Week 8: Modern Trends in Cryptography

Post-Quantum Cryptography

Post-Quantum Cryptography (PQC) refers to cryptographic algorithms that are secure **against attacks by quantum computers**.

Quantum computers will break many currently used public-key systems:

Classical System	Broken by Quantum Algorithm	Quantum Algorithm
RSA	Yes	Shor's Algorithm
ECC (Elliptic Curve)	Yes	Shor's Algorithm
DH (Diffie–Hellman)	Yes	Shor's Algorithm
AES / SHA	Partially (weakened)	Grover's Algorithm

Symmetric systems like **AES** and **SHA** just need to **double key sizes** to stay secure. But **public-key crypto** needs **new math**.

New Foundations for PQC

Quantum-safe cryptography avoids number-theoretic problems. Instead, it uses:

PQC Type	Hard Problem	Example Algorithms
Lattice-based	Learning With Errors (LWE) / SIS	Kyber, Dilithium, FrodoKEM

Code-based	Decoding random linear codes	Classic McEliece
Multivariate	Solving multivariate polynomials	Rainbow (rejected by NIST)
Hash-based	Hash collisions, preimages	SPHINCS+
Isogeny-based	Supersingular isogeny graphs	SIKE (broken in 2022)

NIST PQC Standardization

NIST (U.S. National Institute of Standards and Technology) is leading global PQC standardization.

Category	Algorithm	Type	Use
Encryption / KEM	**Kyber**	Lattice-based	Public-key encryption, key exchange
Digital Signature	**Dilithium**	Lattice-based	Signatures
Digital Signature	**SPHINCS+**	Hash-based	Stateless, conservative backup

PQC in Practice: Key Concepts

Public-Key Encryption (KEM)

A **Key Encapsulation Mechanism (KEM)** allows two parties to agree on a shared key securely.

- **Kyber** is the leading NIST choice.
- Efficient, small ciphertexts, and fast.

Digital Signatures

Used to verify message authenticity.

- **Dilithium** (fast, lattice-based)
- **SPHINCS+** (based on hashes, more secure but slower/larger)

Hybrid Cryptography

Combines classic (e.g. RSA) and PQC to hedge bets.

- Deployed in **TLS**, **OpenSSH**, and **VPNs**.

Attacks on PQC

While quantum computers can't break PQC (yet), **classical cryptanalysis** still matters.

- **Side-channel attacks** (timing, power analysis)
- **Implementation flaws**
- **Structural weaknesses** in untested schemes (e.g., Rainbow was broken before standardization)

Elliptic Curve Cryptography (ECC)

ECC is a type of **public-key cryptography** based on the algebraic structure of elliptic curves over finite fields. It's used for **encryption, digital signatures, and key exchange** (like ECDSA, ECDH).

Part 1: Mathematical Foundations

1.1 What is an Elliptic Curve?

An **elliptic curve** over real numbers is defined by the equation:

y^2 = x^3 + ax + b

- The curve is **non-singular** if 4a^3 + 27b^2 /=0
- It looks like a smooth, symmetrical curve.

1.2 Points on the Curve

Any pair (x,y) satisfying the equation is a **point on the curve**. There's also a **special point** called the **point at infinity ($\mathcal{O}$)** which acts like a "zero" for point addition.

Part 2: Elliptic Curve Arithmetic

2.1 Point Addition

Given two points P and Q, you can **add** them to get another point R on the curve:

- If P≠Q: Use the line between them.
- If P=Q: Use the tangent to the curve at that point.

Formulas (over real numbers):

Formulas (over real numbers):

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}, \quad x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1$$

2.2 Point Doubling

Same point: R=2P

$$\lambda = \frac{3x^2 + a}{2y}, \quad \text{(other formulas are similar)}$$

2.3 Scalar Multiplication

ECC uses **repeated addition**:

kP=P+P+⋯+P (k times)

This is the core of ECC. It’s **easy to compute** but **hard to reverse** (known as the **Elliptic Curve Discrete Logarithm Problem**, or ECDLP).

Part 3: Finite Fields

3.1 Why Finite Fields?

In real-world cryptography, we don't use real numbers — we use **finite fields** (modulo a prime p) to make things discrete and secure.

An elliptic curve over a finite field Fp is:

$y^2 \bmod p = (x^3 + ax + b) \bmod p$

Part 4: ECC Key Generation & Use

4.1 Key Generation

1. Choose an elliptic curve E and a **base point** G
2. Choose a random private key d
3. Compute public key: $Q=dG$

4.2 Encryption & Decryption (ECIES)

Elliptic Curve Integrated Encryption Scheme:

- Uses key agreement (ECDH) + symmetric encryption

4.3 Digital Signatures (ECDSA)

Elliptic Curve Digital Signature Algorithm:

1. Sign: Use private key to sign a hash
2. Verify: Use public key to verify the signature

Part 5: Why ECC?

Benefits:

- **Stronger security** per bit vs RSA
- **Smaller keys**: 256-bit ECC ~ 3072-bit RSA
- **Faster** computation for equivalent security

Part 6: Practical Curves

Here are common standardized curves:

Name	Field	Size
secp256k1	Fp	256-bit (used in Bitcoin)
secp256r1	Fp	256-bit (NIST P-256)
Curve25519	Fp	255-bit (used in Signal)

IMPLEMENTATION

Step 1: Define the Curve and Field

```
# Elliptic curve parameters over F_p
class Curve:
    def __init__(self, a, b, p):
        self.a = a  # coefficient a
        self.b = b  # coefficient b
        self.p = p  # prime field p

        # Ensure the curve is non-singular
        if (4 * a**3 + 27 * b**2) % p == 0:
            raise ValueError("Singular curve!")

    def is_on_curve(self, point):
        if point is None:
            return True  # Point at infinity
        x, y = point
        return (y**2 - (x**3 + self.a*x + self.b)) % self.p == 0
```

Step 2: Point Addition and Doubling

```
def inverse_mod(k, p):
    """Returns the modular inverse of k modulo p."""
    return pow(k, -1, p)  # Python 3.8+: built-in modular inverse

def point_add(P, Q, curve):
    """Adds two points P and Q on the elliptic curve."""
    if P is None: return Q
    if Q is None: return P
```

```
    x1, y1 = P
    x2, y2 = Q
    p = curve.p

    if x1 == x2 and y1 != y2:
        return None  # Point at infinity

    if P == Q:
        # Point doubling
        m = (3 * x1 * x1 + curve.a) * inverse_mod(2 * y1, p)
    else:
        # Point addition
        m = (y2 - y1) * inverse_mod(x2 - x1, p)

    m %= p
    x3 = (m * m - x1 - x2) % p
    y3 = (m * (x1 - x3) - y1) % p
    return (x3, y3)
```

Step 3: Scalar Multiplication

```
def scalar_mult(k, P, curve):
    """Performs scalar multiplication k * P."""
    result = None  # Start with point at infinity
    addend = P

    while k:
        if k & 1:
            result = point_add(result, addend, curve)
        addend = point_add(addend, addend, curve)
        k >>= 1

    return result
```

Step 4: Key Generation

```
import random

def generate_keypair(G, curve, n):
    """Generates a private/public key pair."""
    private_key = random.randint(1, n - 1)
    public_key = scalar_mult(private_key, G, curve)
    return private_key, public_key
```