Elliptic Curve Cryptography: A Comprehensive Guide

From Mathematical Theory to C++17 Implementation

CryptoGL Educational Series

July 31, 2025

Abstract

This document provides a comprehensive introduction to Elliptic Curve Cryptography (ECC), a modern approach to public-key cryptography that offers high security with smaller key sizes compared to traditional methods like RSA. We begin with the mathematical foundations of elliptic curves, including group operations and discrete logarithm problems. We then present step-by-step practical examples of ECC key generation, encryption, and digital signatures, followed by efficient C++17 implementation strategies. This guide is designed for beginners with basic mathematical knowledge who want to understand both the theory and practical implementation of ECC.

Contents

Introduction to Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is a modern approach to public-key cryptography that uses the mathematics of elliptic curves over finite fields. ECC was independently proposed by Neal Koblitz and Victor S. Miller in 1985 and has become increasingly popular due to its efficiency and security properties.

Elliptic Curve Cryptography: A public-key cryptographic system based on the algebraic structure of elliptic curves over finite fields. ECC provides the same level of security as RSA but with much smaller key sizes.

Key Advantages of ECC

- **Security:** 256-bit ECC keys provide equivalent security to 3072-bit RSA keys
- **Efficiency:** Faster computation and lower power consumption
- **Key Size:** Smaller key sizes reduce storage and transmission overhead
- **Scalability:** Better performance on resource-constrained devices

Mathematical Foundations

Elliptic Curves

Definition

Definition 2.1. *An **elliptic curve** over a field K is the set of points $(x, y) \in K \times K$ that satisfy the Weierstrass equation:*

$$y^2 = x^3 + ax + b$$

where $a, b \in K$ and the discriminant $\Delta = -16(4a^3 + 27b^2) \neq 0$.

Example: The curve $y^2 = x^3 - x + 1$ over $\mathbb{R}$

- Points: $(0, 1), (0, -1), (1, 1), (1, -1), (-1, 1), (-1, -1), \ldots$
- The curve is smooth (no singular points)
- Has infinitely many points over $\mathbb{R}$

Point Addition

The key insight of ECC is that we can define a group operation on the points of an elliptic curve.

Point Addition Algorithm:

Input: Points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ on curve E
Output: Point $R = P + Q$
if $P = \mathcal{O}$ (point at infinity) **then**
  return Q
else if $Q = \mathcal{O}$ **then**
  return P
else if $x_1 = x_2$ and $y_1 = -y_2$ **then**
  return $\mathcal{O}$
else
  $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$ (if $P \neq Q$)
  $\lambda = \frac{3x_1^2 + a}{2y_1}$ (if $P = Q$)
  $x_3 = \lambda^2 - x_1 - x_2$
  $y_3 = \lambda(x_1 - x_3) - y_1$
  return (x_3, y_3)
end if

Elliptic Curves over Finite Fields

For cryptographic applications, we work with elliptic curves over finite fields $\mathbb{F}_p$ where p is a prime.

Definition 2.2. *An elliptic curve over $\mathbb{F}_p$ is the set of points $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ satisfying:*

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

where $a, b \in \mathbb{F}_p$ and $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$.

Example: Curve $y^2 = x^3 + 2x + 3$ over $\mathbb{F}_7$

Points: $(0, 2), (0, 5), (1, 1), (1, 6), (2, 1), (2, 6),$
$(3, 1), (3, 6), (4, 2), (4, 5), (5, 0), (6, 2), (6, 5), \mathcal{O}$

Total: 14 points (including point at infinity)

Discrete Logarithm Problem

The security of ECC relies on the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP).

Definition 2.3. *Given points P and $Q = kP$ on an elliptic curve, the **Elliptic Curve Discrete Logarithm Problem** is to find the integer k such that $Q = kP$.*

Theorem 2.4. *The ECDLP is computationally hard for well-chosen curves and large prime fields.*

ECC Key Generation

Key Generation Process

1. Choose a prime p and curve parameters a, b

2. Find a point G (generator) with large prime order n
3. Choose a private key d randomly from $[1, n-1]$
4. Compute the public key $Q = dG$

ECC Key Components:

$$\text{Public Parameters} = (p, a, b, G, n)$$
$$\text{Private Key} = d \in [1, n-1]$$
$$\text{Public Key} = Q = dG$$

Step-by-Step Practical Examples

Example 1: Small Field for Understanding

Let's work through a simple example with a small field.

Step 1: Choose Curve Parameters

- Prime field: $p = 17$
- Curve: $y^2 = x^3 + 2x + 3$ over $\mathbb{F}_{17}$
- Generator point: $G = (5, 1)$
- Order of G: $n = 19$

Step 2: Key Generation

- Choose private key: $d = 7$
- Compute public key: $Q = 7G$

Step 3: Point Multiplication Compute $7G$ using double-and-add:

$$G = (5, 1)$$
$$2G = G + G = (6, 3)$$
$$4G = 2G + 2G = (10, 6)$$
$$7G = 4G + 2G + G = (3, 1)$$

Keys:

$$\text{Private Key} = d = 7$$
$$\text{Public Key} = Q = (3, 1)$$

Example 2: ECDH Key Exchange

ECDH Key Exchange:

- Alice chooses private key $a = 3$, computes $A = 3G = (10, 6)$
- Bob chooses private key $b = 5$, computes $B = 5G = (6, 3)$
- Alice computes shared secret: $S = aB = 3(6, 3) = (3, 1)$
- Bob computes shared secret: $S = bA = 5(10, 6) = (3, 1)$
- Both parties now have the same shared secret point

Efficient Implementation in C++17

Basic ECC Class Structure

```cpp
#include <iostream>
#include <vector>
#include <random>
#include <cstdint>

struct Point {
    uint64_t x, y;
    bool is_infinity;

    Point() : x(0), y(0), is_infinity(true) {}
    Point(uint64_t x, uint64_t y) : x(x), y(y), is_infinity(false) {}

    static Point infinity() { return Point(); }
};

class ECC {
private:
    uint64_t p;  // prime field
    uint64_t a, b;  // curve parameters
    Point G;  // generator point
    uint64_t n;  // order of G
    uint64_t d;  // private key
    Point Q;  // public key

    // Helper functions
    uint64_t mod_inverse(uint64_t a, uint64_t m);
    uint64_t mod_pow(uint64_t base, uint64_t exp, uint64_t mod);
    Point point_add(const Point& P, const Point& Q);
    Point point_double(const Point& P);
    Point scalar_multiply(const Point& P, uint64_t k);
    bool is_on_curve(const Point& P);

public:
    ECC(uint64_t p, uint64_t a, uint64_t b, Point G, uint64_t n);
    void generate_keys();
    Point get_public_key() const;
    Point ecdh_shared_secret(const Point& other_public_key) const;
};
```

Listing 1: Basic ECC Class Structure

Mathematical Helper Functions

```cpp
// Modular inverse using extended Euclidean algorithm
uint64_t ECC::mod_inverse(uint64_t a, uint64_t m) {
    int64_t m0 = m, t, q;
    int64_t x0 = 0, x1 = 1;

    if (m == 1) return 0;

    while (a > 1) {
        q = a / m;
        t = m;
        m = a % m;
        a = t;
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
    }

    if (x1 < 0) x1 += m0;
    return x1;
}

// Fast modular exponentiation
uint64_t ECC::mod_pow(uint64_t base, uint64_t exp, uint64_t mod) {
    uint64_t result = 1;
    base = base % mod;

    while (exp > 0) {
        if (exp & 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp >>= 1;
    }
    return result;
}
```

Listing 2: Mathematical Helper Functions

Point Operations

```cpp
// Point addition on elliptic curve
Point ECC::point_add(const Point& P, const Point& Q) {
    if (P.is_infinity) return Q;
    if (Q.is_infinity) return P;

    uint64_t lambda;
    if (P.x == Q.x && P.y == Q.y) {
        // Point doubling
        if (P.y == 0) return Point::infinity();
        lambda = ((3 * mod_pow(P.x, 2, p) + a) *
                  mod_inverse(2 * P.y, p)) % p;
    } else {
        // Point addition
        if (P.x == Q.x) return Point::infinity();
        lambda = ((Q.y - P.y + p) *
                  mod_inverse(Q.x - P.x + p, p)) % p;
    }

    uint64_t x3 = (mod_pow(lambda, 2, p) - P.x - Q.x + 2 * p) % p;
    uint64_t y3 = (lambda * (P.x - x3 + p) - P.y + p) % p;
```

```
21
22      return Point(x3, y3);
23  }
24
25  // Scalar multiplication using double-and-add
26  Point ECC::scalar_multiply(const Point& P, uint64_t k) {
27      Point result = Point::infinity();
28      Point temp = P;
29
30      while (k > 0) {
31          if (k & 1) {
32              result = point_add(result, temp);
33          }
34          temp = point_add(temp, temp);
35          k >>= 1;
36      }
37      return result;
38  }
```

Listing 3: Point Operations

Key Generation and ECDH

```
1   // Constructor
2   ECC::ECC(uint64_t p, uint64_t a, uint64_t b, Point G, uint64_t n)
3       : p(p), a(a), b(b), G(G), n(n) {
4       generate_keys();
5   }
6
7   // Generate key pair
8   void ECC::generate_keys() {
9       std::random_device rd;
10      std::mt19937_64 gen(rd());
11      std::uniform_int_distribution<uint64_t> dis(1, n - 1);
12
13      d = dis(gen);  // private key
14      Q = scalar_multiply(G, d);  // public key
15  }
16
17  // Get public key
18  Point ECC::get_public_key() const {
19      return Q;
20  }
21
22  // ECDH shared secret
23  Point ECC::ecdh_shared_secret(const Point& other_public_key) const {
24      return scalar_multiply(other_public_key, d);
25  }
```

Listing 4: Key Generation and ECDH

Complete Example Usage

```
1   int main() {
2       // Curve parameters for secp256k1 (simplified)
3       uint64_t p = 17;  // Small prime for demonstration
4       uint64_t a = 2, b = 3;
5       Point G(5, 1);  // Generator point
6       uint64_t n = 19;  // Order of G
7
8       // Create ECC instances for Alice and Bob
```

```
    ECC alice(p, a, b, G, n);
    ECC bob(p, a, b, G, n);

    // Get public keys
    Point alice_pub = alice.get_public_key();
    Point bob_pub = bob.get_public_key();

    std::cout << "Alice's public key: (" << alice_pub.x
              << ", " << alice_pub.y << ")\n";
    std::cout << "Bob's public key: (" << bob_pub.x
              << ", " << bob_pub.y << ")\n";

    // Compute shared secrets
    Point alice_secret = alice.ecdh_shared_secret(bob_pub);
    Point bob_secret = bob.ecdh_shared_secret(alice_pub);

    std::cout << "Shared secret: (" << alice_secret.x
              << ", " << alice_secret.y << ")\n";

    // Verify they match
    if (alice_secret.x == bob_secret.x &&
        alice_secret.y == bob_secret.y) {
        std::cout << "ECDH successful!\n";
    }

    return 0;
}
```

Listing 5: Complete ECC Example

ECDSA Digital Signatures

ECDSA Algorithm

ECDSA Signature Generation:

Input: Message m, private key d, curve parameters
Output: Signature (r, s)
Choose random $k \in [1, n-1]$
Compute $R = kG = (x_R, y_R)$
$r = x_R \bmod n$
if $r = 0$ **then**
  Go to step 1
end if
Compute $e = H(m)$ (hash of message)
$s = k^{-1}(e + dr) \bmod n$
if $s = 0$ **then**
  Go to step 1
end if
return (r, s)

ECDSA Signature Verification:

Input: Message m, signature (r, s), public key Q
Output: Valid/Invalid
Verify $r, s \in [1, n-1]$
Compute $e = H(m)$
$w = s^{-1} \bmod n$
$u_1 = ew \bmod n$
$u_2 = rw \bmod n$
$X = u_1 G + u_2 Q$
if $X = \mathcal{O}$ **then**
  return Invalid
end if
$v = x_X \bmod n$
return $v = r$

ECDSA Implementation

```
class ECDSA {
private:
    ECC ecc;

    uint64_t hash_message(const std::string& message) {
        // Simple hash function for demonstration
        uint64_t hash = 0;
        for (char c : message) {
            hash = (hash * 31 + c) % ecc.get_n();
        }
        return hash;
    }

public:
    ECDSA(const ECC& ecc) : ecc(ecc) {}

    std::pair<uint64_t, uint64_t> sign(const std::string& message) {
        uint64_t n = ecc.get_n();
        uint64_t e = hash_message(message);

        std::random_device rd;
        std::mt19937_64 gen(rd());
        std::uniform_int_distribution<uint64_t> dis(1, n - 1);

        uint64_t k, r, s;
        do {
            k = dis(gen);
            Point R = ecc.scalar_multiply(ecc.get_G(), k);
            r = R.x % n;
        } while (r == 0);

        uint64_t k_inv = ecc.mod_inverse(k, n);
        s = (k_inv * (e + ecc.get_d() * r)) % n;

        return {r, s};
    }

    bool verify(const std::string& message,
                const std::pair<uint64_t, uint64_t>& signature,
                const Point& public_key) {
        uint64_t r = signature.first, s = signature.second;
```

```
        uint64_t n = ecc.get_n();

        if (r < 1 || r >= n || s < 1 || s >= n) return false;

        uint64_t e = hash_message(message);
        uint64_t w = ecc.mod_inverse(s, n);
        uint64_t u1 = (e * w) % n;
        uint64_t u2 = (r * w) % n;

        Point X = ecc.point_add(
            ecc.scalar_multiply(ecc.get_G(), u1),
            ecc.scalar_multiply(public_key, u2)
        );

        if (X.is_infinity) return false;

        uint64_t v = X.x % n;
        return v == r;
    }
};
```

Listing 6: ECDSA Implementation

Security Considerations

Curve Selection

Important: Always use standardized curves that have been extensively analyzed for security.

Curve	Field Size	Security Level
secp256k1	256 bits	128 bits
secp384r1	384 bits	192 bits
secp521r1	521 bits	256 bits

Table 1: Standard ECC Curves

Common Attacks and Mitigations

- **Pollard's Rho:** Mitigated by using large prime fields
- **Small Subgroup Attacks:** Mitigated by verifying point order
- **Timing Attacks:** Mitigated by constant-time implementations
- **Invalid Curve Attacks:** Mitigated by validating curve parameters

Performance Considerations

Optimization Techniques

1. **NAF (Non-Adjacent Form):** Reduces number of additions in scalar multiplication
2. **Window Methods:** Precompute multiples for faster multiplication
3. **Montgomery Ladder:** Constant-time scalar multiplication

4. **Projective Coordinates:** Avoid modular inversions

Comparison with RSA

Algorithm	Key Size	Performance
RSA-2048	2048 bits	Slower
ECC-256	256 bits	Faster

Table 2: ECC vs RSA Comparison

Real-World Applications

Applications

- **SSL/TLS:** Secure web communications
- **Bitcoin:** Digital signatures for transactions
- **Smart Cards:** Resource-constrained devices
- **Mobile Devices:** Efficient cryptography

Conclusion

Elliptic Curve Cryptography represents a significant advancement in public-key cryptography, offering superior security and efficiency compared to traditional methods.

Advantages of ECC:

- Higher security per bit of key length
- Faster computation and lower power consumption
- Smaller key sizes reduce storage requirements
- Better performance on constrained devices

Limitations of ECC:

- More complex implementation
- Requires careful parameter selection
- Vulnerable to quantum attacks (Shor's algorithm)
- Less standardized than RSA

Future Considerations

Quantum Threat: Like RSA, ECC is vulnerable to quantum attacks. Post-quantum cryptography research is essential for long-term security.

Further Reading

- "Guide to Elliptic Curve Cryptography" by Hankerson, Menezes, and Vanstone
- "Elliptic Curves: Number Theory and Cryptography" by Lawrence Washington
- NIST Special Publication 800-186: Recommendations for Discrete Logarithm-Based Cryptography
- RFC 6090: Fundamental Elliptic Curve Cryptography Algorithms