, NTRUGen with the tower-recursive NTRU solver, the LDL* decomposition and the Falcon tree, fast Fourier sampling with the isochronous discrete Gaussian sampler, signing, verification, and the compressed signature and key encodings.

Which specification the code follows. This matters more here than in the other three chapters. At the book's standards snapshot NIST lists FIPS 206 as *in development*: no initial public draft has been published, and consequently no ACVP test vectors exist. The authoritative specification for FN-DSA is therefore still the round-3 Falcon submission, and that is what the file implements, using Falcon's algorithm numbering. The places most likely to be respecified when the draft appears – the key and signature encodings, and the domain separation of the hashed message – are marked in the source. The lattice machinery below them is not expected to change.

Solving the NTRU equation. Key generation's one genuinely hard step is the line that Algorithm 23 states in five words: *solve $fG - gF = q$ for short F, G*. The method is a tower recursion. The field norm N maps $\mathbb{Z}[x]/(x^n + 1)$ down to $\mathbb{Z}[x]/(x^{n/2} + 1)$; applying it repeatedly reaches $\mathbb{Z}$,

where the equation is an extended gcd; the multiplicativity of the norm then lifts the solution back up, one level at a time.

```
def NTRUSolve(f, g):
    """Algorithm 6. Find F, G with f*G - g*F = q in Z[x]/(x^n + 1)."""
    n = len(f)
    if n == 1:
        d, u, v = xgcd(ZZ(int(f[0])), ZZ(int(g[0])))
        if d != 1:
            return None, None
        return [int(-FALCON_q*v)], [int(FALCON_q*u)]
    fp, gp = field_norm(f), field_norm(g)
    Fp, Gp = NTRUSolve(fp, gp)              # recurse in the smaller ring
    if Fp is None:
        return None, None
    F = _cyclo_mul(lift(Fp), galois_conjugate(g), n)
    G = _cyclo_mul(lift(Gp), galois_conjugate(f), n)
    return Reduce(f, g, F, G)               # shorten F, G against f, g
```

Here `field_norm` is $N(a) = a(x)a(-x)$ written in x^2, `lift` is $a(x) \mapsto a(x^2)$, and `galois_conjugate` is $a(x) \mapsto a(-x)$. The lifted F, G come back with enormous coefficients, which is what REDUCE is for: it repeatedly subtracts the multiple of (f, g) nearest to (F, G), rescaling to 53 significant bits each round so that the rounding stays exact enough to converge.

Three kinds of arithmetic, three Sage rings. Falcon is unusual in needing exact and inexact arithmetic side by side, and the implementation keeps them in separate, explicitly named Sage domains:

- the NTRU solver runs over `ZZ['x']`, so the trapdoor equation $fG - gF = q$ is established *exactly* rather than up to floating point – `verify_ntru_equation()` checks it as an identity in $\mathbb{Z}[x]/(x^n + 1)$;
- the public key and the verification relation $s_1 = c - s_2 h$ are computed in $\mathbb{F}_{12289}[x]/(x^n + 1)$;
- the signing FFT runs over `CDF`, Sage's complex double field, which *is* the IEEE 754 binary64 arithmetic that Falcon prescribes – not a higher-precision stand-in that would silently change the sampler's output distribution.

The function `verify_fft_against_exact_ring()` closes the loop between the first and third of these, checking that pointwise multiplication in the FFT domain reproduces the product Sage computes exactly in $\mathbb{Z}[x]/(x^n + 1)$.

Validation without NIST vectors. Since FIPS 206 has none, FN-DSA is validated against the Falcon submission's own known-answer tests [32], in four independent ways:

1. SAMPLERZ reproduces all sixteen test vectors of the Falcon specification's sampler table, byte for byte – including the cases where rejection sampling iterates twice or BEREXP consumes a second random byte;
2. for each Falcon KAT key, the private key is decoded, $fG - gF = q$ is verified exactly, and the recomputed $h = gf^{-1}$ re-encodes to the KAT public key byte for byte;
3. every Falcon KAT signature verifies, and the same signature over a tampered message is rejected;

4. signatures produced by this implementation under a Falcon KAT *private* key verify under the corresponding KAT public key.

Signing itself cannot be pinned to the KAT byte string, and the reason is instructive: Falcon signing is randomized, and the reference KATs were generated with the reference implementation's own PRNG chain, so identical signatures would test PRNG replication rather than the scheme. Verification is deterministic, and it is checked byte-exactly. The gap is precisely the Gaussian sampler discussed in the next section – the part that is hardest to specify, hardest to implement, and hardest to test.

18.11 Parameter sets

Scheme	n	NIST level	Public key	Signature
FN-DSA-512	512	1	≈ 897 B	≈ 666 B
FN-DSA-1024	1024	5	≈ 1793 B	≈ 1280 B

Table 18.1: The two FN-DSA (Falcon) parameter sets, both with $q = 12289$. Signatures and keys are markedly smaller than ML-DSA's or SLH-DSA's – the reason Falcon exists.

18.12 Pitfalls

Pitfall 1: sampling short vectors deterministically. If the signer always returned the *single* closest lattice point (e.g. via Babai rounding), every signature would trace the shape of the secret basis, and a few signatures would reveal it. The Gaussian *randomization* of the output is essential, not a mere convenience.

Pitfall 2: treating the floating-point sampler as ordinary code. The sampler's timing, branches, and rounding must not depend on secret data. Naive floating-point code leaks; Falcon implementations go to great lengths (emulated fixed-precision, careful table lookups) to be constant-time.

Pitfall 3: reusing the salt. The per-signature salt r must be fresh. Reusing it across different messages correlates their targets c and erodes the Gaussian's hiding property.

18.13 Summary

FN-DSA (Falcon) is lattice hash-and-sign done right:

- the public key h and the secret short basis (f, g, F, G) describe the same NTRU lattice, differing only in quality – the trapdoor;
- signing hashes the message to a target and uses a Gaussian sampler over the good basis to find a short, secret-hiding solution;
- verification recovers $s_1 = c - s_2 h$ and checks shortness, a closest-vector problem for anyone without the trapdoor;

- the reward is very compact signatures; the cost is delicate floating-point, constant-time engineering.

This completes the *lattice-based* signatures. ML-DSA and FN-DSA both live on lattices but reach a signature by opposite routes – Fiat–Shamir with aborts versus Gaussian hash-and-sign – and trade off size against implementation delicacy accordingly. The next part leaves lattices entirely for signatures built on a different hard problem: hash functions alone, culminating in the third signature standard, SLH-DSA (FIPS 205).

Part III

Hash-Based Cryptography

Chapter 19

Hash-Based Signatures: One-Time and Few-Time

19.1 Why this chapter matters

The previous part built signatures on lattice problems. This part builds them on a completely different foundation – nothing but a cryptographic hash function. There is no number theory, no lattice, no algebraic structure for an attacker to exploit: if the hash is secure, so is the signature. That makes hash-based signatures the most conservative post-quantum option, and it is why NIST standardized one (SLH-DSA, FIPS 205) as a hedge against a future structural break in lattices.

Because the foundation is so different, we introduce it the same way we introduced lattices: from the ground up. This chapter builds the atomic pieces – one-time and few-time signatures – and the next assembles them into a many-time, stateless scheme with Merkle trees. Only then, in Chapter 21, do we turn to the FIPS 205 standard itself.

19.2 Learning goals

After reading this chapter, you should be able to:

- state the three security properties a hash function must provide, and why Grover only mildly weakens them;
- construct and verify a Lamport one-time signature, and explain why it is strictly one-time;
- explain how Winternitz (WOTS$^+$) trades hashing time for much shorter signatures;
- describe the few-time signature FORS and how its security bound changes with the signature count.

19.3 What a secure hash function guarantees

Earlier chapters used hash functions such as SHAKE only as *expanders* – deterministic ways to stretch a short seed into a matrix or noise (Chapters 15–16). Here a hash is the *sole source of security*, so we state precisely what is assumed of $H : \{0,1\}^* \to \{0,1\}^n$.

Definition 19.1 (Hash-function security)**.** A cryptographic hash H is assumed to be

- *preimage resistant* (one-way): given y, it is hard to find any x with $H(x) = y$;
- *second-preimage resistant*: given x, it is hard to find $x' \neq x$ with $H(x') = H(x)$;
- *collision resistant*: it is hard to find any pair $x \neq x'$ with $H(x) = H(x')$.

That is the whole assumption. Because there is no hidden algebraic structure, there is no Shor-style break; the only quantum help is Grover's search (Chapter 1), which cuts preimage resistance from n bits to about $n/2$. Hash-based schemes simply use a larger output n (16, 24, or 32 bytes) to keep a comfortable margin. The threat that shatters RSA leaves hashing merely needing bigger parameters.

19.4 The Lamport one-time signature

The simplest possible hash-based signature, due to Lamport [28], already contains the key idea: *reveal a hash preimage as proof.*

To sign an m-bit message, the signer prepares, for each bit position, *two* secret random values – one to reveal if that bit is 0, one if it is 1 – and publishes their hashes. Signing a message reveals exactly one secret per position: the one matching the bit. A verifier hashes each revealed secret and checks it against the correct public value.

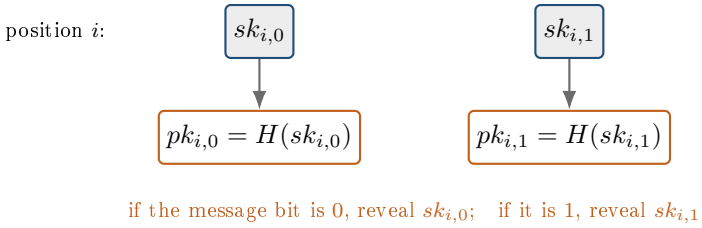


Figure 19.1: One position of a Lamport key. Each of the m message-bit positions has two secret values whose hashes are public. A signature reveals exactly one secret per position – the one matching the message bit – and the verifier confirms it hashes to the published value. The unrevealed secrets stay hidden by preimage resistance.

Algorithm 25 Lamport one-time signature

KeyGen:

1: for $i = 0, \ldots, m-1$ and $b \in \{0,1\}$, sample $sk_{i,b} \leftarrow \{0,1\}^n$ and set $pk_{i,b} \leftarrow H(sk_{i,b})$

2: **return** secret key $(sk_{i,b})$, public key $(pk_{i,b})$

Sign message $M = (M_0, \ldots, M_{m-1}) \in \{0,1\}^m$:

3: **return** $\sigma = (sk_{0,M_0},\ sk_{1,M_1},\ \ldots,\ sk_{m-1,M_{m-1}})$

Verify $(M, \sigma = (\sigma_0, \ldots, \sigma_{m-1}))$:

4: **return** *accept* iff $H(\sigma_i) = pk_{i,M_i}$ for every i

Remark (Why it is strictly one-time). A single signature reveals one secret per position – the ones matching M. To forge a *different* message M', an attacker would need sk_{i,M'_i} at each position where $M'_i \neq M_i$, and those secrets are still protected by preimage resistance. But two signatures on different messages jointly expose *both* secrets at every differing position, after which any message can be forged. Hence a Lamport key is safe for exactly one signature.

Lamport keys and signatures are large (each is $2m$ or m hash values). The next construction keeps the one-time security but shrinks the signature dramatically.

19.5 Winternitz: trading time for size (WOTS$^+$)

Instead of two independent secrets per bit, Winternitz uses one *hash chain* per base-w digit of the message. Starting from a secret sk, repeated hashing produces a chain, and the public value is its end:

$$sk \xrightarrow{H} H(sk) \xrightarrow{H} \cdots \xrightarrow{H} H^{w-1}(sk) = pk.$$

To sign the digit m, the signer reveals the intermediate point $H^m(sk)$; the verifier hashes it the remaining $w-1-m$ times and checks it reaches pk.

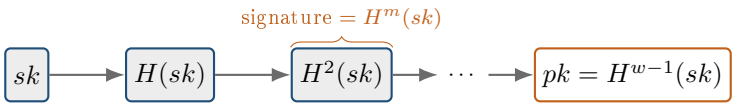


Figure 19.2: One WOTS$^+$ hash chain of length w. The secret is the start, the public value is the end, and a signature reveals an intermediate point. A verifier hashes forward to check it reaches the public end. A full message digest is split into base-w digits, each signed by its own chain.

There is a catch a checksum must fix. Revealing $H^m(sk)$ lets anyone hash *forward* to $H^{m'}(sk)$ for any $m' > m$, so an attacker could "advance" a digit to a larger value for free. WOTS$^+$ appends a short *checksum* of the digits, itself signed by extra chains, arranged so that increasing any message digit forces *decreasing* some checksum digit – which would require moving a chain *backward*, i.e. inverting H. The parameter w tunes a trade-off: larger w means fewer, longer chains (shorter signatures, more hashing), smaller w the reverse. Like Lamport, a WOTS$^+$ key is safe for exactly one signature.

19.6 FORS: a few-time signature

One-time keys are unforgiving: a single reuse is catastrophic. The stateless scheme of the next chapters uses **FORS** (Forest Of Random Subsets), a *few-time* signature, inside a very large address space. This does not authorize key reuse; it makes the security analysis degrade more gradually under the tiny probability of an accidental address collision.

Definition 19.2 (FORS)**.** FORS consists of k independent Merkle trees, each of height a (so each has 2^a leaves, and each leaf hides a secret value). A message digest is split into k indices, one per tree; the signature reveals, for each tree, the secret at the selected leaf together with its authentication path to that tree's root. The k roots are hashed together into a single FORS public key.

Because each tree reveals only the *one* leaf its index selects, the security bound degrades with the number of signatures and with any address collisions. SLH-DSA uses message randomization and a huge address space to make such collisions extremely unlikely. An implementation must never deliberately reuse a FORS or WOTS address; WOTS leaf reuse remains a serious event. (The Merkle trees FORS uses are the subject of the next chapter.)

19.7 Summary

This chapter built the atoms of hash-based signing:

- security rests only on hash preimage/collision resistance, which Grover weakens merely to $n/2$ bits;
- **Lamport** signs once by revealing one preimage per message bit – simple but large;
- **WOTS$^+$** replaces bit-pairs with hash chains plus a checksum, keeping one-time security at a fraction of the size;
- **FORS** is a *few-time* signature whose security bound degrades with the number of signatures; address reuse is never an implementation strategy.

Every one of these keys is still usable only once (or a few times). The next chapter removes that limitation: a Merkle tree binds many one-time keys under a single public root, and a hypertree of such trees, addressed by hashing the message, yields a stateless many-time signature.

Chapter 20

Merkle Trees and Stateless Signatures

20.1 Why this chapter matters

The previous chapter left us with a problem: every key we built – Lamport, WOTS$^+$, even FORS – is usable only once or a few times. A real signer must produce many signatures under *one* public key. This chapter supplies the two ideas that make that possible. A *Merkle tree* binds a large batch of one-time keys under a single public root, and a *hypertree* addressed by hashing the message turns the whole construction *stateless*, so the signer need not remember which keys it has already used. These are the last pieces before the FIPS 205 standard of Chapter 21.

20.2 Learning goals

After reading this chapter, you should be able to:

- build a Merkle tree over many one-time public keys and verify an authentication path;
- explain what makes a many-time hash-based scheme (XMSS) *stateful* and why that is dangerous;
- describe the hypertree and how it enlarges the number of usable keys;
- explain how hashing the message to select a leaf removes the state entirely.

20.3 The Merkle tree

A single one-time key is nearly useless on its own. The Merkle tree [29] fixes this: generate many one-time (WOTS$^+$) key pairs, hash each public key into a *leaf*, and build a binary tree in which every internal node is the hash of its two children. The single value at the top – the *root* – is the public key for *all* the leaves at once.

To use leaf i, the signer publishes the leaf's one-time signature *plus* the *authentication path*: the sibling hash at each level from the leaf up to the root. The verifier recomputes the leaf, folds it up the path, and accepts only if the final value equals the known root.

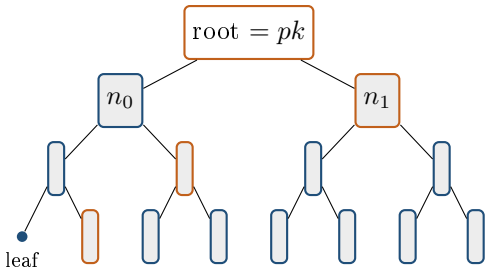


Figure 20.1: A Merkle tree of height 3. Each leaf is the hash of one WOTS$^+$ public key; internal nodes hash their two children; the single root is the public key. To authenticate the marked leaf, the signature carries the three orange sibling nodes (the authentication path). The verifier recomputes the leaf and hashes up the path, accepting only if it lands on the known root.

Algorithm 26 Verify a Merkle authentication path

Input: Leaf value ℓ, leaf index i, authentication path $(a_0, \ldots, a_{h-1})$, root r
Output: *accept* or *reject*

$node \leftarrow \ell$
for $j = 0, \ldots, h-1$ **do**
  if bit j of i is 0 **then**
    $node \leftarrow H(node \,\|\, a_j)$ ▷ current node is the left child
  else
    $node \leftarrow H(a_j \,\|\, node)$ ▷ current node is the right child
  end if
end for
return *accept* if $node = r$, else *reject*

A tree of height h turns 2^h one-time keys into a single public key of one hash value, at the cost of an h-node authentication path in every signature.

20.4 XMSS and the burden of state

A Merkle tree of WOTS$^+$ keys is essentially the scheme **XMSS**. It signs 2^h messages under one root – but only if each leaf is used *at most once*. That is a *stateful* requirement: the signer must remember which leaves are spent and never repeat one.

Remark (Why state is dangerous). State makes a many-time hash signature fragile in practice. If a key is backed up and restored, or run on two machines, or a virtual machine is rolled back, the same leaf index can be used twice – and reusing a one-time key breaks it completely. Stateful hash-based schemes (XMSS, LMS) are standardized and useful, but their safety depends on flawless state management, which is exactly what fails in real deployments.

20.5 The hypertree

Two problems remain: a single tree tall enough to never repeat a leaf would be astronomically expensive to build, and we still want to remove the state. The hypertree addresses the first. It

stacks d layers of Merkle trees so that each tree's root is signed by a one-time key in the tree one layer up; only the very top root is published.

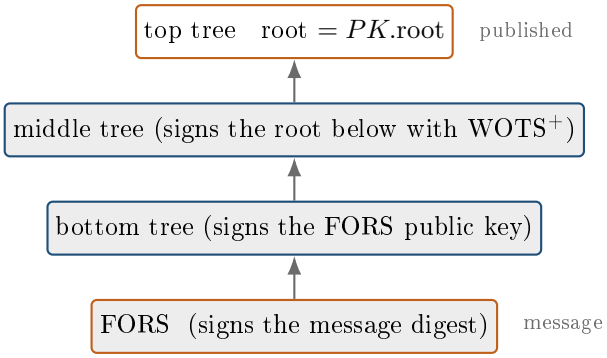


Figure 20.2: A hypertree with $d = 3$ layers. FORS signs the message digest; the bottom Merkle tree signs the FORS public key; each higher layer signs the root of the tree below it; only the top root is the public key. A single "signature" is this whole chain of one-time signatures and authentication paths from the message up to the published root. Chaining d trees of height h yields 2^{dh} effective leaves while only ever building trees of height h.

Chaining d trees of height h multiplies the exponents: the hypertree behaves like one tree of height dh, giving 2^{dh} usable bottom-layer keys, while never building a tree taller than h. With dh around sixty, the number of keys is so vast that collisions become negligible – which is what finally lets us drop the state.

20.6 From stateful to stateless

The last idea is how the leaf is chosen. Instead of a counter that must be remembered, the signer *hashes the message* (together with a fresh random value R) to pseudorandomly select which bottom-layer tree and leaf to use, and which FORS leaves to reveal:

$$(R,\ \text{tree index},\ \text{leaf index},\ \text{FORS indices}) \longleftarrow H(R \,\|\, PK \,\|\, M).$$

Two different messages almost never land on the same addresses because the hypertree is enormous and the message randomizer is included in the selection hash. This is a probability statement, not permission to reuse keys: implementations must never deliberately reuse a WOTS or FORS address. FORS security degrades with the number of signatures and any accidental collisions, while WOTS leaf reuse remains serious. No index is ever recorded, so there is no state to corrupt.

Remark (The role of the public seed). One subtlety carries over to the standard. Every hash in the tree is tweaked by a public seed and by an *address* identifying the node's position, so that structurally identical nodes in different trees hash differently. Without this, an attacker could attack many positions at once (a multi-target attack). The seed is public but essential, and we return to it in the next chapter.

20.7 Summary

This chapter completed the ladder from one-time keys to a practical many-time signature:

- a **Merkle tree** binds 2^h one-time keys under a single root, authenticated by an h-node path;

- using each leaf once gives a *stateful* scheme (XMSS), whose safety is fragile in deployment;
- a **hypertree** of d layers multiplies the key supply to 2^{dh} without building tall trees;
- **hashing the message** and message randomization select the path without signer state, while the enormous address space makes accidental collisions negligible; intentional address reuse is forbidden.

Everything is now in place. The next chapter assembles these pieces exactly as the FIPS 205 standard specifies them, deriving all secrets deterministically from a handful of seeds and addresses.

Chapter 21

SLH-DSA (SPHINCS$^+$, FIPS 205): Algorithm Walkthrough

21.1 What is SPHINCS$^+$?

SPHINCS$^+$ is the hash-based signature scheme that NIST standardized as **SLH-DSA** in FIPS 205 [3, 33], and it is the most conservative of all the post-quantum standards. Where the lattice signatures of the previous part rest on the presumed hardness of lattice problems, SPHINCS$^+$ rests on *nothing but the security of a hash function* – the single best-understood assumption in cryptography. If every lattice and code assumption fell tomorrow, a secure hash function would still yield a secure signature; that is exactly why the family is kept in the standard portfolio as a hedge.

The scheme is the endpoint of a short lineage: the stateless design *SPHINCS* appeared in 2015, was hardened into *SPHINCS$^+$* for the NIST competition through 2017–2019 [33], and was finalized as *SLH-DSA* in FIPS 205 in 2024. The three names denote essentially the same construction; this book uses SLH-DSA for the standardized algorithm and SPHINCS$^+$ for the scheme it came from. In one sentence, it is *a giant hypertree of one-time and few-time hash-based signatures, arranged so that no signing state need ever be kept.*

Why "stateless" is the hard part. The "SL" in SLH-DSA stands for *stateless*, and that is the property the design fights for. Earlier hash-based schemes such as XMSS and LMS are *stateful*: each one-time key may be used exactly once, so the signer must remember which leaves are already spent, and a single slip – a restored virtual-machine snapshot, a copied key file – reuses a one-time key and can be catastrophic. SPHINCS$^+$ removes the state entirely. It uses message randomization and a hypertree address space so large that accidental address collisions are negligible. This is not permission to reuse addresses: WOTS leaves must remain unique, while FORS security degrades with the finite signature count and any accidental collisions. The price of statelessness is size: SPHINCS$^+$ signatures are the largest of the standards, several kilobytes each. The scheme is instantiated over either **SHA-2** or **SHAKE**, in a *simple* or *robust* mode, and in the size-versus-speed **s** and **f** families detailed in the parameter section below.

21.2 Learning objectives

The two previous chapters built the pieces of hash-based signing from the ground up: one-time and few-time signatures (Chapter 19) and the Merkle-tree and hypertree machinery that makes them

stateless and many-time (Chapter 20). This chapter assembles those pieces exactly as the standard specifies them. After finishing this chapter, you should be able to:

- explain why SPHINCS$^+$ is the most conservative standard and what "stateless" buys and costs;
- recall how WOTS$^+$, FORS, Merkle trees, and the hypertree fit together;
- explain how every secret is derived deterministically from a few seeds and an address;
- follow the SLH-DSA key generation, signing, and verification algorithms;
- read off the parameter families and their size–speed trade-off.

21.3 Review: the pieces, assembled

It is worth restating the ladder in one place, since SLH-DSA is precisely its assembly:

- **WOTS$^+$** (Chapter 19) signs one value with hash chains and a checksum;
- a **Merkle tree** (Chapter 20) binds 2^h WOTS$^+$ keys under one root, authenticated by a path – one such tree is an XMSS instance;
- a **hypertree** stacks d layers of these trees so that each root is signed by the layer above, exposing only the top root as the public key;
- **FORS** signs the actual message digest as a *few-time* signature, and its public key is what the bottom hypertree layer signs.

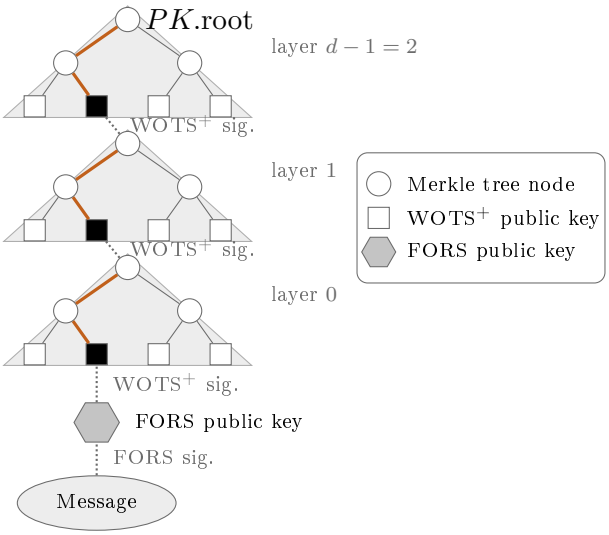


Figure 21.1: An SLH-DSA signature: the full hypertree with $d = 3$ layers. The message is signed by **FORS** (a few-time signature); the FORS public key is signed by a WOTS$^+$ leaf of the layer-0 Merkle tree; each tree's root is signed by a WOTS$^+$ leaf of the tree one layer up; and only the top root PK.root is published. In each tree the used leaf (filled) and its path to the root are highlighted – the authentication data that ties the leaf to the root. Redrawn in the style of FIPS 205 [3].

Figure 21.1 draws a complete SLH-DSA signature: the whole chain of one-time and few-time signatures from the message at the bottom up to the published root (the block-level view of the same hypertree is Figure 20.2). What remains is to specify *where every secret comes from* and to write down the three algorithms.

21.4 The real input: seeds and addresses

Like ML-KEM and ML-DSA, SLH-DSA is deterministic once its seeds are fixed. Everything is derived from three n-byte values:

- `SK.seed` – secret; every WOTS$^+$ and FORS secret value is generated from it by a PRF;
- `SK.prf` – secret; a PRF key used to derive the per-signature randomizer R;
- `PK.seed` – public; tweaks every hash so that structurally identical nodes differ.

The device that keeps the many hashes distinct is the *address* (`ADRS`): a structured label identifying exactly which tree, which layer, which leaf, and which position within a chain a given hash belongs to. Every hash call takes the public seed and an address, so no two structurally identical computations in different places produce the same value. This is what defeats multi-target attacks, and it is why `PK.seed`, though public, is essential.

21.5 Key generation

Key generation samples the three seeds and computes the one value that summarizes the entire hypertree: the root of its top-layer Merkle tree.

Algorithm 27 SLH-DSA.KeyGen (teaching form)

Input: Randomness for three n-byte seeds
Output: Public key (`PK.seed`, `PK.root`); secret key also stores (`SK.seed`, `SK.prf`)

1: sample `SK.seed`, `SK.prf`, `PK.seed` uniformly
2: `PK.root` ← root of the top-layer Merkle tree, derived from `SK.seed` and `PK.seed` via addressed hashing
3: **return** public (`PK.seed`, `PK.root`), secret (`SK.seed`, `SK.prf`, `PK.seed`, `PK.root`)

21.6 Signing and verification

Signing hashes the message (with a randomizer) to a digest, splits it into a FORS message and a pair of indices selecting a bottom-layer tree and leaf, signs the digest with FORS, and then signs the FORS public key up the hypertree.

Algorithm 28 SLH-DSA.Sign (teaching form)

Input: Secret key, formatted message M', optional n-byte randomizer opt_rand
Output: Signature $\sigma = (R,\ \sigma_{\text{FORS}},\ \sigma_{\text{HT}})$

1: $R \leftarrow \text{PRF\_MSG}(\texttt{SK.prf}, opt\_rand, M')$ ▷ zero opt_rand is deterministic; random input is hedged
2: $digest \leftarrow H(R \,\|\, \texttt{PK.seed} \,\|\, \texttt{PK.root} \,\|\, M')$
3: split $digest$ into the FORS message md and indices $(idx_{\text{tree}}, idx_{\text{leaf}})$
4: $\sigma_{\text{FORS}} \leftarrow \text{ForsSign}(md, \dots)$; $pk_{\text{FORS}} \leftarrow$ its public key
5: $\sigma_{\text{HT}} \leftarrow \text{HypertreeSign}(pk_{\text{FORS}}, idx_{\text{tree}}, idx_{\text{leaf}}, \dots)$
6: **return** $(R,\ \sigma_{\text{FORS}},\ \sigma_{\text{HT}})$

Algorithm 29 SLH-DSA.Verify (teaching form)

Input: Public key $(\texttt{PK.seed}, \texttt{PK.root})$, message M, signature $(R, \sigma_{\text{FORS}}, \sigma_{\text{HT}})$
Output: *accept* or *reject*

1: $digest \leftarrow H(R \,\|\, \texttt{PK.seed} \,\|\, \texttt{PK.root} \,\|\, M)$; split into $md, (idx_{\text{tree}}, idx_{\text{leaf}})$
2: $pk_{\text{FORS}} \leftarrow \text{ForsPkFromSig}(md, \sigma_{\text{FORS}}, \dots)$
3: $root' \leftarrow \text{HypertreeRootFromSig}(pk_{\text{FORS}}, \sigma_{\text{HT}}, idx_{\text{tree}}, idx_{\text{leaf}}, \dots)$
4: **return** *accept* if $root' = \texttt{PK.root}$, else *reject*

Verification is nothing but a chain of hash recomputations: rebuild the FORS public key from the message, fold it up the hypertree using the supplied authentication paths, and accept only if the final value equals the published root. There is no arithmetic to protect and no secret-dependent branch, which makes SLH-DSA unusually easy to implement in constant time (Chapter 26).

21.7 Parameter sets

SLH-DSA comes in two flavours at each security level: **s** (*small* signatures, slower signing) and **f** (*fast* signing, larger signatures), instantiated with either SHA2 or SHAKE.

Family	n (bytes)	NIST level	Trade-off
SLH-DSA-128{s,f}	16	1	smallest keys
SLH-DSA-192{s,f}	24	3	balanced
SLH-DSA-256{s,f}	32	5	highest security

Table 21.1: SLH-DSA parameter families. Within each level, the **s** variant minimizes signature size and the **f** variant minimizes signing time.

Signatures are large – roughly 8 to 50 kilobytes – which is the price of relying on hashing alone. In return, the security argument is the simplest and most conservative of any post-quantum scheme.

21.8 SageMath experiment: addressed Merkle authentication

This small hash-only example checks the part of SLH-DSA verification that folds an authentication path back to a public root. It uses eight leaves and a SHA-256-based toy hash; it is not a FORS, WOTS$^+$, or full SLH-DSA implementation.

```
from hashlib import sha256
from operator import xor

n = 16
pk_seed = b"public seed for a toy tree"

def H(label, address, data=b""):
    return sha256(pk_seed + label + address + data).digest()[:n]

def leaf(index):
    return H(b"leaf", int(index).to_bytes(4, "big"))

def parent(left, right, level, index):
    address = (int(level).to_bytes(1, "big") +
               int(index).to_bytes(4, "big"))
    return H(b"node", address, left + right)

levels = [[leaf(i) for i in range(8)]]
for level in range(3):
    previous = levels[-1]
    levels.append([parent(previous[2*i], previous[2*i + 1], level, i)
                   for i in range(len(previous)//2)])
root = levels[-1][0]

leaf_index = 5
auth = [levels[level][xor(int(leaf_index >> level), 1)]
        for level in range(3)]

def root_from_path(node, index, path):
    for level, sibling in enumerate(path):
        parent_index = index >> 1
        node = (parent(node, sibling, level, parent_index)
                if index % 2 == 0 else
                parent(sibling, node, level, parent_index))
        index >>= 1
    return node

assert root_from_path(leaf(leaf_index), leaf_index, auth) == root
bad_auth = list(auth)
bad_auth[0] = bytes([xor(bad_auth[0][0], 1)]) + bad_auth[0][1:]
assert root_from_path(leaf(leaf_index), leaf_index, bad_auth) != root
print("addressed Merkle path accepts intact data and rejects a tampered path")
```

The address is included in every hash call, so the same byte strings at a different layer or node index are domain-separated. That is the small-scale version of the address discipline used throughout SLH-DSA.

21.9 A complete SageMath implementation

The Merkle experiment above is a toy in one respect that matters: it uses a made-up address format. The companion file `sage/fips205_slhdsa.sage` implements the real thing – all twenty-five numbered algorithms of FIPS 205, each as its own function annotated with its algorithm number, for all twelve approved parameter sets. That means both hash families (SHA2 and SHAKE) at all three security levels in both the **s** and **f** variants, together with the WOTS+ chains, the XMSS trees, the hypertree, FORS, the pure and pre-hash top-level algorithms, and the 32-byte address

structure with its 22-byte compressed form.

The address is a small class carrying the member functions of Table 1 of the standard, so the recursive tree walk of Algorithm 9 reads exactly as specified:

```
def xmss_node(self, skseed, i, z, pkseed, adrs):
    """Algorithm 9. Root of a Merkle subtree of WOTS+ public keys."""
    if z == 0:
        adrs.setTypeAndClear(WOTS_HASH)
        adrs.setKeyPairAddress(i)
        return self.wots_pkGen(skseed, pkseed, adrs)
    lnode = self.xmss_node(skseed, 2*i,     z - 1, pkseed, adrs)
    rnode = self.xmss_node(skseed, 2*i + 1, z - 1, pkseed, adrs)
    adrs.setTypeAndClear(TREE)
    adrs.setTreeHeight(z)
    adrs.setTreeIndex(i)
    return self._H(pkseed, adrs, lnode + rnode)
```

The hypertree is then just that tree, stacked d deep, with each layer signing the root of the layer below:

```
def ht_sign(self, M, skseed, pkseed, idxtree, idxleaf):
    """Algorithm 12. A hypertree signature: d chained XMSS signatures."""
    adrs = ADRS()
    adrs.setTreeAddress(idxtree)
    SIGtmp = self.xmss_sign(M, skseed, idxleaf, pkseed, adrs)
    SIG_HT = SIGtmp
    root   = self.xmss_pkFromSig(idxleaf, SIGtmp, M, pkseed, adrs)
    for j in range(1, self.d):
        idxleaf = idxtree % (1 << self.hp)   # low h' bits
        idxtree = idxtree >> self.hp
        adrs.setLayerAddress(j)
        adrs.setTreeAddress(idxtree)
        SIGtmp  = self.xmss_sign(root, skseed, idxleaf, pkseed, adrs)
        SIG_HT += SIGtmp
        if j < self.d - 1:
            root = self.xmss_pkFromSig(idxleaf, SIGtmp, root, pkseed, adrs)
    return SIG_HT
```

What Sage contributes to a scheme with no algebra. SLH-DSA is built from hash functions alone, so unlike ML-KEM and ML-DSA there is no ring here for a computer-algebra system to check the code against. What Sage supplies instead is exact integer reasoning about the combinatorics. The file's `verify_parameter_consistency()` re-derives Table 2 from the defining equations for all twelve parameter sets: that $h = d \cdot h'$, so the hypertree really has 2^h leaves; that the checksum length len_2 produced by Algorithm 1 agrees with the closed form $\lfloor \log_w(len_1(w-1)) \rfloor + 1$, computed with Sage's exact integer logarithm; that the digest length m is large enough to supply the FORS message and both tree indices; and that the published key and signature sizes match $2n$ and $(1 + k(1+a) + h + d \cdot len)\, n$ respectively. A parameter table is exactly the kind of thing that is easy to transcribe wrongly and hard to notice, and this turns it into a test.

Validation. The implementation is checked against NIST's ACVP vectors [5] for FIPS 205 – key generation, signature generation, and signature verification, across all twelve parameter sets in both the pure and pre-hash forms – and reproduces them byte for byte.

One practical caveat is worth stating plainly, because it is a property of the scheme and not of the code. Signing with an `s` parameter set requires rebuilding whole XMSS trees: SLH-DSA-

128s traverses on the order of a million hash calls per signature, which in Sage takes about sixteen seconds, and the 256-bit variants about twice that. The quick test run therefore covers signature *generation* only for the **f** sets, and reports how many groups it left out; `make kat-full` includes them. Verification is cheap for every parameter set and is always checked in full. This asymmetry is the same one that gives the two variants their names.

21.10 Pitfalls

Pitfall 1: reusing a one-time (WOTS$^+$) leaf. Signing two different messages with the same WOTS$^+$ chain reveals enough chain points to forge (Chapter 19). The address mechanism and message randomization make accidental collisions extremely unlikely; neither WOTS nor FORS addresses may be intentionally reused.

Pitfall 2: thinking SLH-DSA needs state. Earlier hash-based schemes (XMSS, LMS) are *stateful*: the signer must remember which leaves are spent, and a single reuse is catastrophic. SLH-DSA is *stateless* – the message hash selects the path – which is why it was chosen for standardization despite larger signatures.

Pitfall 3: forgetting the public seed's role. `PK.seed` tweaks every hash call through the address mechanism. Without it, structurally identical nodes in different trees would hash identically, enabling multi-target attacks. It is public but essential.

21.11 Summary

SLH-DSA is the assembly of the hash-based ladder into a single stateless standard:

- every secret is derived deterministically from `SK.seed` and an address; `PK.seed` keeps all hashes distinct;
- a signature is a FORS signature on the message digest, carried up a hypertree of Merkle-tree (XMSS) instances to the one published root;
- verification is pure hashing, so the scheme is conservative in assumption and easy to make constant-time.

Its security depends on nothing but the hash function, making it the backstop among post-quantum signatures. This completes the two signature families – lattice-based (ML-DSA, FN-DSA) and hash-based (SLH-DSA). The next part turns to a third and independent foundation: code-based cryptography.

Part IV

Code-Based Cryptography

Chapter 22

Classic McEliece

22.1 Learning objectives

So far the book has drawn on two hard-problem families: lattices (ML-KEM, ML-DSA, FN-DSA) and hash functions (SLH-DSA). This part opens a third, entirely independent foundation. Code-based cryptography does not ask you to find short vectors in a lattice; it asks you to *decode* a random-looking linear code. It is the oldest post-quantum family of all – Robert McEliece proposed his scheme in 1978, the same year RSA appeared [34] – and after more than forty years of attack it stands essentially unbroken. That track record is precisely why it is prized: where the lattice standards trade a little conservatism for small keys and speed, Classic McEliece makes the opposite bet.

After finishing this chapter, you should be able to:

1. state the basic objects of a binary linear code – generator matrix, parity-check matrix, Hamming weight and distance, and the syndrome of an error;

2. explain the *syndrome decoding* problem and why its NP-hardness (with no known quantum shortcut beyond Grover) is the security foundation of the scheme;

3. describe the trapdoor: a secret code that *can* be decoded efficiently, disguised as a code that cannot;

4. follow the KeyGen, Encapsulate, and Decapsulate algorithms and see exactly where the secret decoder is used;

5. explain the trade-off Classic McEliece makes – very large public keys in exchange for tiny ciphertexts and a long, conservative security record.

Why this chapter matters. Code-based cryptography is one of the three problem families named in the Quantum Threat chapter (Chapter 1), alongside lattices and hashes. Each is a different bet against an uncertain quantum future, and the value of holding more than one bet is that they fail independently: a breakthrough against lattices would say nothing about codes. Classic McEliece is the flagship of the code-based hedge. Understanding it also completes the picture of what "post-quantum" means – not one idea, but several unrelated hard problems, any one of which could carry public-key cryptography through the quantum era.

22.2 A crash course in linear codes

The reader has met lattices and polynomial rings but not, so far, coding theory. This section builds the handful of objects the scheme needs. Everything lives over the binary field $\mathbb{F}_2 = \{0, 1\}$, where addition is XOR.

Codes as subspaces. A **binary linear code** C is simply a linear subspace of $\mathbb{F}_2^n$. If that subspace has dimension k, we call C an $[n, k]$ code: it contains 2^k codewords, each an n-bit string. The idea is redundancy. We take a k-bit message and spread it across $n > k$ bits, so that if a few of those bits are flipped in transit we can still recover the original.

The generator matrix. Because C is a k-dimensional subspace, it has a basis of k codewords. Stacking that basis as the rows of a $k \times n$ matrix G gives the **generator matrix**. Encoding a message $m \in \mathbb{F}_2^k$ is then a single matrix product:

$$c = m\,G \in \mathbb{F}_2^n.$$

As m ranges over all 2^k messages, mG ranges over all codewords. The generator matrix *is* the code, written down.

The parity-check matrix. The same subspace can be described from the outside, by the linear constraints its codewords satisfy. The **parity-check matrix** H is an $(n-k) \times n$ matrix whose null space is exactly C:

$$H\,c^\top = 0 \quad \text{for every codeword } c \in C.$$

So G tells you how to *build* codewords and H tells you how to *recognize* them. The two are linked by $GH^\top = 0$.

Weight, distance, and error correction. The **Hamming weight** $\mathrm{wt}(x)$ of a vector is the number of nonzero coordinates, and the **Hamming distance** between two vectors is the weight of their difference, $d(x, y) = \mathrm{wt}(x - y)$. The **minimum distance** d of a code is the smallest distance between two distinct codewords. A code with minimum distance d can *correct* up to

$$t = \left\lfloor \tfrac{d-1}{2} \right\rfloor$$

bit errors: if a codeword is corrupted in at most t positions, it is still closer to the original codeword than to any other, so the original can in principle be recovered.

The syndrome. Suppose a codeword c is sent and arrives as $y = c + e$, where e is an unknown **error vector**. Multiplying the received word by the parity-check matrix gives

$$s = H\,y^\top = H\,c^\top + H\,e^\top = H\,e^\top,$$

because $Hc^\top = 0$. The result $s \in \mathbb{F}_2^{n-k}$ is the **syndrome**. Two things are worth pausing on. First, the syndrome depends only on the error, not on which codeword was sent. Second, *decoding* – recovering e (and hence c) – is exactly the task of finding a low-weight e that produces the observed syndrome. This single observation is the hinge of the entire chapter.

22.3 The hard problem: syndrome decoding

For a code with helpful structure, decoding is easy – that is the whole point of designing codes. But strip the structure away and the picture inverts.

> **Syndrome Decoding.** Given a *random* parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$, a target syndrome $s \in \mathbb{F}_2^{n-k}$, and a weight t, find an error vector e with $\mathrm{wt}(e) \le t$ and $H\,e^\top = s$.

For a code with no exploitable structure, this is genuinely hard. It was proved **NP-complete** by Berlekamp, McEliece, and van Tilborg in 1978 [38] – the same year the encryption scheme was proposed. Intuitively, the linear system $He^\top = s$ has many solutions, but almost all of them have large weight; the constraint $\mathrm{wt}(e) \le t$ picks out an exponentially rare needle, and there is no linear-algebra shortcut to it.

The best known attacks are the family of **information-set decoding** (ISD) algorithms, refined steadily since the 1960s. They run in time exponential in the code parameters, and decades of optimization have improved only the constant in the exponent, never removed it. Crucially, quantum computers do not change this picture qualitatively: the strongest quantum variants apply Grover-style search inside ISD and gain at most a square-root speedup, exactly the mild erosion the Quantum Threat chapter described for all the post-quantum families. There is no Shor-like collapse. Doubling the parameters restores the lost margin, just as it does for a symmetric cipher.

This is the security foundation. If we can present an attacker with what looks like a random parity-check matrix and a syndrome, and the secret is a low-weight error, then breaking the scheme means solving syndrome decoding.

22.4 The trapdoor: a decodable code in disguise

The problem, of course, is that the honest parties also see only the public matrix. If it were truly random, *nobody* could decode, and the scheme would be useless. The resolution is the central trick of trapdoor cryptography: start from a code that *does* have an efficient decoder, then disguise it so thoroughly that from the outside it is indistinguishable from a random code.

The secret code. Classic McEliece builds its secret from a **binary Goppa code**. The full construction, parameters, and decoding of these codes are developed in Appendix A; here the two properties we rely on are simple to state:

- it is an $[n, k]$ linear code, with a generator matrix G, chosen so that it can correct t errors;
- it comes with an *efficient* t-error decoding algorithm $\mathcal{D}_g$ (Patterson's algorithm), which takes any word within distance t of a codeword and returns the codeword and the error.

Goppa codes are the workhorse here for a specific reason: after four decades of scrutiny, a scrambled Goppa code still cannot be told apart from a random linear code by any known efficient test. Other code families that looked equally promising (for instance, certain algebraic codes) were later shown to leak their structure and were broken. Goppa codes have held.

The disguise. Two secret ingredients hide the structure:

- a random invertible **scrambler** matrix S of size $k \times k$ over $\mathbb{F}_2$, which mixes the rows of G (it changes the basis, not the code);

- a random $n \times n$ **permutation** matrix P, which shuffles the coordinates.

The public key is the transformed generator matrix

$$G' = S\,G\,P.$$

Scrambling by S and permuting by P both preserve the fact that this is *some* $[n, k]$ code, but they destroy every visible trace of the Goppa structure that made G decodable. To anyone without S, P, and $\mathcal{D}_g$, the matrix G' is just a random-looking code with no known efficient decoder.

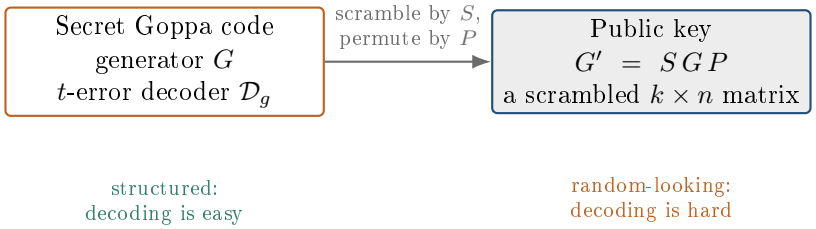


Figure 22.1: The trapdoor as a disguise. A structured Goppa code, which the secret decoder $\mathcal{D}_g$ can correct up to t errors, is scrambled by an invertible matrix S and a coordinate permutation P into a public matrix G' that looks like a random code. Anyone can *encrypt* against G' – take a codeword and add a weight-t error – but only the holder of S, P, and $\mathcal{D}_g$ can peel the disguise away and *decode*. Everyone else faces syndrome decoding of a random code.

The asymmetry in the caption is the whole scheme in one sentence. Encryption adds a weight-t error to a public codeword; this is easy for anyone. Decryption removes that error; this is easy only for the trapdoor holder and, for everyone else, is syndrome decoding.

22.5 The scheme

We present two views of the same idea. The original 1978 form encrypts a message directly and is the clearest illustration of the trapdoor. The modern form – the one the NIST "Classic McEliece" submission [39] actually specifies – is a key-encapsulation mechanism (KEM) built on the dual, parity-check description of the code. Both rely on the same secret decoder.

22.5.1 The original 1978 encryption

Key generation produces the disguise of the previous section: a secret Goppa code with decoder $\mathcal{D}_g$, a scrambler S, a permutation P, and the public generator $G' = SGP$.

To **encrypt** a message $m \in \mathbb{F}_2^k$, the sender encodes it against the public matrix and deliberately adds a fresh random error e of weight exactly t:

$$c = m\,G' + e, \qquad \mathrm{wt}(e) = t.$$

The ciphertext c is a codeword of the public code, knocked off the code in t positions. To an eavesdropper, recovering m means removing e – syndrome decoding of a random-looking code.

To **decrypt**, the receiver peels the disguise off one layer at a time, and this is where the secret is spent. The key algebraic fact is that a permutation only reshuffles the error's positions, so it never changes its weight.

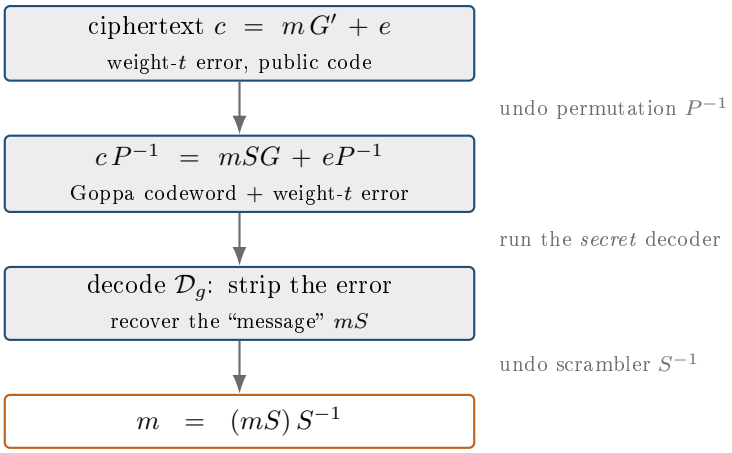


Figure 22.2: McEliece decryption peels the disguise in reverse. Applying P^{-1} turns the ciphertext into a genuine Goppa codeword mSG plus an error eP^{-1} of the same weight t; the secret decoder $\mathcal{D}_g$ removes that error and returns mS; finally S^{-1} recovers the message m. The middle step is the only one an attacker cannot perform, because only the trapdoor holder knows the Goppa structure.

Written out, decryption is: compute $cP^{-1} = mSG + eP^{-1}$, which is a Goppa codeword mSG corrupted in exactly t positions; run $\mathcal{D}_g$ to strip the error and read off mS; then multiply by S^{-1} to get m. Without the secret decoder, the second step is impossible, and that is the whole security of the scheme.

22.5.2 The modern KEM: sending a syndrome

The NIST submission uses the **Niederreiter** presentation [37], which is the dual of the picture above. Instead of a scrambled generator matrix it publishes a scrambled parity-check matrix, and instead of encrypting a chosen message it encapsulates a random key. This form has smaller ciphertexts and is what you should think of as "Classic McEliece" today.

The public key is now a disguised parity-check matrix

$$H' = S\,H\,P,$$

where H is the parity-check matrix of the secret Goppa code, S is an invertible $(n-k)\times(n-k)$ scrambler, and P is again a coordinate permutation. (In the actual specification H' is reduced to systematic form $[\,I \mid T\,]$, so only the sub-block T needs to be stored; the scrambler S is exactly the row reduction that achieves this. We keep S explicit here for clarity.)

The idea of encapsulation is that *the secret is a low-weight error vector*, and the ciphertext is its syndrome. Because the syndrome of a weight-t error under a random-looking H' is a hard thing to invert, the ciphertext hides the error from everyone but the trapdoor holder. The shared key is then derived by hashing that error.

Algorithm 30 Classic McEliece KeyGen (teaching form)

Input: Code parameters (n, k, t)
Output: Public key H'; secret key $(\mathcal{D}_g, S, P)$

choose a binary Goppa code with parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ and efficient t-error decoder $\mathcal{D}_g$ ▷ the structured secret code
$S \leftarrow$ random invertible $(n-k)\times(n-k)$ matrix over $\mathbb{F}_2$ ▷ scrambler
$P \leftarrow$ random $n \times n$ permutation matrix ▷ coordinate shuffle
$H' \leftarrow S\,H\,P$ ▷ disguise: looks like a random code
return $\big(H',\ (\mathcal{D}_g, S, P)\big)$

Encapsulation never touches the secret. It just samples a random error of the prescribed weight and reports its syndrome.

Algorithm 31 Classic McEliece Encapsulate (teaching form)

Input: Public key H'; parameters (n, t)
Output: Ciphertext c; shared key K

$e \leftarrow$ random vector in $\mathbb{F}_2^n$ with $\mathrm{wt}(e) = t$ ▷ the secret is the error itself
$c \leftarrow H'\,e^\top$ ▷ ciphertext is the syndrome, only $n-k$ bits
$K \leftarrow \mathrm{Hash}(e)$ ▷ shared key hashes the error
return (c, K)

Decapsulation is where the trapdoor is spent. The receiver undoes the scrambler, hands the result to the secret Goppa decoder to recover a permuted error, then undoes the permutation and re-hashes – arriving at the same key the sender computed.

Algorithm 32 Classic McEliece Decapsulate (teaching form)

Input: Ciphertext c; secret key $(\mathcal{D}_g, S, P)$
Output: Shared key K

$s' \leftarrow S^{-1}\,c$ ▷ now s' is a Goppa syndrome: $s' = H(\,eP^\top)^\top$
$\tilde{e} \leftarrow \mathcal{D}_g(s')$ ▷ *secret* decoder recovers the weight-t error
$e \leftarrow \tilde{e}\,P$ ▷ undo the permutation
$K \leftarrow \mathrm{Hash}(e)$ ▷ re-derive the shared key
return K

The correctness is the mirror of Figure 22.2: multiplying by S^{-1} turns the public ciphertext $c = SHP\,e^\top$ into $H(Pe^\top)$, a genuine syndrome of the Goppa code for a permuted-but-same-weight error; the decoder $\mathcal{D}_g$ recovers that error because its weight is exactly t; undoing P restores the original e; and hashing e reproduces the sender's key. An attacker who sees only H' and c is left with syndrome decoding.

A note on real-world hardening. The teaching form above sets $K = \mathrm{Hash}(e)$. To achieve security against active (chosen-ciphertext) attackers, the actual specification hashes the ciphertext into the key as well and uses *implicit rejection*: if decoding fails or the recovered error has the wrong weight, decapsulation does not abort but returns a pseudorandom key derived from a secret stored in the key. This hides from the attacker whether decoding succeeded, closing a class of reaction

attacks. The essential structure – error in, syndrome out, decode with the trapdoor – is exactly as shown.

22.6 Trade-offs and status

Classic McEliece makes an unusual and deliberate trade. Its public key is enormous – a dense matrix of hundreds of kilobytes, up to about a megabyte at the highest security level – because the public key essentially *is* a random-looking code, and there is no seed to compress it down to (contrast the lattice standards, where the public matrix regenerates from a 32-byte seed). But everything else is small and fast: the ciphertext is merely a syndrome, well under a few hundred bytes, and encapsulation and decapsulation are quick. Where a lattice KEM balances all its sizes, Classic McEliece pushes the entire cost into a one-time, static public key that can be generated once and reused for a long time.

Scheme	Public key	Ciphertext	Hardness assumption
Classic McEliece `348864`	≈ 261 KB	96 B	syndrome decoding (codes)
Classic McEliece `6960119`	≈ 1.0 MB	194 B	syndrome decoding (codes)
ML-KEM-768	1184 B	1088 B	Module-LWE (lattices)

Table 22.1: Classic McEliece against a lattice KEM. The public keys differ by three orders of magnitude, but the ciphertexts run the other way, and – the point of this whole part of the book – the two rest on *independent* hard problems. A break of one says nothing about the other.

Status. Classic McEliece competed in the NIST post-quantum project as a key-encapsulation candidate and advanced into the fourth round dedicated to additional KEMs. It is not a published NIST FIPS, and NIST IR 8545 selected HQC rather than Classic McEliece for the additional-KEM standardization path. Classic McEliece nevertheless has a long public cryptanalytic record; that record is a stable reason to study it, without making unsupported claims about present deployment or other standardization activity.

22.7 Pitfalls

Pitfall 1: thinking the large public key is a flaw to be fixed. The size is intrinsic. The public key is a random-looking code, and a random code has no shorter description than the matrix itself. You cannot regenerate it from a small seed the way ML-KEM regenerates its matrix A, because a seed-expandable matrix would carry exploitable structure. The large key is the price of the conservative assumption, not an implementation oversight.

Pitfall 2: thinking the error weight is negotiable. The decoder $\mathcal{D}_g$ corrects *up to* t errors and no more. Encapsulation must use weight exactly t: too few and it is easier to attack, too many and honest decoding fails. The weight is a fixed scheme parameter, not a tunable knob at encryption time.

Pitfall 3: assuming any decodable code family will do. The security rests on the scrambled code being *indistinguishable* from random. Several families that offered efficient decoders (such as

certain algebraic and low-density codes) were shown to leak their structure through the disguise and were broken. Binary Goppa codes are used precisely because, so far, they do not.

Pitfall 4: skipping the chosen-ciphertext hardening. The bare scheme in the algorithm boxes is only passively secure. Real deployments must bind the ciphertext into the key and use implicit rejection, exactly as the lattice KEMs apply a Fujisaki–Okamoto-style transform. Omitting it exposes reaction attacks that learn the secret from whether decoding succeeds.

22.8 Summary

Code-based cryptography rests on a hard problem wholly separate from lattices: decoding a random linear code, formalized as syndrome decoding, which is NP-hard and enjoys no quantum shortcut beyond Grover's mild square-root erosion. Classic McEliece turns this into a cryptosystem by the classic trapdoor manoeuvre:

- it starts from a **binary Goppa code**, which has an efficient t-error decoder $\mathcal{D}_g$;
- it **disguises** that code with a secret scrambler S and permutation P, publishing $G' = SGP$ (or, dually, $H' = SHP$), which looks random;
- anyone can **encapsulate** – add or send a weight-t error – but only the holder of S, P, and $\mathcal{D}_g$ can **decapsulate**, because only they can decode;
- it buys unusually strong **conservatism** – a forty-year unbroken record – at the cost of a very large public key, while keeping ciphertexts tiny.

The lesson beyond the mechanics is that "post-quantum" is not a single idea but a portfolio of independent bets. Classic McEliece is the code-based family's oldest and most cautious member, decoding disguised as a random code. The next chapter, on HQC, takes a different route through the same code-based territory: rather than hiding a strongly structured Goppa code, it builds security on the hardness of decoding *quasi-cyclic* codes, trading Classic McEliece's giant keys for something far more compact – and, in doing so, became the code-based scheme NIST chose to standardize.

Chapter 23

HQC

23.1 Learning objectives

After reading this chapter, you should be able to:

- explain how HQC differs from Classic McEliece (Chapter 22): McEliece *hides the code*, HQC *hides the noise*;
- state the hard problem HQC rests on – decoding a random quasi-cyclic code – and why the code itself is public;
- read the ring $\mathbb{F}_2[x]/(x^n-1)$ as quasi-cyclic blocks, and multiplication in it as cyclic convolution over $\mathbb{F}_2$;
- follow the KeyGen / Encapsulate / Decapsulate pipeline and verify the algebraic cancellation $v - u\,y \approx mG$ by hand;
- explain why a publicly known, efficiently decodable code C removes the leftover low-weight error and recovers the message;
- place HQC among the schemes selected by NIST for standardization as a code-based complement to ML-KEM.

23.2 Why this chapter matters

The previous chapter built Classic McEliece, the oldest code-based scheme, whose security has survived since 1978. Its drawback is well known: the public key is enormous, hundreds of kilobytes to a megabyte. HQC (*Hamming Quasi-Cyclic*) is a much younger code-based KEM that attacks the same goal – post-quantum key encapsulation from coding theory – but from the opposite direction, and with far smaller keys.

The reason this matters beyond efficiency is *diversity*. Nearly all of the first published NIST post-quantum standards rest on lattice problems: ML-KEM and ML-DSA are lattice schemes. If a single unexpected advance were to weaken lattice assumptions, a backup built on a genuinely different hard problem would be valuable. In March 2025 NIST selected **HQC** [40] for standardization as an additional key-encapsulation mechanism, to complement ML-KEM with a scheme whose security rests on the hardness of decoding random codes rather than on lattices. At this standards snapshot, the final HQC FIPS has not yet been published; this chapter describes the current submission/specification rather than a final compliance interface.

23.3 The key idea: hide the noise, not the code

Both Classic McEliece and HQC are code-based, and both ultimately rely on the fact that *decoding a random-looking linear code is hard*. But they exploit that fact in opposite ways.

- **Classic McEliece hides the code.** The secret is a structured code (a binary Goppa code) that the owner knows how to decode efficiently. The public key is a scrambled generator matrix that looks like a random code with no usable structure. Security rests on the attacker being unable to tell the disguised structured code apart from a random one, and being unable to decode it. Because the public object is essentially a full random-looking generator matrix, it is large.
- **HQC hides the noise.** The decodable code C is *public and fixed*. Everyone – including the attacker – knows C and knows how to decode it. What is secret is a pair of *low-weight* vectors. Security rests on the attacker being unable to recover those low-weight secrets from the public data, which is exactly the problem of decoding a random *quasi-cyclic* code. Nothing structural is hidden, so there is no large disguised matrix to publish.

This is the same move that took us from LWE to Ring-LWE earlier in the book, transplanted into coding theory. Plain code-based cryptography, like plain LWE, needs a big unstructured matrix. Adding *quasi-cyclic* structure lets a single row (a single ring element) stand in for an entire circulant block, shrinking the key by a large factor – exactly as a circulant matrix is determined by its first row. HQC is to McEliece roughly as Ring-LWE is to LWE: it buys small keys by adding algebraic structure, at the cost of relying on a more structured assumption. The NTRU construction of Section 12.15 makes the same trade in the lattice world; HQC is its coding-theory cousin.

23.4 The ring: quasi-cyclic blocks as ring elements

Fix a length n (a prime, in the real scheme). The objects of HQC live in the ring

$$R = \mathbb{F}_2[x]/(x^n - 1).$$

An element of R is a binary polynomial of degree less than n, equivalently a binary vector of length n. Two facts make this ring the right home for HQC.

Multiplication is cyclic convolution. Because $x^n \equiv 1$, multiplying by x cyclically rotates the coefficient vector. Multiplying two ring elements h and y therefore convolves their coefficient vectors cyclically over $\mathbb{F}_2$:

$$(h \cdot y)_k \;=\; \bigoplus_{i+j \equiv k \pmod{n}} h_i \, y_j.$$

This uses the cyclic/circulant side of the convolution picture introduced in Chapter 5: the map $y \mapsto h \cdot y$ is multiplication by the circulant matrix whose first row is h. Storing the single vector h is enough to describe that entire $n \times n$ circulant – this is the "quasi-cyclic" compression, and it is why HQC keys are small.

Weight is what stays secret. The *Hamming weight* of a ring element is the number of nonzero coefficients. A *low-weight* element has only a handful of ones among its n coordinates. HQC's secrets are low-weight elements; a uniformly random h is not. The whole security argument turns on the gap between "low weight" and "random." (For scale, HQC-128 uses $n = 17669$ with secret weight $w = 66$: sixty-six ones hidden among more than seventeen thousand coordinates.)

23.5 The scheme

HQC has the same algebraic silhouette as the toy ML-KEM pipeline (Figure 14.2): a public "syndrome" built from a secret, a ciphertext that reuses the public key against fresh randomness, and a decryption that cancels the shared term and decodes a small leftover error. Only the arithmetic changes – integers modulo q become bits, and "small" means *low Hamming weight* rather than *small magnitude*.

23.5.1 Key generation

Sample a uniformly random $h \in R$ and two low-weight secrets $x, y \in R$ (each of weight w). Publish

$$s \;=\; x + h \cdot y.$$

The public key is (h, s) and the secret key is (x, y).

The pair (h, s) is exactly a quasi-cyclic syndrome: s is a random-looking ring element, and recovering the low-weight (x, y) from (h, s) is the **Quasi-Cyclic Syndrome Decoding** problem, believed hard even for a quantum computer. Note that h and s together are just two length-n vectors – there is no large matrix to store.

Algorithm 33 HQC KeyGen (teaching form)

Input: Ring $R = \mathbb{F}_2[x]/(x^n - 1)$; secret weight w
Output: Public key (h, s); secret key (x, y)

1: $h \leftarrow$ uniformly random element of R
2: $x \leftarrow$ random element of R of Hamming weight w
3: $y \leftarrow$ random element of R of Hamming weight w
4: $s \leftarrow x + h \cdot y$ ▷ quasi-cyclic syndrome; looks random
5: **return** $\big((h, s),\ (x, y)\big)$

23.5.2 Encapsulation

Encapsulation encrypts a message m under the public code C, then derives the shared key by hashing m. Let G be the public generator matrix of C, so that $mG \in R$ is the codeword carrying the k-bit message m. Sample fresh low-weight randomness r_1, r_2, e and compute

$$u \;=\; r_1 + h \cdot r_2, \qquad v \;=\; mG + s \cdot r_2 + e.$$

The ciphertext is (u, v) and the shared key is $K = H(m)$ for a hash function H.

The two halves mirror KeyGen: u reuses the public h against the fresh secret r_2 (just as s used h against y), and v hides the codeword mG under the public s times r_2 plus a little extra noise e. Every one of r_1, r_2, e is low weight – that is what makes decapsulation's leftover small.

Algorithm 34 HQC Encapsulate (teaching form)

Input: Public key (h, s); public code C with generator G; message $m \in \mathbb{F}_2^k$
Output: Ciphertext (u, v); shared key K

$r_1 \leftarrow$ low-weight element of R ▷ weight w_r
$r_2 \leftarrow$ low-weight element of R ▷ weight w_r
$e \leftarrow$ low-weight element of R ▷ weight w_e
$u \leftarrow r_1 + h \cdot r_2$
$v \leftarrow mG + s \cdot r_2 + e$ ▷ mG is the codeword of C carrying m
$K \leftarrow H(m)$ ▷ shared secret derived from the message
return $\big((u, v),\ K\big)$

23.5.3 Decapsulation

The holder of (x, y) computes $v - u \cdot y$ and decodes the result with the *public* code C. The point of the algebra is that the term shared between s and u cancels, leaving a codeword plus a low-weight error:

$$\begin{aligned} v - u \cdot y &= \big(mG + s\,r_2 + e\big) - \big(r_1 + h\,r_2\big)\,y \\ &= mG + (x + h\,y)\,r_2 + e - r_1\,y - h\,r_2\,y \\ &= mG + x\,r_2 + \underbrace{h\,y\,r_2 - h\,r_2\,y}_{=0} + e - r_1\,y \\ &= mG + \big(x\,r_2 - y\,r_1 + e\big). \end{aligned}$$

The two copies of $h\,y\,r_2$ cancel because R is commutative. (Over $\mathbb{F}_2$ subtraction is the same as addition, so the leftover is really $xr_2 + yr_1 + e$; we keep the minus signs only to mirror the cancellation.) The leftover

$$e' \;=\; x\,r_2 - y\,r_1 + e$$

is a sum of products of low-weight vectors: $x\,r_2$ has weight at most $w \cdot w_r$, and likewise $y\,r_1$, so e' has small, bounded weight. It is precisely a low-weight error added to the genuine codeword mG. Because C is a *known, efficiently decodable* code chosen so that its decoder corrects up to that many errors, running the public decoder on $v - u\,y$ removes e' and returns m.

Algorithm 35 HQC Decapsulate (teaching form)

Input: Secret key (x, y); public code C with decoder $\textsc{Decode}_C$; ciphertext (u, v)
Output: Shared key K

$w' \leftarrow v - u \cdot y$ ▷ $= mG + (xr_2 - yr_1 + e)$: codeword + low-weight error
$m' \leftarrow \textsc{Decode}_C(w')$ ▷ public decoder removes the low-weight error
$K \leftarrow H(m')$
return K

Decapsulation succeeds whenever the weight of e' stays within the decoding radius of C. As in ML-KEM, the parameters n, w, w_r, w_e and the code C are tuned so that this fails only with negligible probability.

23.5.4 The pipeline at a glance

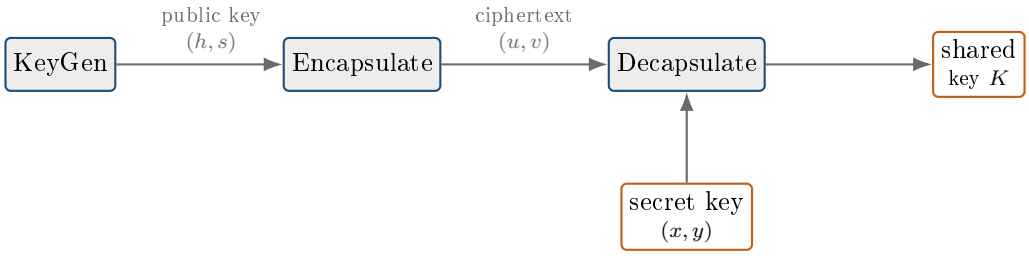


Figure 23.1: The HQC pipeline. KeyGen publishes (h, s) and keeps (x, y); Encapsulate turns a message into a ciphertext (u, v) and a shared key $K = H(m)$; Decapsulate uses the secret (x, y) to recover m and recompute K. The public code C is fixed and known to everyone – it is not part of any key.

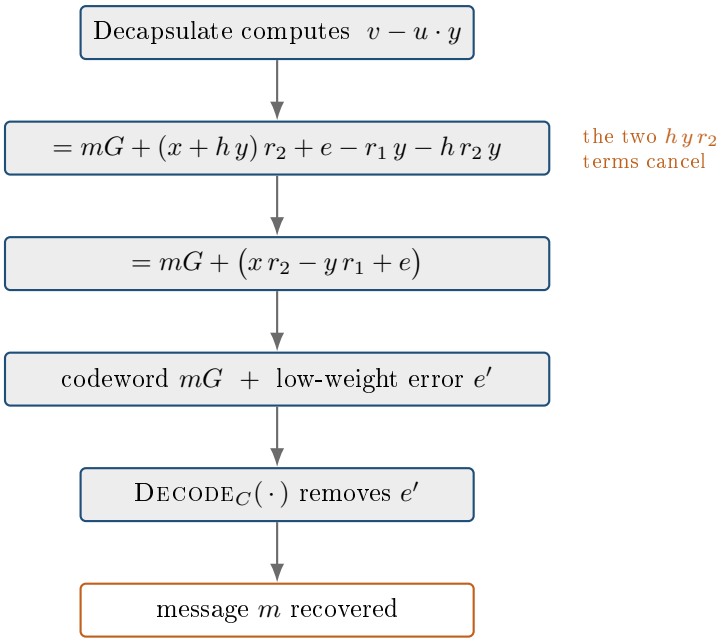


Figure 23.2: Why decapsulation works. The public term shared between s and u (the two copies of $h\,y\,r_2$) cancels because the ring is commutative, leaving the genuine codeword mG plus a low-weight leftover $e' = xr_2 - yr_1 + e$. Since every ingredient is low weight, e' is small enough for the public decoder of C to strip away, revealing m.

23.6 The public code C

Everything hinges on C being a fixed, public code that is *efficiently decodable* up to the weight of e'. This is the exact opposite of McEliece, where the ability to decode is the closely guarded secret. In HQC there is nothing to hide about C: the attacker gains no advantage from knowing how to decode it, because decoding $v - u\,y$ requires first computing $u\,y$, which needs the secret y.

The current HQC specification uses a *concatenated* code for C: an outer Reed–Solomon code composed with an inner duplicated Reed–Muller code. The concatenation is chosen for decoding

performance – a large, reliable error-correction radius at a good rate – since C contributes nothing to security. For this chapter the only property that matters is the interface: C has a public generator G (so mG is a codeword) and a public decoder $\textsc{Decode}_C$ that corrects any error of weight up to its radius.

23.7 From CPA to CCA: the FO transform

Algorithms 33–35 describe the *passively secure* core: they resist an eavesdropper, but not an attacker who submits chosen ciphertexts and watches how decapsulation responds. As with ML-KEM, HQC upgrades this core to chosen-ciphertext (CCA) security with a **Fujisaki–Okamoto (FO) transform** [26].

At a high level, the FO transform makes encapsulation *deterministic in the message*: the randomness r_1, r_2, e is not sampled freshly but derived by hashing m. Decapsulation then recovers m', *re-runs* encapsulation with that same m', and checks that the resulting ciphertext matches the one received. If it does, the ciphertext was honestly formed and the key $K = H(m')$ is returned; if it does not, decapsulation returns a pseudorandom key derived from a secret seed instead of signalling an error. This "re-encrypt and compare" step is what defeats chosen-ciphertext attacks, and it is the same mechanism used throughout the lattice KEMs. The teaching algorithms above deliberately omit it to keep the algebra in focus.

23.8 Where HQC sits

The following table places HQC between its code-based ancestor and the lattice KEM it backs up. The single most useful column is *what is hidden*: it is the cleanest way to remember why the key sizes come out as they do.

	Classic McEliece	HQC	ML-KEM
Family	code-based	code-based	lattice-based
What is hidden	the decodable code	low-weight noise	low-norm noise
The code C	secret (disguised)	public and fixed	—
Hard problem	decode hidden Goppa code	decode random QC code	Module-LWE
Public key	large ($\sim$100s kB)	medium ($\sim$few kB)	small ($<$ 1 kB)
Extra structure	none (unstructured)	quasi-cyclic	module / ring

Table 23.1: Classic McEliece versus HQC versus ML-KEM. McEliece hides an entire structured code and pays for it with a huge public key; HQC keeps the code public and hides only low-weight noise, so quasi-cyclic structure shrinks the key to a few kilobytes; ML-KEM is smaller still but rests on lattices. HQC and ML-KEM solve the same problem on deliberately different assumptions.

23.9 Pitfalls

Pitfall 1: thinking the code is the secret. In McEliece the secret is the ability to decode; in HQC the code C and its decoder are fully public. Confusing the two inverts the whole security argument. In HQC what protects the message is the secret *low-weight* pair (x, y), without which one cannot even form $u\,y$, let alone decode.

Pitfall 2: reading "low weight" as "small numbers." Everything is over $\mathbb{F}_2$, so coefficients are only 0 or 1. "Small" here means *few ones* – low Hamming weight – not small magnitude. The security gap is between a weight-w vector and a uniformly random one, which is roughly half weight.

Pitfall 3: forgetting that the leftover error must fit the decoder. Decapsulation only works if $e' = xr_2 - yr_1 + e$ stays within the decoding radius of C. Raising the secret weights w, w_r, w_e strengthens the assumption but enlarges e'; if it exceeds the radius, decoding fails. This is HQC's analogue of ML-KEM's decryption-failure budget, and the parameters are tuned to make failure negligible.

Pitfall 4: assuming quasi-cyclic structure is free. Adding quasi-cyclic structure is what shrinks the key from McEliece's hundreds of kilobytes to a few, but it also narrows the assumption: HQC relies on decoding random *quasi-cyclic* codes, a more structured problem than decoding fully random codes. This is the same efficiency-versus-structure trade seen with Ring-LWE and NTRU (Section 12.15), and it is a conscious design choice, not an oversight.

Pitfall 5: using the passive core directly. The teaching algorithms here are only CPA-secure. A deployable HQC wraps them in the FO transform's re-encrypt-and-compare check; skipping it exposes the scheme to chosen-ciphertext attacks.

23.10 Summary

- **HQC hides the noise, not the code.** The decodable code C is public and fixed; the secret is a pair of low-weight vectors (x, y).
- **Security is quasi-cyclic syndrome decoding.** Recovering (x, y) from $(h,\ s = x + hy)$ is decoding a random quasi-cyclic code – believed hard classically and quantumly.
- **The algebra mirrors ML-KEM.** KeyGen publishes a syndrome s; encapsulation builds $u = r_1 + hr_2$ and $v = mG + sr_2 + e$; decapsulation cancels the shared hyr_2 term so that $v - uy = mG + e'$ with e' low weight, and the public decoder of C removes e'.
- **Quasi-cyclic structure buys small keys.** A single ring element stands in for an entire circulant block, replacing McEliece's huge matrix with a few kilobytes – at the cost of a more structured assumption, the coding-theory analogue of the LWE-to-Ring-LWE step.
- **HQC is NIST's code-based backup.** Selected in March 2025 as an additional KEM, it complements the lattice-based ML-KEM with a scheme resting on an independent hard problem, so that a single advance against lattices would not break everything at once.

Part V

Multivariate and Isogeny-Based Cryptography

Chapter 24

Multivariate Cryptography

24.1 Learning objectives

The book has now surveyed lattices, hash functions, and codes. This chapter opens a fourth, again independent, hard-problem family. Multivariate cryptography abandons vectors, hashes, and codewords for a very different object: a system of nonlinear equations. Its security rests on a fact that predates cryptography itself – solving a system of *multivariate quadratic* equations over a finite field is, in general, computationally intractable, and no quantum algorithm is known to change that. The idea is old (it dates to the late 1980s and 1990s [42]) and it has a turbulent history: many concrete proposals have been broken, sometimes years after they were thought safe. That history is worth studying in its own right, because it teaches, more sharply than any other family, how a structured trapdoor can be reverse-engineered.

After finishing this chapter, you should be able to:

1. state the *multivariate quadratic* (MQ) problem – solving a system of quadratic equations over $\mathbb{F}_q$ – and explain why its NP-hardness (with no known quantum shortcut beyond Grover) is the security foundation of the family;
2. work through a tiny quadratic system over a small field and see why brute force costs q^n;
3. describe the shared trapdoor idea: a public system $P = S \circ F \circ T$ that looks random, built from an easily invertible *central map* F hidden between two secret linear maps S and T;
4. explain why multivariate schemes are almost always *signatures* – tiny signatures, very large public keys – rather than KEMs;
5. describe the *oil-and-vinegar* construction and its unbalanced variant UOV, understand why the layered variant Rainbow was a NIST finalist, and know that Rainbow was broken in 2022;
6. place multivariate cryptography as one of the five post-quantum families, and state honestly that – unlike the lattice and hash families – it has no FIPS standard as of this writing.

Why this chapter matters. Post-quantum security is a portfolio of independent bets, and multivariate cryptography is one of its more distinctive holdings. Where the lattice and code families deliver both encryption and signatures, the multivariate family is almost entirely a *signature* technology, and a specialized one: its signatures are among the smallest of any post-quantum scheme, at the cost of public keys measured in kilobytes to megabytes. It is also the family whose flagship

candidates have been broken most publicly, which makes it a case study in the central tension of trapdoor design – the very structure that lets the legitimate signer invert the system is the structure an attacker tries to detect. Understanding both the promise and the fragility rounds out the picture of what post-quantum cryptography is [6].

24.2 The hard problem: multivariate quadratics

Every scheme in this chapter reduces, for the attacker, to the same underlying task: given a system of quadratic equations, find a common solution.

Definition 24.1 (Multivariate quadratic (MQ) problem)**.** Let $\mathbb{F}_q$ be a finite field. Given m quadratic polynomials $p_1, \dots, p_m \in \mathbb{F}_q[x_1, \dots, x_n]$ and a target vector $y = (y_1, \dots, y_m) \in \mathbb{F}_q^m$, find, if it exists, a vector $x = (x_1, \dots, x_n) \in \mathbb{F}_q^n$ such that

$$p_k(x) = y_k \qquad \text{for all } k = 1, \dots, m.$$

Each polynomial has the form

$$p_k(x) = \sum_{1 \le i \le j \le n} a_{ij}^{(k)}\, x_i x_j \;+\; \sum_{i=1}^{n} b_i^{(k)}\, x_i \;+\; c^{(k)},$$

with all coefficients in $\mathbb{F}_q$.

A single quadratic equation is easy; a large system of them, sharing variables, is not. Deciding whether such a system has a solution is **NP-complete**, and this holds even over the smallest field $\mathbb{F}_2$ [42]. The intuition is that the quadratic terms couple the variables together so tightly that fixing some of them does not straightforwardly simplify the rest, and there is no linear-algebra shortcut analogous to Gaussian elimination.

The best classical attacks compute a **Gröbner basis** of the system (via algorithms such as F_4/F_5) or run the closely related **XL** linearization; both run, on random-looking instances, in time exponential in n once m and n are comparable. Quantum computers do not overturn this. As with every family in this part of the book, the strongest quantum approach applies Grover-style search and buys at most a square-root speedup, the same mild erosion described in the Quantum Threat chapter. There is no Shor-like collapse: MQ has no known efficient quantum algorithm, and doubling the parameters restores any margin a quantum search would erode.

Example 24.2 (A tiny system over $\mathbb{F}_2$)**.** Take $\mathbb{F}_2 = \{0, 1\}$, where addition is XOR and multiplication is AND, and note that $x_i^2 = x_i$ there, so the only genuine quadratic monomials are the products $x_i x_j$ with $i \neq j$. Consider two quadratic polynomials in three variables,

$$p_1(x) = x_1x_2 + x_1x_3 + x_2, \qquad p_2(x) = x_2x_3 + x_1 + x_3,$$

and the target $y = (1, 0)$. We seek $x \in \mathbb{F}_2^3$ with $p_1(x) = 1$ and $p_2(x) = 0$. With only $2^3 = 8$ candidates we can simply enumerate; doing so reveals three solutions, among them $x = (0, 1, 0)$:

$$p_1(0,1,0) = 0 + 0 + 1 = 1, \qquad p_2(0,1,0) = 0 + 0 + 0 = 0.$$

Brute force worked here only because the field and the dimension are tiny. In general an attacker faces q^n candidate vectors, and a real scheme chooses q and n so that q^n – and, more precisely, the cost of the Gröbner/XL attacks, which is far below q^n but still exponential – is astronomically large.

This is the security foundation. If we can hand an attacker a system of quadratics that looks random, then forging without the trapdoor means solving an MQ instance.

24.3 The trapdoor: a solvable system in disguise

The difficulty is the familiar one: if the public system were genuinely random, then *nobody* – not even the legitimate key holder – could solve it, and there would be no way to sign. Multivariate cryptography resolves this exactly as code-based cryptography does, by hiding an easy problem inside a hard-looking one.

The central map. The secret starts with a **central map** $F = (f_1, \ldots, f_m) : \mathbb{F}_q^n \to \mathbb{F}_q^m$, a system of quadratics chosen with special structure so that it can be *inverted efficiently*: given a target $a \in \mathbb{F}_q^m$, the key holder can quickly find some b with $F(b) = a$. The particular structure that makes F invertible is the design choice that distinguishes one multivariate scheme from another; the oil-and-vinegar structure of the next section is the most durable example.

The disguise. Left in the open, the structure of F would be plainly visible and an attacker could invert it too. So it is hidden between two secret invertible *affine* (for our purposes, linear) maps: an $n \times n$ map T applied to the input, and an $m \times m$ map S applied to the output. The public key is the composition

$$P = S \circ F \circ T,$$

expanded out and published as m explicit quadratic polynomials in n variables. Composing with the linear maps S and T keeps P quadratic, but it stirs the coefficients so thoroughly that the special shape of F disappears: to anyone without S, F, and T, the public system P looks like a random system of quadratics, and solving it looks like MQ.

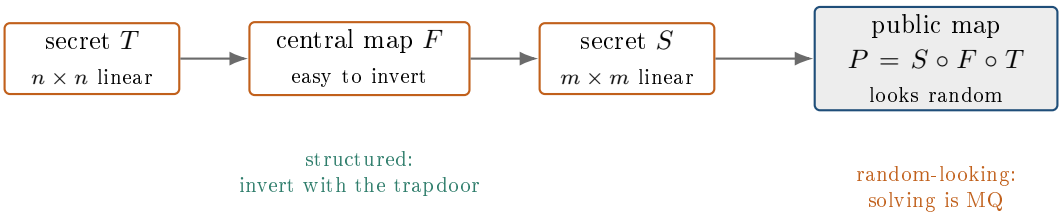


Figure 24.1: The trapdoor as a disguise. A structured central map F, which the key holder can invert efficiently, is sandwiched between two secret linear maps T and S. The published composition $P = S \circ F \circ T$ is an explicit system of quadratic polynomials that, to anyone without the secret factors, is indistinguishable from a random system. Everyone can *evaluate* P; only the trapdoor holder can *invert* it.

Signing, not encrypting. Notice which direction is easy. *Evaluating* P at a point is cheap and needs no secret – it is just plugging numbers into polynomials. *Inverting* P, finding an x with $P(x) = y$, is hard without the trapdoor. This asymmetry is exactly backwards from an encryption scheme, where anyone must be able to encrypt (compute the forward direction) and only the key holder decrypts. It fits a **signature** scheme perfectly: to sign a message M, hash it to a target $y = \mathcal{H}(M) \in \mathbb{F}_q^m$ and use the trapdoor to invert, producing a signature x with $P(x) = y$; to verify, anyone evaluates $P(x)$ and checks that it equals $\mathcal{H}(M)$. For this reason multivariate schemes are almost always signatures. Encryption variants exist but have fared poorly, and the signatures are where the family's strengths – very small signatures and fast verification – actually appear.

24.4 Oil and vinegar

The oldest central-map structure still standing is **oil and vinegar**, introduced by Patarin and analysed in its unbalanced form by Kipnis, Patarin, and Goubin in 1999 [43]. Its inversion trick is a small marvel of linear algebra.

Definition 24.3 (Oil-and-vinegar central map)**.** Partition the n variables into v *vinegar* variables $x_1, \dots, x_v$ and o *oil* variables $x_{v+1}, \dots, x_{v+o}$, with $n = v + o$. A quadratic polynomial f_k is of *oil-and-vinegar* type if every quadratic term contains at least one vinegar variable – equivalently, no two oil variables are ever multiplied together:

$$f_k(x) = \sum_{i=1}^{v} \sum_{j=1}^{n} \alpha_{ij}^{(k)}\, x_i x_j \;+\; \sum_{i=1}^{n} \beta_i^{(k)}\, x_i \;+\; \gamma^{(k)}.$$

The central map is $F = (f_1, \dots, f_o)$, with one polynomial per oil variable.

The oil variables are so named because, like oil in water, they never mix with one another – only ever with vinegar. That single restriction is what makes F invertible.

Proposition 24.4 (Inversion by linearization)**.** *Fix the vinegar variables to arbitrary field elements $x_1 = r_1, \dots, x_v = r_v$. Then each f_k, viewed as a function of the remaining oil variables alone, is* linear*: every surviving quadratic term had a vinegar factor, which is now a constant. Solving $F = a$ therefore reduces to a system of o linear equations in the o oil unknowns, which is solved by Gaussian elimination.*

So to invert F on a target a: pick random values for the vinegar variables, substitute them, and solve the resulting linear system for the oil variables. If the linear system happens to be singular, re-randomize the vinegar values and try again. This is fast, and it is exactly the trapdoor step F^{-1} used in signing.

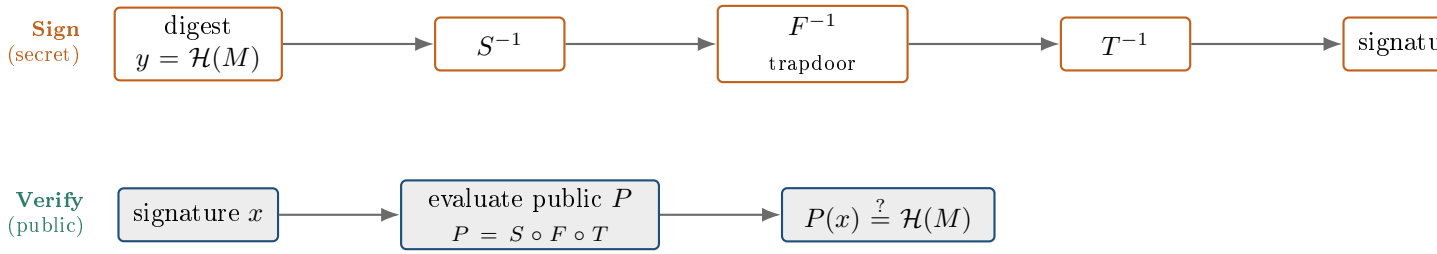


Figure 24.2: Sign and verify with a multivariate trapdoor. To *sign*, the digest $y = \mathcal{H}(M)$ is pushed backwards through the disguise – undo S, invert the structured central map F with the trapdoor, undo T – yielding a signature x with $P(x) = y$. To *verify*, anyone evaluates the public map P at x and checks the result against the digest. Only the middle step, F^{-1}, needs the secret; everything on the verify lane is public and cheap.

Remark. In pure single-layer UOV the outer map S can be omitted (a single layer's output can be re-mixed into the central map without weakening it), so many descriptions write the public key as $P = F \circ T$. We keep S explicit to match the general $P = S \circ F \circ T$ template, which is the right picture for the layered schemes below.

Balanced versus unbalanced. Patarin's original proposal took $v = o$. Kipnis and Shamir broke that balanced version, and the fix, due to Kipnis, Patarin, and Goubin, was to make the scheme *unbalanced* – take many more vinegar variables than oil, $v \gg o$ [43]. This is **Unbalanced Oil and Vinegar (UOV)**. The extra vinegar variables enlarge the hidden subspace an attacker must find and defeat the balanced attack. UOV, with conservative parameters, has resisted attack for over two decades and is the stable core of the whole family.

Algorithm 36 UOV KeyGen (teaching form)

Input: Field $\mathbb{F}_q$; parameters v (vinegar), o (oil), $n = v + o$, $m = o$
Output: Public key P; secret key (F, S, T)

choose an oil-and-vinegar central map $F = (f_1, \ldots, f_o)$ ▷ invertible by linearization
$T \leftarrow$ random invertible $n \times n$ linear map over $\mathbb{F}_q$ ▷ hides the input split
$S \leftarrow$ random invertible $m \times m$ linear map over $\mathbb{F}_q$ ▷ mixes the outputs
$P \leftarrow S \circ F \circ T$, expanded as m explicit quadratics ▷ looks random
return $\big(P,\ (F, S, T)\big)$

Algorithm 37 UOV Sign (teaching form)

Input: Message M; secret key (F, S, T)
Output: Signature $x \in \mathbb{F}_q^n$

$y \leftarrow \mathcal{H}(M) \in \mathbb{F}_q^m$ ▷ hash the message to a target
$a \leftarrow S^{-1}(y)$ ▷ undo the output mixing
pick random vinegar values; solve the resulting linear system for the oil variables to get b with $F(b) = a$ ▷ the trapdoor step; retry if singular
$x \leftarrow T^{-1}(b)$ ▷ undo the input map
return x

Algorithm 38 UOV Verify (teaching form)

Input: Message M; signature x; public key P
Output: Accept or reject

if $P(x) = \mathcal{H}(M)$ **then return** accept **else return** reject ▷ just evaluate the public quadratics

Rainbow: layers, and a fall. **Rainbow** stacks several oil-and-vinegar maps in layers, each layer's variables serving as vinegar for the next. Layering shrinks the keys and signatures relative to plain UOV, and on that strength Rainbow advanced to the *third round* of the NIST post-quantum project as one of three signature finalists. Then, in 2022, Beullens found a key-recovery attack that exploited the extra algebraic structure the layers introduced, recovering a Rainbow private key at the claimed lowest security level in about a weekend on a single laptop [44]. The break did not touch the MQ problem itself, nor plain UOV; it exploited Rainbow's *specific* layered structure, which turned out to leak more than its designers realized. Rainbow was withdrawn from standardization.

Where the family stands. The lesson the community drew was to retreat toward the more conservative, less structured UOV. As of this writing, several multivariate signatures – **UOV** itself, **MAYO** (a UOV variant with a smaller key, using a "whipping" technique to reuse a compact map), and **QR-UOV** (a quotient-ring variant that compresses the key) – are candidates in NIST's

additional-signatures "on-ramp" process, the round opened specifically to broaden the signature portfolio beyond the lattice and hash standards. None of them is a FIPS standard, and the process is ongoing; their standing may change.

24.5 Trade-offs and status

Multivariate signatures occupy a very particular corner of the design space, and Table 24.1 lays out the shape of the bargain. The signatures are extraordinarily small – often tens to a couple of hundred bytes, competitive with or smaller than pre-quantum ECDSA – and verification, being a batch of polynomial evaluations, is fast. The price is the public key, which stores the coefficients of m quadratics in n variables. That is roughly $m \cdot n^2/2$ field elements, growing quadratically with the dimension, so public keys run from tens of kilobytes to over a megabyte, and there is no seed that regenerates them, for the same reason as in the code-based case: a compressible public key would carry structure an attacker could exploit. Key-generation cost and the absence of a KEM round out the profile.

Property	Multivariate signatures (UOV family)
Primitive	signatures only (no practical KEM)
Signature size	very small (tens to a few hundred bytes)
Public key size	very large (tens of KB to $> 1\,\mathrm{MB}$)
Verification	fast (evaluate public quadratics)
Signing	moderate (invert the central map)
Hardness assumption	MQ: solving random multivariate quadratics

Table 24.1: The multivariate signature bargain. Tiny signatures and fast verification are bought with a very large, incompressible public key, and the family offers signatures but no practical key-encapsulation mechanism. This is close to the mirror image of the hash-based signatures of the earlier chapter, which have tiny public keys but large signatures.

Status. No multivariate scheme is among the FIPS standards (ML-KEM, ML-DSA, SLH-DSA, FN-DSA); those are drawn from the lattice and hash families. Rainbow, the family's most prominent standardization candidate, was broken in 2022 and withdrawn. The conservative core, UOV, together with the more compact variants MAYO and QR-UOV, is under evaluation in NIST's additional-signatures round as of this writing, with the outcome not yet settled. The practical appeal, if these schemes are standardized, is a niche but real one: settings where signatures are transmitted or stored constantly but the public key is distributed rarely, so that a small signature matters far more than a large key.

24.6 Pitfalls

Pitfall 1: assuming a structured trapdoor is safe because MQ is hard. This is the defining hazard of the family, and Rainbow is its cautionary tale. The public key is *not* a random MQ instance – it is $S \circ F \circ T$, and the hidden structure of F can leak through the linear disguise. Attacks like Kipnis–Shamir on balanced oil-and-vinegar and Beullens on Rainbow [44] never solved MQ in general; they detected and peeled away the specific structure of the central map. The more structure a variant adds to shrink its keys, the more an attacker has to grip. Security rests

on the disguised system being indistinguishable from random, and that must be argued for each construction, not inherited from the NP-hardness of MQ.

Pitfall 2: forgetting that the public key is large and incompressible. Unlike the lattice standards, whose public matrix expands from a 32-byte seed, a multivariate public key is a dense list of quadratic coefficients with no shorter description. Budgeting for it as if it were a few kilobytes, or planning to ship it inside every handshake, will not work. The family suits scenarios where the key is installed once and reused, not renegotiated per session.

Pitfall 3: using a balanced or under-parameterized oil-and-vinegar split. Plain balanced oil and vinegar ($v = o$) is broken. Security requires an *unbalanced* split with substantially more vinegar than oil, and the exact ratio and field size are load-bearing parameters, not free choices. Under-sizing them re-opens the very attacks the unbalanced design was introduced to close.

Pitfall 4: expecting a KEM. The trapdoor is easy to evaluate and hard to invert, which is the right shape for signing and the wrong shape for public-key encryption. Multivariate encryption schemes have a poor security track record; if you need post-quantum key establishment, reach for a lattice or code-based KEM, not this family.

24.7 Summary

Multivariate cryptography is the fourth of the post-quantum families this book surveys, and it rests on the **MQ problem**: solving a system of multivariate quadratic equations over a finite field, an NP-hard task with no known quantum shortcut beyond Grover's mild square-root erosion [42]. It is turned into a signature scheme by the now-familiar trapdoor manoeuvre:

- start from a **central map** F with special structure – classically the **oil-and-vinegar** form – that can be inverted by linearization;
- **disguise** it between two secret linear maps and publish $P = S \circ F \circ T$, an explicit system of quadratics that looks random;
- anyone can **verify** by evaluating P, but only the trapdoor holder can **sign** by inverting it;
- accept a **very large public key** in exchange for an unusually **small signature** and fast verification – and no KEM.

The family's history is a lesson in the fragility of structured trapdoors: the balanced oil-and-vinegar scheme fell, and the layered Rainbow, a NIST third-round finalist, was broken by Beullens in 2022 [44], both by attacks that exploited structure rather than by any progress against MQ itself. The response has been a retreat to the conservative core, UOV, and its compact relatives MAYO and QR-UOV, which are under evaluation in NIST's additional-signatures round as of this writing.

Placed among the five post-quantum families – lattices, codes, hashes, multivariate systems, and isogenies – multivariate cryptography is the specialist of the group: signature-only, smallest-in-class signatures, largest-in-class keys, and a hard-won reputation for caution after some public failures. Crucially, and unlike the lattice and hash families that supply today's FIPS standards, it has *no* standardized scheme yet. It remains a live and valuable hedge – an independent hard problem held in reserve – but one still proving, candidate by candidate, that its trapdoors can withstand the scrutiny that broke its predecessors.

Chapter 25

Isogeny-Based Cryptography

25.1 Learning objectives

The families covered so far – lattices, hashes, and codes – account for every scheme NIST has standardized, but they do not exhaust the landscape of post-quantum cryptography. The standard taxonomy names five broad families [6], and this chapter surveys the youngest and most mathematically demanding of them: **isogeny-based** cryptography, whose hardness comes not from decoding or short vectors but from navigating a vast graph of elliptic curves. It is the family with the smallest keys and signatures of any post-quantum approach, and also the one that produced the field's most dramatic cautionary tale – a flagship scheme that reached the final round of the NIST competition and was then broken outright in an afternoon. Both facts are worth understanding, and they are connected.

This chapter is a survey, not an implementation guide. The mathematics of isogenies runs deep, and we deliberately trade rigour for intuition: the aim is to convey what the objects are, why the underlying problem is believed hard, how the schemes use it, and what the 2022 break did and did not destroy.

After finishing this chapter, you should be able to:

1. recall what an elliptic curve over a finite field is, and describe an *isogeny* as a structure-preserving map between two such curves;
2. explain what makes a curve *supersingular* and picture the **supersingular isogeny graph**, whose rapid mixing is the source of cryptographic hardness;
3. state the **supersingular isogeny path problem** and explain why it is believed hard for classical and quantum computers alike, and why it yields the smallest parameters of any post-quantum family;
4. describe SIDH/SIKE as a Diffie–Hellman-style key exchange on the graph, and explain how the *Castryck–Decru attack* recovered its secret key in polynomial time by exploiting the auxiliary torsion-point information the protocol reveals;
5. place what survives the break – CSIDH and SQIsign – and explain why isogeny signatures are the smallest known, even though no isogeny scheme is yet a FIPS standard.

Why this chapter matters. The value of a portfolio of post-quantum families is that they fail independently: a breakthrough against lattices would say nothing about codes or isogenies. Isogeny-based cryptography is the most extreme point in that portfolio – the smallest bandwidth by a wide

margin, resting on a problem with no known subexponential attack, but built on machinery only a couple of decades old and probed far less than lattices or codes. The story of SIDH is the sharpest illustration in this book of a general principle: a hard problem can be quietly undermined not by attacking it head-on, but by the extra information a protocol hands the attacker for free. That lesson outlives the scheme that taught it.

25.2 A crash course in elliptic curves and isogenies

The reader has met finite fields and polynomial rings, but not the geometry this family is built on. This section introduces just enough of it. Everything lives over a finite field $\mathbb{F}_q$, and for the cryptographically interesting case $q = p^2$ for a large prime p.

Elliptic curves. An elliptic curve is a particular kind of cubic equation whose solutions can be *added* to one another, turning a geometric object into an algebraic group.

Definition 25.1 (Elliptic curve over a finite field)**.** Let $\mathbb{F}_q$ be a finite field of characteristic greater than 3. An **elliptic curve** E over $\mathbb{F}_q$ is the set of solutions $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ to an equation

$$y^2 = x^3 + ax + b, \qquad a, b \in \mathbb{F}_q, \quad 4a^3 + 27b^2 \neq 0,$$

together with a distinguished *point at infinity* $\mathcal{O}$. These points form a finite **abelian group** under a geometric "chord-and-tangent" addition law, with $\mathcal{O}$ as the identity.

Two facts about this group matter for us. First, its size is finite and close to q; the group is where classical elliptic-curve cryptography (ECC) hides its secrets, in the difficulty of the discrete logarithm. That difficulty is exactly what Shor's algorithm destroys, so ECC is *not* post-quantum. Isogeny-based cryptography reuses the curves but abandons the discrete logarithm entirely, hiding its secrets somewhere Shor cannot reach: in the maps *between* curves.

Second, each curve carries a single number, its **j-invariant** $j(E) \in \mathbb{F}_q$, computed from a and b. Two curves are isomorphic exactly when their j-invariants agree, so we may think of $j(E)$ as a fingerprint that names the curve up to relabelling. The schemes in this chapter transmit j-invariants, not equations.

Isogenies. An isogeny is a map from one elliptic curve to another that respects both the geometry and the group structure – the natural notion of "morphism" for these objects.

Definition 25.2 (Isogeny)**.** An **isogeny** $\varphi : E_1 \to E_2$ between two elliptic curves over $\mathbb{F}_q$ is a non-constant rational map that is also a group homomorphism, so $\varphi(\mathcal{O}) = \mathcal{O}$ and $\varphi(P + Q) = \varphi(P) + \varphi(Q)$. Its **kernel** is the finite subgroup of points sent to $\mathcal{O}$, and (for the separable isogenies we use) its **degree** is the size of that kernel.

The single most useful fact about isogenies is that they are pinned down by their kernels. Given any finite subgroup $G \subseteq E_1$, there is an isogeny $\varphi : E_1 \to E_2$ with kernel exactly G, unique up to isomorphism of the target, and it can be computed efficiently from G by **Vélu's formulas**. So "choosing an isogeny out of E_1" and "choosing a finite subgroup of E_1" are the same act. A secret isogeny is really a secret subgroup, and to walk a long isogeny is to compose many small ones.

Two more properties make isogenies behave like the arrows of a well-connected network. Degrees multiply under composition, so a chain of ℓ-degree steps is an isogeny of degree ℓ^e. And every isogeny φ of degree d has a **dual** isogeny $\hat{\varphi}$ of the same degree running the other way, with $\hat{\varphi} \circ \varphi = [d]$ (multiplication by d). Isogenies of a fixed small degree therefore come in matched pairs, which is what lets us regard them as undirected edges of a graph.

25.3 The supersingular isogeny graph

Among all elliptic curves over $\mathbb{F}_{p^2}$, a special and comparatively rare class – the **supersingular** curves – provides the setting for cryptography. The precise definition (a curve whose ring of self-maps, its *endomorphism ring*, is unusually large – an order in a quaternion algebra rather than in an imaginary quadratic field) is beyond a survey. What matters is the consequence.

A small, tightly connected world. There are only about $p/12$ supersingular j-invariants over $\mathbb{F}_{p^2}$ – a finite set, but astronomically large for a cryptographic p of several hundred bits. Fix a small prime ℓ (in practice $\ell = 2$ or $\ell = 3$). Build a graph:

- the **vertices** are the supersingular curves over $\mathbb{F}_{p^2}$, each named by its j-invariant;
- the **edges** are the isogenies of degree ℓ between them; because every ℓ-isogeny has a dual of degree ℓ, edges are undirected.

Each vertex has exactly $\ell+1$ neighbours, so this **supersingular ℓ-isogeny graph** is $(\ell+1)$-regular. It is also connected: from any supersingular curve you can reach any other by a sequence of ℓ-isogeny steps.

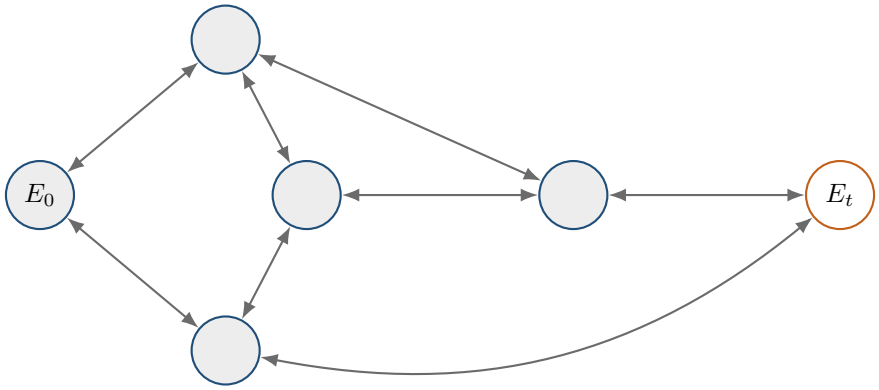


Figure 25.1: A fragment of a supersingular isogeny graph. Vertices are supersingular curves (named by j-invariant); each undirected edge is an ℓ-isogeny, matched with its dual. Every vertex has the same number of neighbours, and the graph is a rapidly mixing *expander*: a short random walk from a start curve E_0 lands on an essentially unpredictable curve. The hard problem is to recover a *path* from E_0 to a given target E_t, when only the two endpoints are known.

Rapid mixing. The property that makes the graph cryptographically useful is that it is an optimal **expander** – in fact a *Ramanujan* graph, meeting the theoretical limit on how well-connected a regular graph can be. In plain terms, a random walk mixes extremely fast: after only about $\log p$ steps, the endpoint of a random path is distributed almost uniformly over all $\sim p/12$ vertices, with no detectable bias toward where it started. A secret path of a few hundred small steps therefore lands on a curve that reveals nothing about the route taken. This is the isogeny analogue of a one-way function: composing the steps forward is cheap, but the endpoint alone does not betray the path.

25.4 The hard problem: finding an isogeny path

The security of the entire family rests on the belief that the walk cannot be run backwards – that seeing only the two endpoints of a secret path gives no efficient way to reconstruct it.

Definition 25.3 (Supersingular isogeny path problem)**.** Given two supersingular elliptic curves E_0 and E_t over $\mathbb{F}_{p^2}$, known to be connected by a path in the ℓ-isogeny graph, **find** an isogeny $\varphi : E_0 \to E_t$ – equivalently, a sequence of ℓ-isogeny steps joining E_0 to E_t.

Because the graph is an expander, the two endpoints look like an unrelated pair of curves; nothing local about E_t points back toward E_0. The best known classical algorithms are generic path-finding (meet-in-the-middle) searches that cost on the order of $\sqrt{p}$ work, and the best known quantum algorithms – claw-finding built on Grover-style search – cost on the order of $p^{1/4}$. Both are fully *exponential* in the bit-length of p: there is no known subexponential attack, classical or quantum, and in particular nothing resembling Shor's collapse of factoring and discrete logarithms. The problem is closely tied to the difficulty of computing the endomorphism ring of a supersingular curve, which is believed equally hard.

This exponential hardness with tiny objects is the family's headline advantage. Because attacks scale as $\sqrt{p}$ or $p^{1/4}$ rather than as a mild erosion of a large structured key, a few hundred bits of p already buy a comfortable security margin. The transmitted data are j-invariants and a handful of curve coordinates – a few dozen to a few hundred bytes. No other post-quantum family comes close on bandwidth. The catch, developed over the rest of the chapter, is that a *protocol* may have to reveal more than the two bare endpoints, and that "more" is where the danger lives.

25.5 SIDH and SIKE: a Diffie–Hellman on the graph

The first practical isogeny scheme was **SIDH** – Supersingular Isogeny Diffie–Hellman – proposed by Jao and De Feo in 2011 [45]. Its shape mirrors classical Diffie–Hellman exactly, with secret *isogenies* playing the role of secret exponents.

The commutative square. Fix a public starting curve E_0 over $\mathbb{F}_{p^2}$, where p is chosen so that E_0 has plentiful torsion of two coprime orders – typically powers of 2 and of 3. The two parties walk in the two different graphs:

- **Alice** picks a secret subgroup of 2-power order and computes her secret isogeny $\varphi_A : E_0 \to E_A$;
- **Bob** picks a secret subgroup of 3-power order and computes his secret isogeny $\varphi_B : E_0 \to E_B$.

Each publishes the endpoint curve (E_A, E_B) of their own walk. If each could now re-apply their own secret isogeny starting from the other's curve, the two routes would commute and meet at a single shared curve E_{AB}, whose j-invariant is the shared secret. This is the commutative square of Figure 25.2.

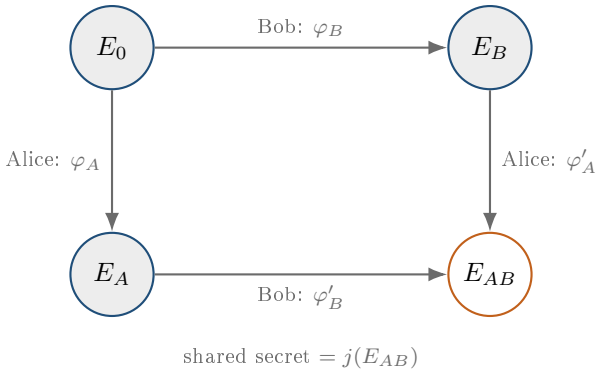


Figure 25.2: The SIDH commutative square. Alice and Bob walk secret isogenies of coprime degree out of the public curve E_0, publishing only the endpoints E_A and E_B. To close the square each must re-apply their own secret isogeny *starting from the other party's curve*, which requires knowing where their private torsion points land under the partner's map. Publishing those torsion-point images is what makes the protocol work – and, as the next section shows, what breaks it.

The auxiliary torsion points. Here lies the subtle ingredient. To let Bob re-apply Alice's isogeny from his curve, Alice must publish not only E_A but also the *images* under φ_A of a fixed public basis of Bob's torsion subgroup – two extra points on E_A. Bob does the symmetric thing. These **auxiliary torsion-point images** are what allow each party to reconstruct the partner's secret subgroup translated onto their own curve, and so to finish the square. They are essential to the protocol. They are also strictly more information than the bare path problem gives an attacker: alongside the endpoints E_0 and E_A, the adversary is handed the action of the secret isogeny on a known set of points.

SIKE. Wrapping SIDH in a Fujisaki–Okamoto-style transform yields **SIKE** (Supersingular Isogeny Key Encapsulation), a chosen-ciphertext-secure KEM. SIKE was a serious contender in the NIST competition: it advanced to the fourth round as one of the few remaining key-encapsulation candidates, prized above all for its exceptionally small keys and ciphertexts – on the order of a few hundred bytes, several times smaller than any lattice or code alternative. For a decade it was the poster child for how little bandwidth post-quantum security might cost.

25.6 The 2022 break

In the summer of 2022, SIDH and SIKE were broken. Castryck and Decru gave an **efficient key-recovery attack** [46]: given a party's public key, it reconstructs the secret isogeny in *polynomial time*. For the actual SIKE parameters the attack runs in minutes to about an hour on a single ordinary computer – not a marginal weakening of the security level, but a total collapse. Independent refinements by others followed within weeks, generalizing and speeding up the result.

What the attack exploited. Crucially, the attack did *not* solve the supersingular isogeny path problem. It targeted the extra information SIDH reveals: the auxiliary torsion-point images. The core idea uses a theorem of Kani to relate Alice's secret isogeny to an isogeny between *higher-dimensional* abelian varieties (products of elliptic curves), and the known torsion-point images provide exactly the data needed to build and evaluate that higher-dimensional object step by step. Each step tests and extends a guess about the secret walk, and because the auxiliary points pin

the walk down so tightly, the search collapses from exponential to polynomial. The starting curve's special structure, whose endomorphisms are known, made the construction concrete.

Remark. The distinction is worth stating sharply. The abstract problem in the definition above – recover an isogeny from two bare curves – was *not* weakened by the attack and is still believed hard. What fell was SIDH, because SIDH does not present an attacker with two bare curves. It presents them with two curves *plus* the images of a known torsion basis, and that auxiliary information was enough to bypass the hard problem entirely. The hardness assumption survived; the scheme's way of using it did not.

The lesson. This is the sharpest example in the book of a recurring hazard: side information volunteered by a protocol can undermine a hard problem without touching the problem itself. SIDH's designers knew the torsion images were extra data, but for eleven years no one saw how to weaponize them; the mathematics that turned them into a break – higher-dimensional isogenies – simply had not been connected to the setting yet. It is a reminder that "the underlying problem is hard" is a necessary but not sufficient condition for a scheme's security, and that young assumptions can hide such connections for a long time.

25.7 What survives: CSIDH and SQIsign

The break was specific to SIDH's torsion-point construction. Isogeny schemes that do *not* publish the action of a secret isogeny on known points were untouched, and the family remains active. Two lines are worth naming.

CSIDH. **CSIDH** ("commutative SIDH", pronounced "sea-side") is a key exchange built on a different structure: a commutative *group action* of an ideal-class group on a set of supersingular curves defined over the prime field $\mathbb{F}_p$. Because the shared secret is derived purely from composing group-action elements, CSIDH transmits only curves – no auxiliary torsion-point images – so the Castryck–Decru attack does not apply to it. Its distinctive feature is that the key exchange is *non-interactive* and commutative in the way classical Diffie–Hellman is, which SIDH never quite was. The trade-off is that a commutative group action is vulnerable to a subexponential *quantum* attack (Kuperberg's hidden-shift algorithm), so CSIDH must use larger parameters than its exponential-hardness cousins, and it is computationally slow. Its exact security level remains debated.

SQIsign. The most prominent survivor is **SQIsign** ("Short Quaternion and Isogeny Signature"), introduced by De Feo, Kohel, Leroux, Petit, and Wesolowski in 2020 [47]. It is a signature scheme, built from the correspondence between supersingular curves and quaternion orders (the Deuring correspondence), and it rests on the hardness of the endomorphism-ring / isogeny-path problem rather than on any torsion-point trick. Its headline property is size: SQIsign produces the **smallest signatures and public keys of any post-quantum signature scheme by a wide margin** – a signature of a couple of hundred bytes and a public key of a few dozen, where lattice signatures run to a kilobyte or more and hash-based signatures to many kilobytes. The price is speed: signing and verification are far slower than lattice schemes, historically on the order of seconds, though later variants have cut this substantially. As of writing, SQIsign is a candidate in NIST's additional-signatures round (the "on-ramp" for signatures beyond the lattice and hash standards), where its tiny footprint is the main attraction and its performance and youth are the main concerns.

25.8 Trade-offs and status

Isogeny-based cryptography occupies a clear and unusual corner of the design space.

Scheme (NIST level 1)	Public key	Signature	Family
SQIsign	≈ 64 B	≈ 177 B	isogenies
FN-DSA (Falcon-512)	897 B	≈ 666 B	lattices
ML-DSA-44	1312 B	2420 B	lattices
SLH-DSA-128s	32 B	7856 B	hashes

Table 25.1: Approximate signature footprints at the lowest NIST security level. SQIsign's combined public-key-plus-signature size is the smallest of any post-quantum signature by a wide margin; the cost, not shown here, is far slower signing and verification and a much younger, less-scrutinized assumption. Figures are indicative and vary across parameter sets and revisions.

Remark. The isogeny bet is the mirror image of Classic McEliece's. Where McEliece sells four decades of conservatism at the cost of a megabyte-scale public key, isogenies sell the smallest bandwidth of any family at the cost of youth, speed, and a demonstrated capacity to surprise. The SIKE break is not a reason to discard the family – the path problem it never solved is still standing – but it is a standing warning that the assumptions here have been probed for years, not decades, and that the ground can still move.

Status. No isogeny-based scheme is a FIPS standard. SIKE, once a NIST fourth-round KEM candidate, was withdrawn after the 2022 break. SQIsign is under evaluation in the additional-signatures round as of writing, valued for its unmatched compactness but held back by performance and by the relative youth of its security foundations; whether it advances to standardization is undecided. CSIDH remains a research-grade proposal with contested parameters. The honest summary is that isogeny-based cryptography is the most promising *small*-footprint post-quantum family and simultaneously the least mature – a live research area rather than a settled toolbox.

25.9 Pitfalls

Pitfall 1: assuming a hard underlying problem makes a scheme safe. This is the SIKE lesson, and it is the most important idea in the chapter. The supersingular isogeny path problem was never broken; SIDH was broken because it revealed *more* than that problem – the images of known torsion points under the secret isogeny – and that surplus was enough. A security proof is only as strong as the assumption it actually reduces to, and a protocol that leaks auxiliary structure may be reducing to a much easier problem than its designers intended.

Pitfall 2: treating "smallest keys" as "best." Isogeny schemes win decisively on bandwidth, and it is tempting to read that as an overall verdict. It is not. They are slow, their assumptions are young, and one flagship has already fallen. Size is one axis among several; conservatism and speed pull the other way, and for most deployments a lattice scheme's balance is the safer default today.

Pitfall 3: thinking the 2022 break killed the whole family. The Castryck–Decru attack was specific to the torsion-point construction of SIDH and SIKE. Schemes that do not publish such

images – CSIDH, SQIsign – are not affected by it, and the family remains an active standardization candidate through SQIsign. "SIDH is broken" and "isogenies are broken" are very different statements; only the first is true.

Pitfall 4: forgetting how young the field is. Lattices and codes have been attacked for decades; the supersingular-isogeny assumptions have been under concentrated cryptanalysis for a far shorter time, and the mathematics involved – endomorphism rings, higher-dimensional isogenies, the Deuring correspondence – is deep enough to hide further connections. Youth is not a flaw, but it is a reason to demand hybrid deployment and to avoid betting long-lived secrets on isogenies alone.

25.10 Summary

Isogeny-based cryptography is the fifth and youngest of the post-quantum families, and the one that departs furthest from the rest of this book:

- its objects are **elliptic curves** over a finite field and the **isogenies** – structure-preserving maps – between them, organized into a **supersingular isogeny graph** that is a rapidly mixing expander;
- its hard problem is the **supersingular isogeny path problem** – recover a secret path from its two endpoint curves – which has no known subexponential attack, classical or quantum, and so yields the **smallest keys and signatures** of any family;
- **SIDH/SIKE** [45] turned this into a Diffie–Hellman-style key exchange, reached NIST's fourth round, and was then **broken in polynomial time in 2022** [46] – not by solving the path problem, but by exploiting the auxiliary torsion-point images the protocol reveals;
- what survives – **CSIDH** for key exchange and **SQIsign** [47] for the smallest signatures known – avoids that leak, and SQIsign is a live candidate in NIST's additional-signatures round, though **no isogeny scheme is yet a FIPS standard**.

The enduring lesson is not that isogenies failed – the assumption at their core is intact – but that a hard problem can be undermined by the information a protocol volunteers around it. That, together with the family's unmatched compactness and its unmistakable youth, is what to remember. Isogeny-based cryptography is the high-risk, high-reward corner of the post-quantum portfolio: the smallest footprint on offer, resting on the least-settled ground.

Part VI

Post-Quantum Cryptography in Practice

Chapter 26

Side-Channel Security

26.1 Learning objectives

Every earlier chapter of this book argued about security *on paper*: a lattice problem is hard, a forgery reduces to Module-SIS, a distinguisher would break LWE. That kind of security is necessary. It is also not sufficient. A scheme does not run on paper; it runs on a physical device that takes time, draws current, radiates, and can be poked with a laser. If any of those measurable, physical quantities depends on the secret key, an attacker can read the key off the machine without ever touching the mathematics. This chapter's warning is blunt: *the math can be perfect and the deployment still broken.* It opens the final part of the book, on practice: correct, side-channel-safe implementation is one requirement among several for getting these schemes into real systems, alongside the protocol integration and migration that the chapters after it take up.

After finishing this chapter, you should be able to:

1. explain why provable, mathematical security is necessary but not sufficient, and say precisely what a *side channel* is;
2. name the main families of implementation attacks – timing, cache, power, electromagnetic, and fault – and state what each one actually observes;
3. state the rules of *constant-time* programming and write a constant-time conditional select and a constant-time comparison;
4. connect these ideas to concrete PQC design choices already met in this book: CBD instead of Gaussian sampling (15), Falcon's floating-point sampler (18), ML-DSA's deterministic abort loop (17), and the implicit rejection built into ML-KEM decapsulation (16);
5. describe the higher-order countermeasures – masking and blinding – and weigh their cost against constant-time coding.

Why this chapter matters. Shor's algorithm, the threat that motivated the whole book, attacks the *structure* of the problem: it factors, it computes discrete logs. The schemes we built defeat it because no efficient quantum algorithm is known for the lattice and code problems underneath. A side-channel attack does something completely different. It ignores the problem structure and instead measures the execution: how long a decryption took, which cache lines it touched, how much power the chip drew on each clock cycle. These attacks are often far *cheaper* than the mathematical ones – minutes with an oscilloscope rather than aeons on a quantum computer – which is exactly

why they matter. The concern is not hypothetical for PQC: several of the schemes in this book have delicate implementation requirements, and those requirements shaped the standards. It is why ML-KEM samples noise with CBD rather than a Gaussian (15), and why FN-DSA (based on Falcon; FIPS 206 remains in development) is especially delicate to deploy safely (18).

26.2 The attack families

A side channel is any observable output of the computation other than the intended result. Below are the families a PQC implementer must keep in mind. Each is sketched in one paragraph; the common thread is that all of them turn a physical measurement into information about the secret.

Timing attacks. The total time a computation takes depends on the data it processed [48]. If that data includes the secret, the running time leaks it. The textbook example is a comparison that returns as soon as it finds the first differing byte: an attacker who can vary an input and time the response learns the secret one byte at a time. Timing leaks are the most dangerous class because they need no special equipment and can often be measured *remotely*, over a network.

Cache attacks. Modern CPUs cache recently used memory. If a program indexes a table at a secret-dependent position – as many Gaussian samplers and cipher S-boxes do – then which table entries land in the cache depends on the secret. An attacker sharing the same machine (a co-resident cloud tenant, say) does not see the values, but can time its *own* memory accesses to learn which cache lines were touched, using techniques such as Flush+Reload or Prime+Probe. Secret-dependent memory addresses are therefore just as leaky as secret-dependent timing.

Power analysis. The instantaneous power a chip consumes correlates with the operations it performs and the bits it moves. *Simple power analysis* (SPA) reads a single trace to recognize the sequence of operations – for instance, telling a multiply from a square in a modular exponentiation, which spells out exponent bits. *Differential power analysis* (DPA) [49] is stronger: it collects many traces and correlates them, sample by sample, against a hypothesis about a few key bits, pulling the secret out even when it sits far below the measurement noise.

Electromagnetic analysis. A switching transistor radiates. The same information that power analysis reads from the supply line can be read from the electromagnetic emanations near the chip, often with a small probe and no electrical contact. EM analysis can also be spatially localized to a single region of the die, which sometimes defeats countermeasures that only balance the *global* power draw.

Fault injection. The attacks above are passive: they observe. Fault injection is active. By glitching the supply voltage or clock, or firing a laser or EM pulse at the die, an attacker induces a controlled error – a skipped instruction, a flipped bit, a corrupted value – and exploits the faulty output. Signature schemes are especially fragile here: a single faulty signature can, through differential fault analysis, reveal the signing key, and skipping a bounds check can leak the mask in a scheme like ML-DSA.

26.3 Constant-time programming

The central software defense against secret-dependent *timing and cache* leakage is **constant-time programming**. It is necessary but not sufficient for physical power or EM attacks. The name is slightly misleading: the goal is not that the code always takes the same wall-clock time, but that its timing, memory-access pattern, and operation sequence *do not depend on secret data*. Variation that depends only on public data (the length of a message, the public iteration count of a loop) is fine.

The discipline reduces to three rules:

- **No secret-dependent branches.** An `if` whose condition involves a secret takes one path or another, and the two paths almost never cost the same. Replace the branch with straight-line arithmetic.
- **No secret-dependent memory indices.** Never use a secret as an array index or a jump target. A table lookup at a secret address leaks through the cache even if the code has no branches at all.
- **Avoid variable-time instructions on secret data.** Division and modulo, and on some processors even multiplication, take a data-dependent number of cycles. Early-exit library routines such as `memcmp` are the same hazard in disguise. Use fixed-cost operations, or reduce with Montgomery/Barrett arithmetic instead of a hardware divide.

Figure 26.1 contrasts the hazard with its fix. On the left, a secret bit chooses between a short path and a long path; the difference in duration is the leak. On the right, the same choice is computed with a mask so that every input runs the identical sequence of operations.

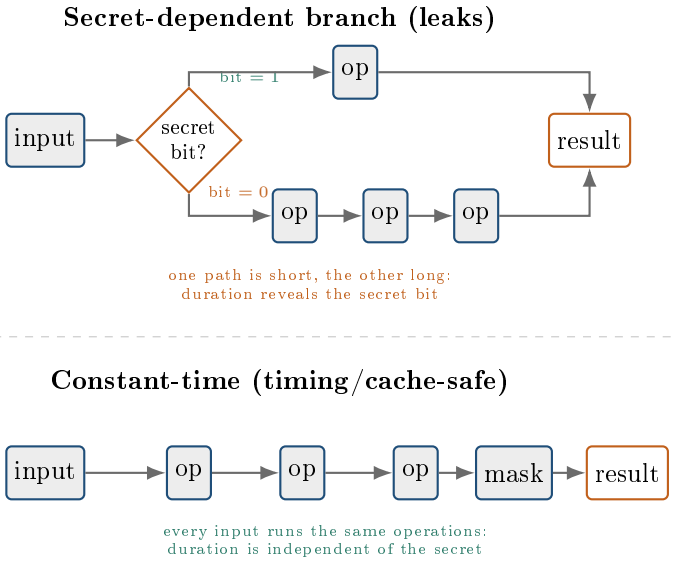


Figure 26.1: Top: a branch on a secret bit takes a short or a long path, so the running time and cache trace can carry the secret. Bottom: the same conditional is computed as one fixed sequence of operations ending in a masked select. This removes secret-dependent control flow and memory behaviour, but intermediate values can still correlate with physical power or EM leakage.

The workhorse that removes a secret branch is the *constant-time select*. It turns the selector bit into an all-ones or all-zero *mask* and uses that mask to blend the two candidate values with bitwise arithmetic, so both candidates are always computed and no branch is taken.

Algorithm 39 CTSELECT – constant-time conditional select

Input: Selector bit $s \in \{0, 1\}$; machine words a and b
Output: a if $s = 1$, else b – with no data-dependent branch

1: $mask \leftarrow -s$ ▷ two's complement: all-ones word if $s = 1$, all-zero if $s = 0$
2: **return** $(a \,\&\, mask) \mid (b \,\&\, \overline{mask})$ ▷ $\overline{mask}$ is the bitwise complement of $mask$

The trick is the mask $-s$: negating the bit 1 in two's complement gives the all-ones word, and negating 0 gives the all-zero word. When $s = 1$ the first term passes a and the second term is zeroed; when $s = 0$ the roles swap. There is no branch and no secret-dependent address. The same masking idea gives a constant-time *equality* test, CT_EQ, which must never return early: it accumulates the differences across *all* bytes and only then collapses the result to a single bit.

Listing 26.1 shows both patterns in C: the leaky branch, its masked replacement, and a no-early-exit comparison.

Listing 26.1: Leaky versus constant-time patterns in C.

```c
#include <stdint.h>
#include <stddef.h>

/* LEAKY: the branch condition is a secret; the two paths differ */
uint32_t leaky_select(int secret_bit, uint32_t a, uint32_t b) {
    if (secret_bit)      /* timing/cache/power all reveal which way */
        return a;
    else
        return b;
}

/* CONSTANT-TIME: no branch, no secret-dependent memory access */
uint32_t ct_select(uint32_t secret_bit, uint32_t a, uint32_t b) {
    uint32_t mask = -secret_bit;   /* 0xFFFFFFFF if 1, 0x00000000 if 0 */
    return (a & mask) | (b & ~mask);
}

/* CONSTANT-TIME equality: 1 if equal, else 0, never exits early */
uint32_t ct_eq(const uint8_t *x, const uint8_t *y, size_t n) {
    uint8_t diff = 0;
    for (size_t i = 0; i < n; i++)
        diff |= x[i] ^ y[i];       /* fold every byte, no break */
    /* top bit of (diff | -diff) is 1 iff diff != 0 */
    uint32_t nz = ((uint32_t)diff | (0u - (uint32_t)diff)) >> 31;
    return nz ^ 1u;                /* invert: equal -> 1 */
}
```

26.4 PQC-specific concerns

Constant-time discipline is general, but post-quantum schemes concentrate the risk in a few specific places. Each of the following ties back to a design choice made in an earlier chapter.

Sampling: the reason CBD exists. The noise in a lattice scheme must be sampled every time a key or ciphertext is made. A discrete *Gaussian* sampler is the natural mathematical choice, but it is a side-channel minefield: it typically needs rejection loops (a secret-dependent number of iterations), table lookups (secret-dependent memory addresses), and floating-point arithmetic (variable-time on most hardware). ML-KEM samples its small noise polynomials from the *centered binomial distribution* of Chapter 15. ML-DSA uses its own bounded-uniform and rejection-sampling procedures for secrets and masks; it is not an instance of the ML-KEM CBD sampler. Both standards avoid a general-purpose discrete-Gaussian sampler, but their sampling interfaces and side-channel reviews must be considered separately. FN-DSA (18) is the cautionary opposite. Its trapdoor sampler genuinely needs a high-precision Gaussian over the reals, computed with floating-point FFT, and making *that* constant-time is the hardest implementation problem among all the standards – if the sampler's timing or rounding leaks, the secret basis can be reconstructed. That difficulty, forward-referenced at the end of Chapter 18, is this chapter: it is the single strongest reason Falcon is the most demanding of the three signature standards to deploy safely.

Rejection sampling in ML-DSA. ML-DSA signing (17) also loops: it draws a mask, forms $\mathbf{z} = \mathbf{y} + c\,\mathbf{s}_1$, and applies rejection conditions involving secret-dependent quantities, including checks related to $\mathbf{s}_2$ and $\mathbf{t}_0$. The retry count and signing time therefore must not be declared harmless merely from distributional intuition. Implementations must follow FIPS 204's sampling and side-channel design, protect intermediate values, and evaluate the resulting machine code on the target platform. FIPS 204 supports hedged signing, with randomness derived from secret K, the message representative μ, and input *rnd*; deterministic operation fixes *rnd*, but does not remove K or the secret-related rejection conditions.

Decapsulation and the Fujisaki–Okamoto transform. The subtlest PQC side channel lives in KEM decapsulation. A raw lattice decryption can *fail*, and whether it failed depends on the secret key and the (possibly attacker-chosen) ciphertext. If decapsulation ever behaved differently on success versus failure – returned an error, took longer, or drew different power – the attacker would gain a *decryption-failure* or *plaintext-checking* oracle, querying malformed ciphertexts and using the yes/no answers to peel the secret apart. The Fujisaki–Okamoto (FO) transform [26] closes this door with **implicit rejection**. Decapsulation re-encrypts the recovered message and compares the result to the received ciphertext in constant time. On a match it returns the real shared key; on a mismatch it returns a *pseudorandom* key derived from a secret seed and the ciphertext – not a distinguishable ciphertext-validity error. This applies after basic input and decapsulation-key checks; an API may still reject malformed types, lengths, or unusable key material. Both cryptographic outcomes produce a key of identical form, and the final choice is a constant-time select, so the caller cannot tell success from failure. Figure 26.2 shows the flow.

The NTT and polynomial arithmetic. The number-theoretic transform and the modular polynomial arithmetic that surround it run on secret data on every operation. They must be constant-time too: the modular reductions after each butterfly should use fixed-cost Montgomery or Barrett reduction rather than a hardware division, and there must be no secret-dependent branch anywhere in the transform. This is usually easy to achieve – the NTT is naturally straight-line – but it is easy to spoil with a careless conditional subtraction that runs only "when the coefficient is too large."

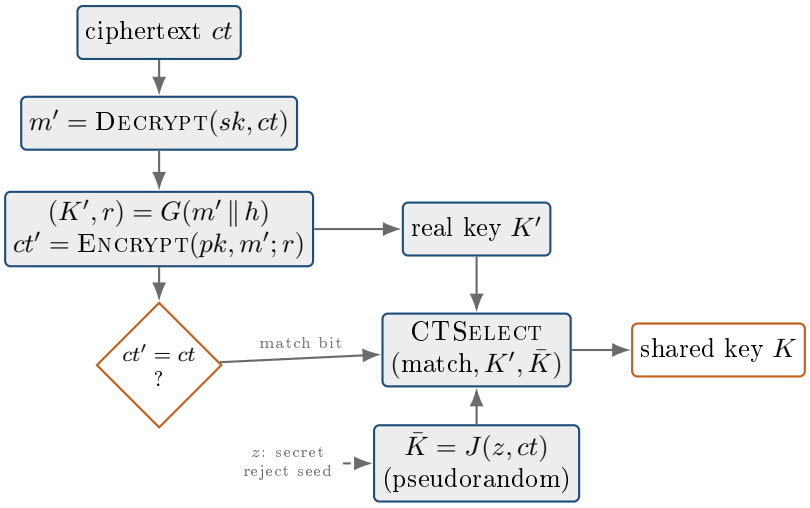


Figure 26.2: FO decapsulation with implicit rejection. The recovered message is re-encrypted and compared to the received ciphertext in constant time. A match yields the real key K'; a mismatch yields a pseudorandom key $\bar{K} = J(z, ct)$ derived from a secret seed z. Because the two outputs have identical form and the choice is a constant-time select, success and failure are indistinguishable to the caller – no decryption-failure oracle exists.

26.5 Countermeasures beyond constant-time

Constant-time coding primarily addresses timing, cache, and secret-dependent control-flow leakage. Differential power and EM analysis are stronger, because they average many traces and can pull out a secret that influences the power draw even when the control flow is perfectly uniform. Defeating those needs a further layer.

Masking. The main tool is **masking**: split every secret value into several random *shares* whose combination (XOR, for Boolean masking; addition mod q, for arithmetic masking) equals the secret, and compute on the shares so that no single intermediate value in the whole computation ever equals, or correlates with, the secret. A first-order masking uses two shares and forces the attacker to combine measurements from two points at once; an order-d masking uses $d+1$ shares. The security grows with d, but so does the cost – the number of shares grows linearly while the number of secure operations and the fresh randomness they consume grow faster, so masking overhead is often several-fold and sometimes an order of magnitude. For lattice schemes the specific difficulty is that they mix *Boolean* operations (hashing, comparison in the FO check) with *arithmetic* operations (the NTT), and securely converting between Boolean and arithmetic masking is the expensive, delicate part.

Blinding. A lighter-weight relative is **blinding**: randomize the *representation* of a computation without changing its result, so that traces do not line up across runs. One can add a random multiple of the modulus to a value before an operation, randomize the projective coordinates in an elliptic-curve step, or shuffle the order in which independent coefficients are processed so the attacker cannot know which sample in a trace corresponds to which secret element. Blinding is cheaper than full masking but weaker; the two are often combined.

Cost and benefit. None of this is free, and none of it is always necessary. A server behind a network, where an attacker can only measure timing remotely, needs constant-time code but rarely full masking. A smart card or an embedded key store, where the attacker holds the device and can wire up an oscilloscope, needs masking and fault countermeasures as well. The right level is a threat-model decision, not a fixed recipe – but constant-time coding is the floor beneath all of it.

26.6 Pitfalls

Pitfall 1: assuming the compiler preserves your constant-time code. An optimizing compiler is free to rewrite $(a \,\&\, mask) \mid (b \,\&\, \overline{mask})$ back into a branch or a conditional move, or to short-circuit a masked comparison, if it decides that is faster. Constant-time behavior is a property of the emitted machine code, not of the source. It must be checked – by reading the assembly, using verification tools, or barriers that stop the optimizer – not assumed.

Pitfall 2: confusing constant-time with constant wall-clock time. Constant-time does not mean the code always runs for the same number of nanoseconds. It means no *secret*-dependent variation. Variation that depends only on public data – for example the length of a message – is perfectly acceptable. Chasing literally constant duration is both unnecessary and, on modern out-of-order CPUs, impossible.

Pitfall 3: reporting a decapsulation failure. Returning an error, or logging, or taking a fast path when lattice decryption fails, hands the attacker exactly the plaintext-checking oracle the FO transform was built to deny. Decapsulation must always run to completion and always return a key; use implicit rejection, never an error branch.

Pitfall 4: comparing secrets with `memcmp`. The standard library comparison returns as soon as it finds a differing byte, so its timing reveals how many leading bytes matched – fatal for a MAC tag or the FO ciphertext check. Always use a no-early-exit comparison such as CT_EQ.

Pitfall 5: masking that leaks anyway. Two shares that are correct on paper still leak if the hardware or the compiler recombines them – for example, if consecutive instructions write both shares to the same register or bus, or if masking is applied without fresh, independent randomness each run. Masking is only as good as the independence of its shares in the actual silicon.

26.7 Summary

It is worth stating the lesson of this chapter plainly. The post-quantum mathematics we spent the previous chapters building is genuine: it defeats Shor's algorithm, and no efficient quantum attack on the underlying lattice and code problems is known. But that mathematics protects only the abstract scheme. A careless implementation can hand the very same secret to an attacker with an oscilloscope, a shared cache, or a laser – an attack that costs minutes and dollars rather than a quantum computer.

The defense is a discipline, not a gadget:

- write **constant-time** code – no secret-dependent branches, no secret-dependent memory addresses, no variable-time instructions on secrets;

- where the threat model includes physical access, add **masking** and **fault** countermeasures on top;
- verify the property in the emitted machine code, because the compiler does not promise to preserve it.

And this is where the theory of the previous parts closes the loop with practice. The standards did not treat implementation security as an afterthought; they engineered for it. ML-KEM's CBD sampler avoids a general discrete Gaussian (15); ML-DSA and FN-DSA require careful treatment of secret-dependent sampling and rejection conditions (17, 18); ML-KEM decapsulation is built on implicit rejection so success and failure are indistinguishable (16); and Falcon's floating-point sampler stands as the standing reminder that when a scheme is hard to implement safely, that difficulty is itself a security cost (18). Post-quantum security is not only a theorem to be proved. It is a property of the running system – and keeping it there, all the way down to the machine code, is not optional.

Chapter 27

Deploying Post-Quantum Cryptography

27.1 Why this chapter matters

The previous parts built and analysed the algorithms. This chapter is about the harder half of the problem: getting them into real systems – web protocols, digital certificates, and even smart contracts – correctly and in time. The forcing function is *harvest now, decrypt later* (Chapter 1): an adversary can record encrypted traffic today and decrypt it once a quantum computer exists, so any long-lived secret is already exposed. Migration must therefore begin before the threat arrives, not after, and migration is an engineering project at least as much as a mathematical one.

27.2 Learning goals

After reading this chapter, you should be able to:

- explain why real deployments use *hybrid* classical-plus-postquantum cryptography, and what crypto-agility means;
- describe how TLS 1.3 carries a hybrid key exchange and post-quantum certificates, and what the size costs are;
- explain why verifying post-quantum signatures on-chain is expensive and how projects reduce that cost;
- weigh the size and performance trade-offs that drive scheme choice in practice.

27.3 Hybrid cryptography and crypto-agility

The safest way to deploy a young algorithm is not to trust it alone. A *hybrid* scheme combines a well-understood classical algorithm with a post-quantum one through an analysed protocol construction.

Definition 27.1 (Hybrid key exchange)**.** A hybrid key exchange combines a classical key-agreement contribution (for example X25519 ECDHE) with a post-quantum KEM contribution (for example ML-KEM-768) through an analysed protocol construction. X25519 is Diffie–Hellman key agreement, not a KEM. A robust construction binds algorithm identifiers, peer identities, and the handshake transcript in a KDF extract/expand step. Its design goal is to retain session-key security when at least one component remains secure, subject to the combiner and protocol's required conditions; arbitrary hash concatenation is not a general theorem.

Hybrid deployment hedges two opposite risks at once: a future quantum computer breaking the classical part, and an as-yet-undiscovered classical weakness in the still-young post-quantum part. It is the mainstream migration strategy precisely because it removes the need to bet everything on one new assumption.

Remark (Crypto-agility). Hybrids only work if systems can *negotiate* which algorithms they use rather than hardcoding one. *Crypto-agility* – designing protocols, certificate formats, and code so the algorithm can be swapped without re-architecting – is the quiet prerequisite for the whole migration. The schemes will keep evolving; the plumbing must not assume any single one.

27.4 Post-quantum TLS 1.3

TLS 1.3 secures most web traffic, and it is the first place post-quantum cryptography has reached production. Two independent parts of the handshake must be upgraded: the *key exchange* (confidentiality) and the *authentication* (signatures on certificates).

27.4.1 Hybrid key exchange

As of this book's standards snapshot, TLS hybrid ECDHE–ML-KEM groups are under IETF standardization and are appearing in implementations. A group such as `X25519MLKEM768` combines an X25519 share with an ML-KEM contribution; the exact wire format, group naming, enablement, and TLS key-schedule binding must follow the applicable IETF specification and implementation release notes.

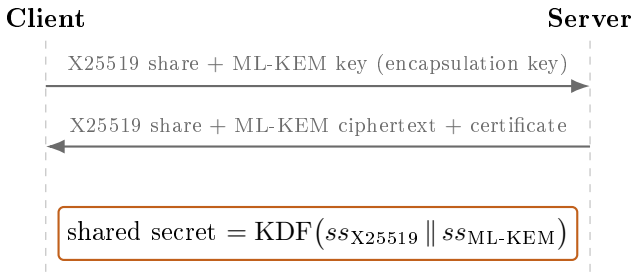


Figure 27.1: Illustrative hybrid key exchange in TLS 1.3. Both an elliptic-curve share and an ML-KEM contribution travel in the handshake; an analysed KDF construction binds both secrets and the protocol context. Post-quantum certificate authentication is a separate, draft/experimental deployment track. Specific group names and deployment status are version-dependent.

27.4.2 Post-quantum authentication and its size cost

Post-quantum certificate authentication is still a separate standardization and deployment track. ML-DSA-65 is one plausible illustrative choice for its balance of speed and size, while SLH-DSA offers a more conservative assumption at larger sizes. The practical obstacle is size: post-quantum public keys and signatures are far larger than the elliptic-curve ones they replace, and a TLS handshake may carry several certificates. The extra kilobytes cost round-trips and can collide with buffer and fragmentation limits.

Signature scheme	Public key (bytes)	Signature (bytes)
Ed25519 (classical)	32	64
ML-DSA-65 (FIPS 204)	1952	3309
FN-DSA / Falcon-512 (FIPS 206 in development)	897	≈ 666
SLH-DSA-128s (FIPS 205)	32	≈ 7856

Table 27.1: Approximate sizes of a classical signature, published PQC standards, and the draft-dependent FN-DSA candidate. Falcon/FN-DSA is not a final FIPS at this standards snapshot.

Implementation and deployment status changes rapidly. Before making a production decision, consult the current IETF document and the release notes of the selected browser, TLS library, or service for its supported groups, defaults, and authentication behaviour.

27.5 Verifying post-quantum signatures on-chain

A very different deployment target is the blockchain, where a smart contract may need to *verify* a post-quantum signature – for a quantum-resistant wallet, or a cross-chain bridge. Here the constraint is not bandwidth but *gas*: every operation the Ethereum Virtual Machine (EVM) executes costs money, and a lattice verifier does a great deal of modular arithmetic.

Direct verification of a post-quantum signature can be expensive in an EVM-like environment because it requires substantial hashing and modular arithmetic. The cost depends on the scheme, parameter set, contract implementation, chain, hard-fork gas schedule, and available precompiles. Claims about a dominant operation or a gas figure require a reproducible benchmark with those details.

When even that is too expensive, an alternative is to avoid running the verifier on-chain at all.

- **Direct verification**: run the full verifier in the contract. It is conceptually simple and minimizes added trust assumptions, but may be expensive.
- **ZK proof of verification**: prove off-chain that a signature verifies, then check a succinct proof on-chain. This moves work to the prover and introduces proof-system assumptions and costs.
- **Optimistic or fraud-proof designs**: accept a claim subject to a challenge procedure. Their cost and trust trade-offs depend on the exact challenge game and security model.

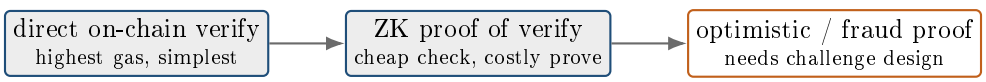


Figure 27.2: Three broad designs for on-chain post-quantum signature verification. They trade direct execution cost against prover work, challenge mechanisms, and security assumptions; there is no universal cost ordering without a specified benchmark and protocol.

27.6 Sizes, performance, and implementation reality

Choosing a scheme in practice is mostly a negotiation among sizes, speed, and assumption strength.

Scheme	Role	Public key	Ciphertext / signature
ML-KEM-768 (FIPS 203)	KEM	1184	1088
ML-DSA-65 (FIPS 204)	signature	1952	3309
FN-DSA / Falcon-512 (FIPS 206 in development)	signature	897	≈ 666
SLH-DSA-128s (FIPS 205)	signature	32	≈ 7856

Table 27.2: Approximate sizes (in bytes) of published standards and the draft-dependent FN-DSA candidate. There is no universally best choice: SLH-DSA minimizes assumptions but has large signatures, while ML-DSA is a general-purpose published standard.

Two constraints cut across all of these. First, operations that handle secret keys, ephemeral secrets, or secret-dependent sampling – especially key generation, signing, and decapsulation – require side-channel review against the applicable threat model (Chapter 26). Public-input verification has different confidentiality requirements, though it still needs robustness and denial-of-service review. Second, the sizes in Table 27.2 are not academic – they determine whether a scheme fits in a TLS handshake, a certificate chain, or a smart-contract call.

27.7 Pitfalls

Pitfall 1: deploying post-quantum only, too early. Dropping the classical algorithm and trusting the post-quantum one alone forfeits the hedge that hybrid provides. Until the post-quantum schemes have had many more years of scrutiny, hybrid is the conservative default.

Pitfall 2: ignoring handshake and message sizes. Post-quantum keys and signatures are kilobytes, not tens of bytes. Buffers, packet-size assumptions, and certificate-chain limits that were fine for elliptic curves can silently break; sizes must be designed for, not discovered in production.

Pitfall 3: assuming a reference implementation is deployable. Reference code optimizes for clarity, not for constant-time behaviour. Shipping it unmodified can reintroduce exactly the side channels of Chapter 26.

Pitfall 4: hardcoding one algorithm. The standards will keep changing. A system that cannot negotiate or swap algorithms will have to be re-engineered at the worst possible time.

27.8 Summary

The mathematics of the previous parts defeats Shor's algorithm, but that is where the abstract scheme ends and the engineering begins:

- **hybrid** classical-plus-postquantum deployment, with **crypto-agility**, is the mainstream migration strategy;
- **TLS 1.3** hybrid key exchange is under standardization and deployment; protocol and implementation versions must be checked at the time of use;
- verifying signatures **on-chain** has scheme- and environment-dependent costs, with direct, proof-based, and optimistic designs offering different trade-offs;

- scheme choice is a trade-off among size, speed, and assumption strength, with side-channel controls scoped to the secrets and threat model of each operation.

Post-quantum cryptography is no longer a paper exercise: the standards exist, and the migration is under way. Real security will arrive only when these schemes are deployed correctly, agilely, and side-channel-safely across the protocols and systems that the world already depends on. The next chapter turns to one of the hardest cases of all – the blockchains, where the funds themselves are secured by exactly the signatures now at risk.

Chapter 28

Post-Quantum Migration of Bitcoin and Ethereum

28.1 Why this chapter matters

Blockchains are one of the hardest post-quantum migration problems in existence, for two reasons the previous chapter only touched. First, the funds are secured by *signatures* whose public keys are, sooner or later, *permanently public* on an immutable ledger – so "harvest now, decrypt later" becomes a much sharper "store now, steal later." Second, there is no administrator: changing the rules of Bitcoin or Ethereum is a governance problem settled by rough consensus and forks, not a software update pushed to servers. This chapter looks at how the two largest chains are actually approaching the transition, as of mid-2026. Everything here is a moving target: the concrete proposals are drafts, not deployed consensus, and the reader should treat dates and statuses as a snapshot.

28.2 Learning goals

After reading this chapter, you should be able to:

- explain which blockchain primitives a quantum computer breaks (Shor) versus merely weakens (Grover), and the public-key-exposure principle;
- describe Bitcoin's exposure by output type and the current draft proposals (BIP-360, BIP-361);
- describe Ethereum's exposure and its two-front plan: an emergency hard fork and account-abstraction-based migration;
- weigh the central controversy – whether to freeze quantum-vulnerable coins – and the honest uncertainty in the timeline.

28.3 The blockchain threat model

The immediate migration pressure is on signatures and pairing-based public-key systems. Shor's algorithm (Chapter 1) threatens Bitcoin's ECDSA and Schnorr, Ethereum's account ECDSA and consensus BLS signatures, and the public-key assumptions behind SNARK and KZG systems [51,

60]. Grover gives hashes a square-root rather than exponential weakening, but hashes still require property- and output-length-specific assessment: HASH160, for example, has only a nominal 2^{80} quantum preimage margin.

The subtlety that governs everything is **when a public key becomes visible**. P2PK and P2TR expose a public key or tweaked public key when a UTXO is created; P2PKH and P2WPKH usually reveal it on first spend, and reuse leaves later outputs exposed. Other scripts must be assessed by their actual path and witness.

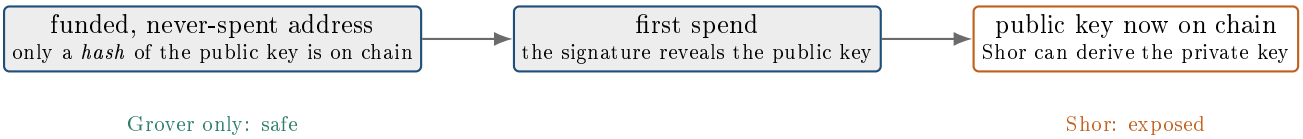


Figure 28.1: The public-key-exposure lifecycle. While an address has only ever received funds, its public key is hidden behind a hash and only Grover applies. The first spend publishes the public key permanently, exposing the account to Shor's algorithm. This is why exposed coins are considered at risk indefinitely: the ledger never forgets, so an attacker with a future quantum computer needs no fresh interception – the target is already recorded.

28.4 Bitcoin

28.4.1 What is exposed

Bitcoin's exposure is precisely a function of output type. Coins in *pay-to-public-key* (P2PK, the earliest, Satoshi-era outputs), raw multisig, and Taproot key-path outputs carry a public key on chain permanently and are long-exposure vulnerable. Coins in the hash-wrapped types (P2PKH, P2SH, P2WPKH) stay protected *until the address is reused or spent*, at which point the key is revealed [53, 52]. BIP-361 states that, as of 1 March 2026, over **34% of all bitcoin have revealed a public key on chain** [54]; an independent Chaincode Labs analysis puts the exposed fraction in the range of roughly one-fifth to one-half of supply [52]. (Older and widely repeated figures attributing a specific amount to Satoshi-era P2PK are best treated as estimates rather than established fact.)

28.4.2 The draft proposals

Two Bitcoin Improvement Proposals anchor the response, and it is worth stressing that both are **Draft and not activated** as of mid-2026:

- **BIP-360, "Pay-to-Merkle-Root" (P2MR)** [53] – formerly circulated as "Pay-to-Quantum-Resistant-Hash" (P2QRH) – introduces a new output type, proposed as a backward-compatible soft fork. In its current form it is essentially Taproot with the key-path spend removed, so that no bare public key is ever committed; notably, the current draft *defers* the choice of the actual post-quantum signature scheme (naming ML-DSA and SLH-DSA only as examples) to a future proposal, rather than mandating one.
- **BIP-361, "Post-Quantum Migration and Legacy Signature Sunset"** [54] – the migration-and-deadline layer. It proposes a phased sunset of ECDSA/Schnorr: a first phase (on the order of three years after activation) that stops sending funds to legacy address types, and a later flag-day (around five years) that restricts legacy signatures, so that only a holder who can prove ownership by a quantum-safe route can still move the coins.

A recurring safety mechanism is *commit–reveal*: a holder first publishes a binding commitment and only later reveals it. It is designed to mitigate front-running under the stated confirmation-depth, reorganization, fee, timeout, and recovery assumptions; it is not an unconditional guarantee.

28.4.3 The freeze controversy

BIP-361's sunset implies that coins whose owners never migrate – and coins for which no ownership proof is possible, notably P2PK – become effectively unspendable, i.e. *frozen*. This is the sharpest open dispute in Bitcoin's roadmap. Proponents argue a pre-announced sunset is defensive: a quantum attacker would otherwise *steal* those coins outright, whereas a freeze at least preserves them. Critics call any forced migration confiscatory, contrary to Bitcoin's property-rights ethos, and warn that freezing long-dormant coins (including Satoshi's) sets a dangerous precedent. This is a genuine, unresolved political disagreement, not a settled plan, and should be read as such.

28.5 Ethereum

28.5.1 What is exposed

Ethereum addresses reveal an effectively public key once an ECDSA transaction has been sent. BLS validator signatures, pairing curves behind SNARK verifiers, and KZG commitments all rely on public-key assumptions threatened by Shor [60, 51]. A comparison of quantum cost for secp256k1 and BLS12-381 needs group-specific resource estimates; the 381-bit base-field name alone is not such an estimate. KZG migration must distinguish blob retention from commitment binding, proof verification, and consensus dependencies; blob pruning alone does not make the risk mild.

28.5.2 The two-front plan

Ethereum's response has an emergency track and a structural track; nothing post-quantum has yet shipped to mainnet.

The quantum-emergency hard fork. Vitalik Buterin's proposal "How to hard-fork to save most users' funds in a quantum emergency" [55] is the contingency plan. On evidence of large-scale theft, the chain would revert to the last pre-theft block, freeze legacy ECDSA accounts, and let users recover funds through a new transaction type carrying a *STARK proof* that the sender knows the hash preimage secret behind their address – authenticating by knowledge of a hash preimage rather than by an ECDSA signature. Because most users' recovery secret was never exposed on chain, this protects the majority, at the cost of a wallet-software update. Buterin notes the infrastructure could in principle be built now.

Account abstraction as the migration vehicle. The structural plan is to let accounts choose their own signature scheme. **EIP-7702** ("Set Code for EOAs," *Final*, shipped in the 2025 Pectra upgrade) [56] lets an externally-owned account delegate to contract code, turning it into a smart account with arbitrary verification logic – although 7702 authorizations are themselves secp256k1-signed, so it is a stepping stone, not itself a post-quantum migration. The dedicated vehicle is **EIP-8141** ("Frame Transaction," *Draft*, 2026, co-authored by Buterin) [57], which explicitly describes native account abstraction as "a native off-ramp ... to post-quantum secure systems," supporting a slot for an arbitrary (custom, post-quantum) signature scheme alongside the classical ones. At the consensus layer, the plan is to replace BLS aggregation with a hash-based signature of the

XMSS/Winternitz family (Chapter 19), aggregated inside a STARK-based zero-knowledge virtual machine, and to move KZG commitments to STARK- or lattice-based alternatives; the Ethereum Foundation's stated assessment targets first-layer post-quantum infrastructure around 2029 [60].

28.5.3 On-chain verification, revisited

The structural migration path may introduce post-quantum signature verification on chain; the emergency path instead proposes a STARK proof of seed/preimage knowledge. These are different authentication mechanisms. Their costs are scheme-, parameter-, implementation-, chain-, and gas-schedule-dependent, so a benchmark must state those details, including code commit and measurement date, before comparing gas figures.

	Bitcoin	**Ethereum**
Fund-securing signature	ECDSA / Schnorr (secp256k1)	ECDSA (accounts); BLS12-381 (consensus)
Exposed today	P2PK, reused P2PKH, Taproot key-path	any account that has sent a transaction
Migration vehicle	new PQ output type (BIP-360)	account abstraction (EIP-7702, EIP-8141)
Emergency plan	legacy-signature sunset (BIP-361)	emergency hard fork with STARK recovery
Fork style	soft fork	protocol upgrades / emergency hard fork
Status (mid-2026)	Draft BIPs, not activated	research; one AA piece shipped; PQ infra ∼2029

Table 28.1: Bitcoin and Ethereum post-quantum migration at a glance, as of mid-2026. Both chains face the same exposure principle but respond with different mechanisms and governance styles, and in both cases the concrete post-quantum proposals are drafts rather than deployed consensus.

28.6 Timelines and honest uncertainty

No quantum computer capable of breaking secp256k1 or BLS12-381 exists as of mid-2026. A 2026 arXiv preprint, not a consensus survey, gives one set of model-dependent estimates for cryptographically relevant quantum computing [51]. A 2025 Google analysis of RSA-2048 illustrates a downward resource-estimation trend, but is RSA-specific and not a direct estimate for Bitcoin or Ethereum curves [50].

28.7 Pitfalls

Pitfall 1: assuming "my address is a hash, so I am safe." True only until the first spend. Any account that has transacted has exposed its public key; on Ethereum that is essentially every active account, and on Bitcoin it includes every reused address.

Pitfall 2: treating draft proposals as decided. BIP-360, BIP-361, and EIP-8141 are drafts; the freeze question in particular is contested. Reporting them as Bitcoin's or Ethereum's "plan" overstates consensus.

Pitfall 3: quoting a single gas figure. On-chain verification costs move between implementations, versions, and precompile assumptions. A number without a source and date is not meaningful.

28.8 Summary

Blockchains sharpen the quantum threat and complicate the response:

- the danger is to signatures (Shor), not hashing (Grover); an exposed public key on an immutable ledger is a permanent, "store now, steal later" liability;
- **Bitcoin** proposes a new quantum-hardened output type (BIP-360) and a contested legacy-signature sunset (BIP-361), both draft soft forks;
- **Ethereum** pairs an emergency hard fork with STARK-based hash-preimage recovery and an account-abstraction migration path (EIP-7702 shipped, EIP-8141 draft), targeting first-layer post-quantum infrastructure around 2029;
- the timeline is uncertain but the required lead time is long, which is why both communities are debating the transition years before any quantum computer can execute the attack.

This is where cryptographic theory meets the messiest realities of deployment – immutable history, decentralized governance, and irreversible loss – and it is a fitting place to close: the mathematics is ready, and the harder work of migrating the systems that depend on it has only begun.

Appendix A

Binary Goppa Codes

Classic McEliece (Chapter 22) hides its trapdoor inside a **binary Goppa code**: a family of algebraic codes that come with a fast t-error decoder yet, after four decades of scrutiny, still cannot be distinguished from random linear codes by any known efficient test. The main text used only two facts about them – that they correct t errors and that they have an efficient decoder. This appendix supplies the construction behind those facts: how a Goppa code is defined, why the binary case corrects the full t errors rather than $t/2$, and how Patterson's algorithm decodes it [35, 36]. Throughout we assume the finite fields of Chapter 4.

A.1 Setup: the field, the support, and the Goppa polynomial

Fix an extension field $\mathbb{F}_{2^m}$ of the binary field. A binary Goppa code is built from two ingredients.

- A **support** $L = (\alpha_1, \ldots, \alpha_n)$ of n *distinct* elements of $\mathbb{F}_{2^m}$. Since the field has 2^m elements, $n \leq 2^m$.
- A **Goppa polynomial** $g(x) \in \mathbb{F}_{2^m}[x]$ of degree t, subject to $g(\alpha_j) \neq 0$ for every j – so that each $x - \alpha_j$ is invertible modulo g. For the Patterson decoder below, assume in addition that g is monic and irreducible, as in Classic McEliece.

The codewords will be binary vectors of length n; the field $\mathbb{F}_{2^m}$ is the scaffolding in which the defining constraints live. The number t is the design parameter: it will be the number of errors the code corrects.

A.2 Definition

Definition A.1 (Binary Goppa code)**.** The *binary Goppa code* $\Gamma(L, g)$ is the set of binary vectors whose coordinates, weighted by the poles $x - \alpha_j$, sum to zero modulo g:

$$\Gamma(L, g) \;=\; \Big\{ c = (c_1, \ldots, c_n) \in \mathbb{F}_2^n \;:\; \sum_{j=1}^{n} \frac{c_j}{x - \alpha_j} \equiv 0 \pmod{g(x)} \Big\},$$

where each $\dfrac{1}{x - \alpha_j}$ denotes the inverse of $x - \alpha_j$ in the ring $\mathbb{F}_{2^m}[x]/(g)$, which exists because $g(\alpha_j) \neq 0$.

The syndrome map is linear over $\mathbb{F}_{2^m}$; restricting its kernel to $\mathbb{F}_2^n$ gives an $\mathbb{F}_2$-linear code. This binary restriction is exactly what makes the binary case special, as Section A.4 shows.

A.3 The parity-check matrix

To turn the congruence into a matrix, expand each pole. Because $g(\alpha_j) \neq 0$, one has the identity

$$\frac{1}{x-\alpha_j} \;\equiv\; -g(\alpha_j)^{-1}\,\frac{g(x)-g(\alpha_j)}{x-\alpha_j} \pmod{g(x)},$$

in which $\big(g(x)-g(\alpha_j)\big)/(x-\alpha_j)$ is an honest polynomial of degree $t-1$. Collecting the coefficients of $1, x, \ldots, x^{t-1}$ in the defining sum and setting each to zero yields a parity-check matrix over $\mathbb{F}_{2^m}$ of the Vandermonde-times-diagonal form

$$H \;=\; \underbrace{\begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_n \\ \vdots & & & \vdots \\ \alpha_1^{t-1} & \alpha_2^{t-1} & \cdots & \alpha_n^{t-1} \end{pmatrix}}_{\text{Vandermonde}} \underbrace{\begin{pmatrix} g(\alpha_1)^{-1} & & \\ & \ddots & \\ & & g(\alpha_n)^{-1} \end{pmatrix}}_{\text{diagonal}} \;\in\; \mathbb{F}_{2^m}^{t\times n}.$$

A vector c lies in $\Gamma(L,g)$ exactly when $Hc^{\top} = 0$. This is the parity check of an *alternant* code, of which Goppa codes are the important special case.

To obtain a *binary* parity-check matrix, replace each entry of H by the column of m bits giving its coordinates in a fixed $\mathbb{F}_2$-basis of $\mathbb{F}_{2^m}$. This expands the $t \times n$ matrix over $\mathbb{F}_{2^m}$ into an $(mt) \times n$ matrix over $\mathbb{F}_2$. Its rows may be dependent, so the dimension satisfies

$$k \;=\; \dim \Gamma(L,g) \;\geq\; n - mt.$$

In Classic McEliece's regime – m around 12–13, n a few thousand, t around 60–130 – this bound is met with near equality.

A.4 Minimum distance: why the binary code corrects t errors

A general alternant code with a degree-t polynomial guarantees only $d \geq t+1$, hence correction of $\lfloor t/2 \rfloor$ errors. The binary Goppa code does twice as well, provided g is squarefree.

Proposition A.2 (Doubled distance for binary Goppa codes)**.** *If g is* squarefree *(equivalently* separable*: it has no repeated roots in the algebraic closure), then*

$$\Gamma(L,g) \;=\; \Gamma(L,g^2),$$

and consequently its minimum distance obeys $d \geq 2t+1$. The code therefore corrects up to t errors.

Proof sketch. For a binary c, let $\sigma_c(x) = \prod_{j:\,c_j=1}(x-\alpha_j)$ be the locator of its support. A short computation gives

$$\sum_{j:\,c_j=1} \frac{1}{x-\alpha_j} \;=\; \frac{\sigma_c'(x)}{\sigma_c(x)},$$

where σ_c' is the formal derivative. Since $\gcd(\sigma_c, g) = 1$ (no α_j is a root of g), the defining condition $c \in \Gamma(L,g)$ is equivalent to $g \mid \sigma_c'$. Now over characteristic 2 the derivative kills every even-degree term, so $\sigma_c'(x)$ contains only even powers of x; because squaring is the Frobenius map on $\mathbb{F}_{2^m}$, such a polynomial is a *perfect square*, $\sigma_c' = B^2$ for some $B \in \mathbb{F}_{2^m}[x]$. As g is squarefree,

$$g \mid B^2 \iff g \mid B \iff g^2 \mid B^2 = \sigma_c',$$

so $g \mid \sigma_c' \Leftrightarrow g^2 \mid \sigma_c'$, i.e. $\Gamma(L,g) = \Gamma(L,g^2)$. The right-hand code is alternant with polynomial of degree $2t$, giving $d \geq 2t+1$. □

The whole point of using binary codewords is this free doubling: a Goppa polynomial of degree t buys the error-correction of a designed distance $2t+1$.

A.5 Patterson's decoding algorithm

Decoding is the trapdoor. Given a received word $y = c+e$ with $c \in \Gamma(L,g)$ and $\mathrm{wt}(e) \le t$, Patterson's algorithm [36] recovers e using g and L as the secret. It exploits the same characteristic-2 square structure used in Proposition A.2.

Write $E = \{\, j : e_j = 1 \,\}$ for the (unknown) set of error positions and let the **error-locator polynomial** be $\sigma(x) = \prod_{j \in E}(x - \alpha_j)$; its roots among L are exactly the errors. The algorithm reconstructs σ from the syndrome.

Algorithm 40 Patterson decoding of a binary Goppa code $\Gamma(L,g)$ with monic irreducible g, $\deg g = t$

Input: Received word $y \in \mathbb{F}_2^n$ with at most t errors; secret L, monic irreducible g
Output: Error vector e (equivalently the codeword $c = y - e$)

1: $S(x) \leftarrow \sum_{j=1}^{n} \dfrac{y_j}{x - \alpha_j} \bmod g(x)$ ▷ syndrome; the codeword part vanishes, so $S \equiv \sum_{j \in E} \frac{1}{x-\alpha_j}$
2: **if** $S \equiv 0$ **then**
3: **return** $e = 0$ ▷ y is already a codeword
4: **end if**
5: $T(x) \leftarrow S(x)^{-1} \bmod g(x)$
6: $R(x) \leftarrow \sqrt{T(x) + x} \bmod g(x)$ ▷ unique square root in the field $\mathbb{F}_{2^m}[x]/(g)$
7: find $a(x), b(x)$ with $a \equiv R\,b \pmod g$, $\deg a \le \lfloor t/2 \rfloor$, $\deg b \le \lfloor (t-1)/2 \rfloor$
▷ one truncated run of the extended Euclidean algorithm on g and R
8: $\sigma(x) \leftarrow a(x)^2 + x\,b(x)^2$ ▷ the error locator
9: $E \leftarrow \{\, j : \sigma(\alpha_j) = 0 \,\}$ ▷ roots of σ in L
10: **return** e with $e_j = 1$ iff $j \in E$

The two non-obvious lines are the split and the key equation, and both come from characteristic 2.

- **The split $\sigma = a^2 + x\,b^2$.** Any polynomial over $\mathbb{F}_{2^m}$ separates into even-degree and odd-degree parts; the even part is a perfect square a^2, and the odd part is x times a perfect square $x\,b^2$. Differentiating, $(a^2)' = 0$ and $(xb^2)' = b^2$, so $\sigma' = b^2$.

- **The key equation.** Because $S = \sigma'/\sigma \bmod g$, we have $\sigma' \equiv \sigma S$, i.e. $b^2 \equiv (a^2 + xb^2)S \pmod g$. Multiplying by $T = S^{-1}$ and rearranging gives $a^2 \equiv b^2\,(T + x) \pmod g$. Taking the square root $R = \sqrt{T + x}$ turns this into the linear-looking congruence $a \equiv R\,b \pmod g$, which the extended Euclidean algorithm solves for a pair of the required small degrees.

Once σ is known, evaluating it at every α_j (a *Chien search*) locates the errors, and over $\mathbb{F}_2$ correcting them is a bit flip. The cost is dominated by the extended Euclidean step and the root search, both quasi-linear in n with fast arithmetic; this is the "efficient decoder $\mathcal{D}_g$" the main text invoked.

A.6 A construction in SageMath

The whole object is short to build. Take $m = 4$, so $\mathbb{F}_{16}$, and a squarefree Goppa polynomial of degree $t = 2$.

```
F.<a> = GF(2^4)
PR.<x> = F[]
g = x^2 + x + a
while not g.is_irreducible():
    g = x^2 + F.random_element()*x + F.random_element()
assert g.is_irreducible()
L = [b for b in F if g(b) != 0]     # support: exclude any roots of g
n = len(L)

from sage.coding.goppa_code import GoppaCode
C = GoppaCode(g, L)
print(C)                            # a binary [n, k] Goppa code
print(C.dimension(), n - g.degree() * F.degree())   # k  vs  n - m t
```

Here $mt = 4 \cdot 2 = 8$, so the dimension satisfies $k \geq n - 8$, and by Proposition A.2 the minimum distance is at least $2t + 1 = 5$, so the code corrects $t = 2$ errors. This demonstrates the algebraic core only. Engineering-grade Classic McEliece additionally fixes parameter sets, support ordering, systematic-matrix reduction, encoding, KEM behavior, failure handling, and constant-time requirements.

A.7 Why Goppa codes, and not others

Three properties, taken together, are what make binary Goppa codes the enduring choice for McEliece.

- **An efficient decoder.** Patterson's algorithm corrects the full t errors in near-linear time – the trapdoor the legitimate key holder uses.
- **A vast family.** There are enormously many choices of support L and polynomial g, so keys cannot be enumerated, and a random secret code is overwhelmingly unlikely to coincide with any particular target.
- **Indistinguishability.** No known efficient algorithm distinguishes a scrambled binary Goppa code from a random linear code of the same parameters. This is the security-critical property, and it is where less conservative families failed: the generalized Reed–Solomon codes of the original Niederreiter proposal were broken by the Sidelnikov–Shestakov structural attack, and several Reed–Muller and algebraic-geometry variants fell the same way. Binary Goppa codes have resisted every such attack since 1978.

This is the concrete content behind the "decodable code in disguise" of Chapter 22: the disguise is the scrambled generator matrix, and the code being disguised is $\Gamma(L, g)$.

Further reading

Goppa introduced the codes in 1970 [35]; Patterson gave the binary decoder in 1975 [36]; McEliece turned them into a cryptosystem in 1978 [34]; and the current parameters and constant-time engineering are specified by the Classic McEliece submission [39].

Bibliography

[1] National Institute of Standards and Technology. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. FIPS 203, 2024.

[2] National Institute of Standards and Technology. *Module-Lattice-Based Digital Signature Standard*. FIPS 204, 2024.

[3] National Institute of Standards and Technology. *Stateless Hash-Based Digital Signature Standard*. FIPS 205, 2024.

[4] National Institute of Standards and Technology. *FN-DSA (Fast-Fourier, NTRU) Digital Signature Standard*. FIPS 206. Listed by NIST as "in development" at this book's standards snapshot: no initial public draft had been released, so the round-3 Falcon submission [32] remains the authoritative specification.

[5] National Institute of Standards and Technology. *Automated Cryptographic Validation Protocol (ACVP): test vectors for ML-KEM, ML-DSA, and SLH-DSA*. `https://github.com/usnistgov/ACVP-Server`.

[6] D. J. Bernstein. Introduction to post-quantum cryptography. In *Post-Quantum Cryptography*, pages 1–14. Springer, 2009.

[7] D. Micciancio and O. Regev. Lattice-based cryptography. In *Post-Quantum Cryptography*, pages 147–191. Springer, 2009.

[8] C. Peikert. Lattice cryptography for the internet. In *Post-Quantum Cryptography (PQCrypto 2014)*, pages 197–219. Springer, 2014.

[9] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[10] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proc. 28th ACM Symposium on Theory of Computing (STOC)*, pages 212–219, 1996.

[11] G. Brassard, P. Høyer, and A. Tapp. Quantum cryptanalysis of hash and claw-free functions. In *Latin American Symposium on Theoretical Informatics*, pages 163–169. Springer, 1998.

[12] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):1–40, 2009. Preliminary version in STOC 2005.

[13] M. Ajtai. Generating hard instances of lattice problems. In *Proc. 28th ACM Symposium on Theory of Computing (STOC)*, pages 99–108, 1996.

[14] D. Micciancio and O. Regev. Worst-case to average-case reductions based on Gaussian measures. *SIAM Journal on Computing*, 37(1):267–302, 2007.

[15] C. Peikert. Public-key cryptosystems from the worst-case shortest vector problem. In *Proc. 41st ACM Symposium on Theory of Computing (STOC)*, pages 333–342, 2009.

[16] Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé. Classical hardness of learning with errors. In *Proc. 45th ACM Symposium on Theory of Computing (STOC)*, pages 575–584, 2013.

[17] R. Lindner and C. Peikert. Better key sizes (and attacks) for LWE-based encryption. In *Topics in Cryptology – CT-RSA 2011*, pages 319–339, 2011.

[18] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[19] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, 2010.

[20] A. Langlois and D. Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[21] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[22] L. Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[23] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory (ANTS-III)*, pages 267–288, 1998.

[24] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proc. 40th ACM Symposium on Theory of Computing (STOC)*, pages 197–206, 2008.

[25] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology – CRYPTO 1986*, pages 186–194, 1986.

[26] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology – CRYPTO 1999*, pages 537–554, 1999.

[27] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[28] L. Lamport. Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International, 1979.

[29] R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology – CRYPTO 1987*, pages 369–378, 1987.

[30] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber: A CCA-secure module-lattice-based KEM. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367, 2018.

[31] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, 2018.

[32] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang. Falcon: Fast-Fourier lattice-based compact signatures over NTRU. NIST Post-Quantum Cryptography standardization submission, 2020.

[33] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe. The SPHINCS$^+$ signature framework. In *ACM Conference on Computer and Communications Security (CCS)*, pages 2129–2146, 2019.

[34] R. J. McEliece. A public-key cryptosystem based on algebraic coding theory. *JPL Deep Space Network Progress Report*, 42-44:114–116, 1978.

[35] V. D. Goppa. A new class of linear correcting codes. *Problemy Peredachi Informatsii*, 6(3):24–30, 1970.

[36] N. J. Patterson. The algebraic decoding of Goppa codes. *IEEE Transactions on Information Theory*, 21(2):203–207, 1975.

[37] H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15(2):159–166, 1986.

[38] E. R. Berlekamp, R. J. McEliece, and H. C. A. van Tilborg. On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[39] D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, and W. Wang. Classic McEliece. NIST Post-Quantum Cryptography standardization submission, 2017–2022.

[40] C. Aguilar-Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, et al. Hamming Quasi-Cyclic (HQC). NIST Post-Quantum Cryptography standardization submission, 2017–2025.

[41] N. T. Courtois, M. Finiasz, and N. Sendrier. How to achieve a McEliece-based digital signature scheme. In *Advances in Cryptology – ASIACRYPT 2001*, pages 157–174, 2001.

[42] J. Ding, J. E. Gower, and D. S. Schmidt. *Multivariate Public Key Cryptosystems*, volume 25 of *Advances in Information Security*. Springer, 2006.

[43] A. Kipnis, J. Patarin, and L. Goubin. Unbalanced oil and vinegar signature schemes. In *Advances in Cryptology – EUROCRYPT 1999*, pages 206–222, 1999.

[44] W. Beullens. Breaking Rainbow takes a weekend on a laptop. In *Advances in Cryptology – CRYPTO 2022*, pages 464–479, 2022.

[45] D. Jao and L. De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Post-Quantum Cryptography (PQCrypto 2011)*, pages 19–34. Springer, 2011.

[46] W. Castryck and T. Decru. An efficient key recovery attack on SIDH. In *Advances in Cryptology – EUROCRYPT 2023*, pages 423–447, 2023.

[47] L. De Feo, D. Kohel, A. Leroux, C. Petit, and B. Wesolowski. SQIsign: Compact post-quantum signatures from quaternions and isogenies. In *Advances in Cryptology – ASIACRYPT 2020*, pages 64–93, 2020.

[48] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO 1996*, pages 104–113, 1996.

[49] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO 1999*, pages 388–397, 1999.

[50] C. Gidney and S. Schmieg. Tracking the cost of quantum factoring. Google Security Blog, May 2025.

[51] Anonymous. Quantum Horizon: An evaluation of quantum computing as a threat to Bitcoin and Ethereum. arXiv preprint arXiv:2606.14484, 2026 (not peer-reviewed).

[52] A. Milton and C. Shikhelman. Bitcoin and quantum computing: Current status and future directions. Chaincode Labs technical report, 2025.

[53] H. Beast, E. Heilman, and I. Foxen Duke. BIP-360: Pay-to-Merkle-Root (P2MR). Bitcoin Improvement Proposals, Draft, 2024–.

[54] J. Lopp, C. Papathanasiou, I. Smith, J. Ross, S. Vaile, and P.-L. Dallaire-Demers. BIP-361: Post-quantum migration and legacy signature sunset. Bitcoin Improvement Proposals, Draft, 2026–.

[55] V. Buterin. How to hard-fork to save most users' funds in a quantum emergency. ethresear.ch, March 2024.

[56] V. Buterin, S. Wilson, A. Dietrichs, and lightclient. EIP-7702: Set code for EOAs. Ethereum Improvement Proposals, Final (shipped in the Pectra upgrade), 2024–2025.

[57] V. Buterin et al. EIP-8141: Frame transaction. Ethereum Improvement Proposals, Draft, 2026–.

[58] R. Dubois, S. Masson, and Y. H. Lee. EIP-7885: Precompile for NTT operations. Ethereum Improvement Proposals, Draft, 2025–.

[59] ZKNOX. ETHDILITHIUM and ETHFALCON for the Ethereum post-quantum era. ZKNOX technical write-up and `ETHFALCON` repository, 2025.

[60] Ethereum Foundation. Post-quantum cryptography (roadmap: future-proofing). ethereum.org, accessed 2026.