RSA 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 RSA (Rivest-Shamir-Adleman) cryptography, one of the most widely used asymmetric encryption algorithms. We begin with the mathematical foundations, including number theory concepts essential for understanding RSA. We then present step-by-step practical examples of RSA encryption and decryption, 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 RSA cryptography.

Contents

1 Introduction to RSA Cryptography **3**
- 1.1 Key Concepts . . . 3

2 Mathematical Foundations **3**
- 2.1 Number Theory Prerequisites . . . 3
  - 2.1.1 Prime Numbers . . . 3
  - 2.1.2 Greatest Common Divisor (GCD) . . . 3
  - 2.1.3 Modular Arithmetic . . . 4
  - 2.1.4 Euler's Totient Function . . . 4
  - 2.1.5 Fermat's Little Theorem . . . 4
  - 2.1.6 Euler's Theorem . . . 4
- 2.2 Extended Euclidean Algorithm . . . 5

3 RSA Algorithm **5**
- 3.1 Key Generation . . . 5
- 3.2 Encryption and Decryption . . . 6
- 3.3 Mathematical Proof of Correctness . . . 6

4 Step-by-Step Practical Examples **6**
- 4.1 Example 1: Small Numbers for Understanding . . . 6
- 4.2 Example 2: Larger Numbers . . . 8

5 Efficient Implementation in C++17 **8**
5.1 Mathematical Optimizations . . . 8
5.1.1 Fast Modular Exponentiation . . . 8
5.1.2 Chinese Remainder Theorem (CRT) . . . 9
5.2 C++17 Implementation . . . 9
5.2.1 Basic RSA Class Structure . . . 9
5.2.2 Mathematical Helper Functions . . . 10
5.2.3 Prime Number Generation . . . 10
5.2.4 RSA Key Generation . . . 11
5.2.5 Encryption and Decryption . . . 12
5.2.6 Complete Example Usage . . . 12
5.3 Advanced Optimizations . . . 13
5.3.1 Chinese Remainder Theorem Implementation . . . 13
5.3.2 Big Integer Support . . . 13

6 Security Considerations **14**
6.1 Key Size Requirements . . . 14
6.2 Common Attacks and Mitigations . . . 14
6.2.1 Factorization Attacks . . . 14
6.2.2 Timing Attacks . . . 15
6.2.3 Chosen Ciphertext Attacks . . . 15
6.3 Padding Schemes . . . 15
6.3.1 PKCS#1 v1.5 Padding . . . 15

7 Performance Considerations **15**
7.1 Computational Complexity . . . 15
7.2 Optimization Techniques . . . 16

8 Real-World Applications **16**
8.1 Digital Signatures . . . 16
8.2 Hybrid Encryption . . . 16

9 Conclusion **16**
9.1 Future Considerations . . . 17
9.2 Further Reading . . . 17

A Mathematical Proofs **17**
A.1 Proof of Euler's Theorem . . . 17

B Code Examples **18**
B.1 Complete RSA Implementation . . . 18
B.2 Testing RSA Implementation . . . 18

Introduction to RSA Cryptography

RSA is an asymmetric cryptographic algorithm named after its creators: Ron Rivest, Adi Shamir, and Leonard Adleman, who first publicly described it in 1977. It is based on the mathematical difficulty of factoring the product of two large prime numbers.

> **RSA Cryptography:** An asymmetric cryptographic system that uses a pair of mathematically related keys - a public key for encryption and a private key for decryption. The security of RSA relies on the computational difficulty of factoring large composite numbers.

Key Concepts

- **Asymmetric Encryption:** Uses different keys for encryption and decryption
- **Public Key:** Can be freely shared and used by anyone to encrypt messages
- **Private Key:** Must be kept secret and is used for decryption
- **Digital Signatures:** Can also be used to create digital signatures

Mathematical Foundations

Number Theory Prerequisites

Prime Numbers

Definition 2.1. *A **prime number** is a natural number greater than 1 that has no positive divisors other than 1 and itself.*

> **Examples of Prime Numbers:**
>
> - 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97
> - For RSA, we typically use primes with hundreds or thousands of digits

Greatest Common Divisor (GCD)

Definition 2.2. *The **greatest common divisor** of two integers a and b, denoted $\gcd(a, b)$, is the largest positive integer that divides both a and b.*

> **Euclidean Algorithm for GCD:**
>
> **Input:** Two positive integers a and b
> **Output:** $\gcd(a, b)$
> **while** $b \neq 0$ **do**
> $r \leftarrow a \bmod b$
> $a \leftarrow b$
> $b \leftarrow r$
> **end while**
> **return** a

Example: Find $\gcd(48, 18)$

$$\begin{aligned}
48 &= 2 \times 18 + 12 \\
18 &= 1 \times 12 + 6 \\
12 &= 2 \times 6 + 0
\end{aligned}$$

Therefore, $\gcd(48, 18) = 6$

Modular Arithmetic

Definition 2.3. *For integers a, b, and $n > 0$, we say $a \equiv b \pmod{n}$ if n divides $(a - b)$. This means a and b have the same remainder when divided by n.*

Properties of Modular Arithmetic:

$$\begin{aligned}
(a + b) \bmod n &= ((a \bmod n) + (b \bmod n)) \bmod n \\
(a \times b) \bmod n &= ((a \bmod n) \times (b \bmod n)) \bmod n \\
(a^b) \bmod n &= ((a \bmod n)^b) \bmod n
\end{aligned}$$

Euler's Totient Function

Definition 2.4. *Euler's totient function $\phi(n)$ counts the number of integers between 1 and n that are coprime to n (i.e., $\gcd(k, n) = 1$ for $1 \le k \le n$).*

Theorem 2.5. *If $n = pq$ where p and q are distinct prime numbers, then:*

$$\phi(n) = \phi(pq) = (p - 1)(q - 1)$$

Example: Let $p = 3$ and $q = 5$, then $n = 15$

$$\phi(15) = \phi(3 \times 5) = (3 - 1)(5 - 1) = 2 \times 4 = 8$$

The numbers coprime to 15 are: 1, 2, 4, 7, 8, 11, 13, 14 (8 numbers)

Fermat's Little Theorem

Theorem 2.6 (Fermat's Little Theorem)**.** *If p is a prime number and a is an integer not divisible by p, then:*

$$a^{p-1} \equiv 1 \pmod{p}$$

Euler's Theorem

Theorem 2.7 (Euler's Theorem)**.** *If $\gcd(a, n) = 1$, then:*

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Extended Euclidean Algorithm

The extended Euclidean algorithm not only finds $\gcd(a, b)$ but also finds integers x and y such that:

$$ax + by = \gcd(a, b)$$

Extended Euclidean Algorithm:

Input: Two positive integers a and b
Output: $\gcd(a, b)$ and integers x, y such that $ax + by = \gcd(a, b)$

```
x_0, x_1 ← 1, 0
y_0, y_1 ← 0, 1
while b ≠ 0 do
    q ← ⌊a/b⌋
    r ← a mod b
    x_2 ← x_0 − q × x_1
    y_2 ← y_0 − q × y_1
    a, b ← b, r
    x_0, x_1 ← x_1, x_2
    y_0, y_1 ← y_1, y_2
end while
return (a, x_0, y_0)
```

RSA Algorithm

Key Generation

The RSA key generation process involves the following steps:

1. Choose two distinct prime numbers p and q
2. Compute $n = pq$ (the modulus)
3. Compute $\phi(n) = (p - 1)(q - 1)$
4. Choose an integer e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$
5. Compute d such that $ed \equiv 1 \pmod{\phi(n)}$ (using extended Euclidean algorithm)

RSA Key Components:

$$\begin{aligned}\text{Public Key} &= (n, e) \\ \text{Private Key} &= (n, d) \\ \text{where } ed &\equiv 1 \pmod{\phi(n)}\end{aligned}$$

Encryption and Decryption

RSA Encryption: For a message M (where $0 \leq M < n$):

$$C = M^e \bmod n$$

RSA Decryption: For a ciphertext C:

$$M = C^d \bmod n$$

Mathematical Proof of Correctness

Theorem 3.1. *If $C = M^e \bmod n$ and $M = C^d \bmod n$, then the RSA algorithm correctly encrypts and decrypts messages.*

Proof. We need to prove that $(M^e)^d \equiv M \pmod{n}$.

Since $ed \equiv 1 \pmod{\phi(n)}$, we have $ed = 1 + k\phi(n)$ for some integer k.

Therefore:

$$(M^e)^d = M^{ed} = M^{1+k\phi(n)} = M \times M^{k\phi(n)}$$

By Euler's theorem, if $\gcd(M, n) = 1$, then $M^{\phi(n)} \equiv 1 \pmod{n}$, so:

$$M^{k\phi(n)} \equiv 1^k \equiv 1 \pmod{n}$$

Thus:

$$(M^e)^d \equiv M \times 1 \equiv M \pmod{n}$$

For the case where $\gcd(M, n) \neq 1$, the proof is more complex but still holds. □

Step-by-Step Practical Examples

Example 1: Small Numbers for Understanding

Let's work through a simple example with small numbers to understand the process.

Step 1: Key Generation

- Choose $p = 3$ and $q = 11$
- Compute $n = p \times q = 3 \times 11 = 33$
- Compute $\phi(n) = (p-1)(q-1) = 2 \times 10 = 20$
- Choose $e = 3$ (since $\gcd(3, 20) = 1$)
- Find d such that $3d \equiv 1 \pmod{20}$

Step 2: Finding the Private Key Using extended Euclidean algorithm to find d:

$$\begin{aligned} 20 &= 6 \times 3 + 2 \\ 3 &= 1 \times 2 + 1 \\ 2 &= 2 \times 1 + 0 \end{aligned}$$

Working backwards:

$$\begin{aligned} 1 &= 3 - 1 \times 2 \\ &= 3 - 1 \times (20 - 6 \times 3) \\ &= 7 \times 3 - 1 \times 20 \end{aligned}$$

Therefore, $d = 7$ (since $3 \times 7 = 21 \equiv 1 \pmod{20}$)
Keys:

$$\begin{aligned} \text{Public Key} &= (n = 33, e = 3) \\ \text{Private Key} &= (n = 33, d = 7) \end{aligned}$$

Step 3: Encryption Let's encrypt the message $M = 5$:

$$C = 5^3 \bmod 33 = 125 \bmod 33 = 26$$

Step 4: Decryption Let's decrypt the ciphertext $C = 26$:

$$M = 26^7 \bmod 33 = 8031810176 \bmod 33 = 5$$

The original message is recovered!

Example 2: Larger Numbers

Step 1: Key Generation

- Choose $p = 61$ and $q = 53$
- Compute $n = p \times q = 61 \times 53 = 3233$
- Compute $\phi(n) = (p-1)(q-1) = 60 \times 52 = 3120$
- Choose $e = 17$ (since $\gcd(17, 3120) = 1$)
- Find d such that $17d \equiv 1 \pmod{3120}$

Step 2: Finding the Private Key Using extended Euclidean algorithm:

$$\begin{aligned} 3120 &= 183 \times 17 + 9 \\ 17 &= 1 \times 9 + 8 \\ 9 &= 1 \times 8 + 1 \\ 8 &= 8 \times 1 + 0 \end{aligned}$$

Working backwards:

$$\begin{aligned} 1 &= 9 - 1 \times 8 \\ &= 9 - 1 \times (17 - 1 \times 9) \\ &= 2 \times 9 - 1 \times 17 \\ &= 2 \times (3120 - 183 \times 17) - 1 \times 17 \\ &= 2 \times 3120 - 367 \times 17 \end{aligned}$$

Therefore, $d = -367 \equiv 2753 \pmod{3120}$
Keys:

$$\begin{aligned} \text{Public Key} &= (n = 3233, e = 17) \\ \text{Private Key} &= (n = 3233, d = 2753) \end{aligned}$$

Step 3: Encryption Let's encrypt the message $M = 123$:

$$C = 123^{17} \bmod 3233 = 855$$

Step 4: Decryption Let's decrypt the ciphertext $C = 855$:

$$M = 855^{2753} \bmod 3233 = 123$$

The original message is recovered!

Efficient Implementation in C++17

Mathematical Optimizations

Fast Modular Exponentiation

The naive approach of computing $a^b \bmod n$ by first computing a^b and then taking the modulus is computationally infeasible for large numbers. We use the square-and-multiply algorithm (also known as binary exponentiation).

Square-and-Multiply Algorithm:

Input: a, b, n
Output: $a^b \bmod n$
$result \leftarrow 1$
$base \leftarrow a \bmod n$
while $b > 0$ **do**
  if b is odd **then**
    $result \leftarrow (result \times base) \bmod n$
  end if
  $base \leftarrow (base \times base) \bmod n$
  $b \leftarrow b \div 2$
end while
return $result$

Chinese Remainder Theorem (CRT)

For decryption, we can use the Chinese Remainder Theorem to speed up computation:

Theorem 5.1 (Chinese Remainter Theorem)**.** *If $n = pq$ where p and q are coprime, and we know $M \bmod p$ and $M \bmod q$, then we can efficiently compute $M \bmod n$.*

For RSA decryption with CRT:

$$M_p = C^{d_p} \bmod p \text{ where } d_p = d \bmod (p-1)$$
$$M_q = C^{d_q} \bmod q \text{ where } d_q = d \bmod (q-1)$$
$$M = M_p + p \times ((M_q - M_p) \times p^{-1} \bmod q)$$

C++17 Implementation

Basic RSA Class Structure

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

class RSA {
private:
    // Key components
    uint64_t n;  // modulus
    uint64_t e;  // public exponent
    uint64_t d;  // private exponent
    uint64_t p;  // first prime
    uint64_t q;  // second prime

    // Helper functions
    uint64_t gcd(uint64_t a, uint64_t b);
    uint64_t mod_inverse(uint64_t a, uint64_t m);
    uint64_t mod_pow(uint64_t base, uint64_t exp, uint64_t mod);
    bool is_prime(uint64_t n);
    uint64_t generate_prime(uint64_t min, uint64_t max);

public:
```

```
    RSA(uint64_t bit_length = 1024);
    uint64_t encrypt(uint64_t message);
    uint64_t decrypt(uint64_t ciphertext);
    std::pair<uint64_t, uint64_t> get_public_key() const;
};
```

Listing 1: Basic RSA Class Structure

Mathematical Helper Functions

```
// Greatest Common Divisor using Euclidean algorithm
uint64_t RSA::gcd(uint64_t a, uint64_t b) {
    while (b != 0) {
        uint64_t temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

// Extended Euclidean Algorithm for modular inverse
uint64_t RSA::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 using square-and-multiply
uint64_t RSA::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

Prime Number Generation

```cpp
// Miller-Rabin primality test
bool RSA::is_prime(uint64_t n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0) return false;

    // Write n as 2^r * d + 1
    uint64_t d = n - 1;
    uint64_t r = 0;
    while (d % 2 == 0) {
        d /= 2;
        r++;
    }

    // Test with first few prime bases
    std::vector<uint64_t> bases = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};

    for (uint64_t base : bases) {
        if (base >= n) continue;

        uint64_t x = mod_pow(base, d, n);
        if (x == 1 || x == n - 1) continue;

        bool is_composite = true;
        for (uint64_t i = 1; i < r; i++) {
            x = (x * x) % n;
            if (x == n - 1) {
                is_composite = false;
                break;
            }
        }
        if (is_composite) return false;
    }
    return true;
}

// Generate a random prime number
uint64_t RSA::generate_prime(uint64_t min, uint64_t max) {
    std::random_device rd;
    std::mt19937_64 gen(rd());
    std::uniform_int_distribution<uint64_t> dis(min, max);

    uint64_t candidate;
    do {
        candidate = dis(gen);
        // Ensure odd number
        if (candidate % 2 == 0) candidate++;
    } while (!is_prime(candidate));

    return candidate;
}
```

Listing 3: Prime Number Generation

RSA Key Generation

```cpp
// Constructor with key generation
RSA::RSA(uint64_t bit_length) {
    uint64_t min_prime = 1ULL << (bit_length / 2 - 1);
    uint64_t max_prime = 1ULL << (bit_length / 2);

    // Generate two large prime numbers
```

```
    p = generate_prime(min_prime, max_prime);
    do {
        q = generate_prime(min_prime, max_prime);
    } while (q == p);

    // Calculate modulus
    n = p * q;

    // Calculate Euler's totient function
    uint64_t phi_n = (p - 1) * (q - 1);

    // Choose public exponent (commonly 65537)
    e = 65537;
    while (gcd(e, phi_n) != 1) {
        e += 2;
    }

    // Calculate private exponent
    d = mod_inverse(e, phi_n);
}
```

Listing 4: RSA Key Generation

Encryption and Decryption

```
// Encrypt a message
uint64_t RSA::encrypt(uint64_t message) {
    if (message >= n) {
        throw std::invalid_argument("Message too large for current key size");
    }
    return mod_pow(message, e, n);
}

// Decrypt a ciphertext
uint64_t RSA::decrypt(uint64_t ciphertext) {
    return mod_pow(ciphertext, d, n);
}

// Get public key
std::pair<uint64_t, uint64_t> RSA::get_public_key() const {
    return {n, e};
}
```

Listing 5: Encryption and Decryption

Complete Example Usage

```
#include <iostream>
#include <string>

int main() {
    try {
        // Create RSA instance with 1024-bit keys
        RSA rsa(1024);

        // Get public key
        auto [n, e] = rsa.get_public_key();
        std::cout << "Public Key (n, e): (" << n << ", " << e << ")\n";

        // Example message
        uint64_t message = 12345;
```

```cpp
        std::cout << "Original message: " << message << "\n";

        // Encrypt
        uint64_t encrypted = rsa.encrypt(message);
        std::cout << "Encrypted: " << encrypted << "\n";

        // Decrypt
        uint64_t decrypted = rsa.decrypt(encrypted);
        std::cout << "Decrypted: " << decrypted << "\n";

        // Verify
        if (message == decrypted) {
            std::cout << "RSA encryption/decryption successful!\n";
        } else {
            std::cout << "Error in RSA implementation!\n";
        }

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }

    return 0;
}
```

Listing 6: Complete RSA Example

Advanced Optimizations

Chinese Remainder Theorem Implementation

```cpp
// CRT-optimized decryption
uint64_t RSA::decrypt_crt(uint64_t ciphertext) {
    // Precompute CRT parameters
    uint64_t d_p = d % (p - 1);
    uint64_t d_q = d % (q - 1);
    uint64_t q_inv = mod_inverse(q, p);

    // Compute using CRT
    uint64_t m_p = mod_pow(ciphertext, d_p, p);
    uint64_t m_q = mod_pow(ciphertext, d_q, q);

    // Combine using CRT
    uint64_t h = (q_inv * (m_p - m_q)) % p;
    uint64_t m = m_q + h * q;

    return m;
}
```

Listing 7: CRT-Optimized Decryption

Big Integer Support

For production use, you'll need arbitrary-precision arithmetic. Here's how to integrate with a big integer library:

```cpp
#include "big_integers/BigIntegerLibrary.hh"

class BigIntRSA {
private:
    BigInteger n, e, d, p, q;

    BigInteger mod_pow(const BigInteger& base, const BigInteger& exp,
```

```
                      const BigInteger& mod) {
        BigInteger result = 1;
        BigInteger base_mod = base % mod;
        BigInteger exp_copy = exp;

        while (exp_copy > 0) {
            if (exp_copy % 2 == 1) {
                result = (result * base_mod) % mod;
            }
            base_mod = (base_mod * base_mod) % mod;
            exp_copy /= 2;
        }
        return result;
    }

public:
    BigIntRSA(int bit_length = 2048) {
        // Generate large primes using a proper big integer library
        // This is a simplified version - in practice, use established libraries
        // like GMP, OpenSSL, or Crypto++ for secure prime generation
    }

    BigInteger encrypt(const BigInteger& message) {
        return mod_pow(message, e, n);
    }

    BigInteger decrypt(const BigInteger& ciphertext) {
        return mod_pow(ciphertext, d, n);
    }
};
```

Listing 8: Big Integer RSA Implementation

Security Considerations

Key Size Requirements

Important: The security of RSA depends on the difficulty of factoring the modulus n. As computational power increases, larger key sizes are required.

Key Size (bits)	Security Level	Recommended Until
1024	80 bits	2010
2048	112 bits	2030
3072	128 bits	2040
4096	192 bits	2050+

Table 1: RSA Key Size Recommendations

Common Attacks and Mitigations

Factorization Attacks

- **General Number Field Sieve (GNFS):** Most efficient known algorithm for factoring large integers
- **Quadratic Sieve:** Effective for numbers up to about 100 digits

- **Mitigation:** Use sufficiently large key sizes (2048+ bits)

Timing Attacks

- **Attack:** Measure time taken for encryption/decryption to infer information about the private key
- **Mitigation:** Use constant-time implementations and blinding techniques

Chosen Ciphertext Attacks

- **Attack:** Exploit properties of RSA to decrypt messages without knowing the private key
- **Mitigation:** Use proper padding schemes (PKCS#1, OAEP)

Padding Schemes

Padding: Adding random data to messages before encryption to prevent certain attacks and ensure messages are the correct length.

PKCS#1 v1.5 Padding

```cpp
std::vector<uint8_t> pkcs1_pad(const std::vector<uint8_t>& message,
                               size_t block_size) {
    std::vector<uint8_t> padded(block_size);
    padded[0] = 0x00;
    padded[1] = 0x02;

    // Fill with random non-zero bytes
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 255);

    for (size_t i = 2; i < block_size - message.size() - 1; i++) {
        padded[i] = dis(gen);
    }

    padded[block_size - message.size() - 1] = 0x00;
    std::copy(message.begin(), message.end(),
              padded.begin() + block_size - message.size());

    return padded;
}
```

Listing 9: PKCS1 v1.5 Padding

Performance Considerations

Computational Complexity

- **Key Generation:** $O(k^4)$ where k is the key size in bits
- **Encryption/Decryption:** $O(k^3)$ for each operation
- **CRT Decryption:** Approximately 4x faster than standard decryption

Optimization Techniques

1. **Use CRT for Decryption:** Can speed up decryption by 3-4x
2. **Choose Optimal Public Exponent:** $e = 65537$ is commonly used
3. **Use Efficient Big Integer Libraries:** GMP, OpenSSL, or Crypto++
4. **Implement Montgomery Multiplication:** For very large numbers

Real-World Applications

Digital Signatures

RSA can be used to create digital signatures:

```cpp
class RSASignature {
private:
    RSA rsa;

public:
    // Sign a message
    uint64_t sign(uint64_t message_hash) {
        return rsa.decrypt(message_hash); // Use private key
    }

    // Verify a signature
    bool verify(uint64_t message_hash, uint64_t signature) {
        uint64_t decrypted = rsa.encrypt(signature); // Use public key
        return decrypted == message_hash;
    }
};
```

Listing 10: RSA Digital Signature

Hybrid Encryption

RSA is often used in hybrid systems:

1. Generate a random symmetric key
2. Encrypt the message with the symmetric key (AES, ChaCha20, etc.)
3. Encrypt the symmetric key with RSA
4. Send both the encrypted message and encrypted key

Conclusion

RSA cryptography remains one of the most important and widely used asymmetric encryption algorithms. Understanding its mathematical foundations is crucial for implementing it correctly and securely.

Advantages of RSA:

- Well-understood and extensively analyzed
- Supports both encryption and digital signatures
- Widely supported in cryptographic libraries
- Relatively simple to implement (basic version)

Limitations of RSA:

- Slower than symmetric encryption
- Requires large key sizes for security
- Vulnerable to quantum attacks (Shor's algorithm)
- Complex to implement securely (padding, timing attacks)

Future Considerations

Quantum Threat: Shor's algorithm can factor large numbers efficiently on quantum computers, potentially breaking RSA. Post-quantum cryptography research is ongoing.

Further Reading

- "Handbook of Applied Cryptography" by Menezes, van Oorschot, and Vanstone
- "Cryptography: Theory and Practice" by Stinson
- RFC 8017: PKCS#1 v2.2
- NIST Special Publication 800-56B: Key Establishment Schemes

Mathematical Proofs

Proof of Euler's Theorem

Proof. Let $S = \{a_1, a_2, \ldots, a_{\phi(n)}\}$ be the set of integers coprime to n.
Consider the set $T = \{aa_1, aa_2, \ldots, aa_{\phi(n)}\}$ where $\gcd(a, n) = 1$.
We can show that:

1. All elements in T are coprime to n
2. All elements in T are distinct modulo n
3. Therefore, T is a permutation of S modulo n

This implies:

$$(aa_1)(aa_2)\cdots(aa_{\phi(n)}) \equiv a_1 a_2 \cdots a_{\phi(n)} \pmod{n}$$

Since $\gcd(a_i, n) = 1$ for all i, we can cancel the a_i terms:

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

□

Code Examples

Complete RSA Implementation

The complete implementation is available in the CryptoGL library at `src/RSA.cpp` and `src/RSA.hpp`.

Testing RSA Implementation

```cpp
#include <cassert>
#include <iostream>

void test_rsa() {
    // Test with small numbers
    RSA rsa_small(64);

    uint64_t test_message = 12345;
    uint64_t encrypted = rsa_small.encrypt(test_message);
    uint64_t decrypted = rsa_small.decrypt(encrypted);

    assert(test_message == decrypted);
    std::cout << "Small RSA test passed!\n";

    // Test with larger numbers
    RSA rsa_large(512);

    uint64_t large_message = 987654321;
    uint64_t large_encrypted = rsa_large.encrypt(large_message);
    uint64_t large_decrypted = rsa_large.decrypt(large_encrypted);

    assert(large_message == large_decrypted);
    std::cout << "Large RSA test passed!\n";
}

int main() {
    test_rsa();
    return 0;
}
```

Listing 11: RSA Test Suite