

RSA Algorithm Implementation and Mathematical Analysis

January 2025

Abstract

This document provides a comprehensive mathematical analysis of the RSA encryption algorithm. We examine the underlying number theory, implementation details, security considerations, and computational complexity. The analysis includes proofs of correctness, discussions of common attacks, and optimization techniques for practical implementations.

Contents

1	Introduction to RSA Cryptography	1
1.1	Mathematical Foundation	1
1.2	Key Generation	1
1.3	Encryption and Decryption	2
2	Mathematical Proofs	2
2.1	Correctness Proof	2
3	Implementation Details	2
3.1	Modular Exponentiation	2
3.2	Prime Generation	3
4	Security Analysis	3
4.1	Common Attacks	3
4.2	Security Recommendations	4
5	Performance Analysis	4
5.1	Computational Complexity	4
5.2	Optimization Techniques	4

6	Mathematical Extensions	4
6.1	RSA with Multiple Primes	4
6.2	Blinding Techniques	5
7	Practical Considerations	5
7.1	Key Management	5
7.2	Compliance Standards	5
8	Conclusion	5
8.1	Future Directions	5
A	Appendix A: Mathematical Background	6
A.1	Euler’s Theorem	6
A.2	Chinese Remainder Theorem	6
B	Appendix B: Sample Code	6

1 Introduction to RSA Cryptography

The RSA algorithm, named after Rivest, Shamir, and Adleman (1977), is one of the first practical public-key cryptosystems and is widely used for secure data transmission.

1.1 Mathematical Foundation

RSA is based on the computational difficulty of factoring large integers and the properties of modular arithmetic.

1.2 Key Generation

The RSA key generation process involves the following steps:

- 1. Choose two distinct prime numbers p and q
- 2. Compute $n = p \cdot q$
- 3. Compute Euler’s totient function: $\varphi(n) = (p - 1)(q - 1)$
- 4. Choose an integer e such that $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$
- 5. Compute d such that $d \cdot e \equiv 1 \pmod{\varphi(n)}$

1.3 Encryption and Decryption

For a message M represented as an integer with $0 \leq M < n$:

Encryption: $C \equiv M^e \pmod{n}$

Decryption: $M \equiv C^d \pmod{n}$

2 Mathematical Proofs

2.1 Correctness Proof

Using Euler's theorem and the Chinese Remainder Theorem, we can prove:

Theorem 2.1 (RSA Correctness). *For any message M with $0 \leq M < n$ and $\gcd(M, n) = 1$, the RSA encryption and decryption satisfy:*

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

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

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

Therefore:

$$C^d \equiv (M^e)^d \equiv M^{ed} \equiv M^{1+k\varphi(n)} \equiv M \cdot (M^{\varphi(n)})^k \equiv M \cdot 1^k \equiv M \pmod{n}$$

For the case where $\gcd(M, n) \neq 1$, the Chinese Remainder Theorem ensures the result still holds. \square

3 Implementation Details

3.1 Modular Exponentiation

Efficient modular exponentiation is crucial for RSA performance. The square-and-multiply algorithm provides $O(\log e)$ complexity:

Algorithm: Modular Exponentiation

Function ModExp(*base*, *exponent*, *modulus*):

1. *result* \leftarrow 1
2. *base* \leftarrow *base* mod *modulus*
3. **while** *exponent* > 0:
 - (a) **if** *exponent* mod 2 = 1:
 - i. *result* \leftarrow (*result* \times *base*) mod *modulus*
 - (b) *exponent* \leftarrow \lfloor *exponent*/2 \rfloor
 - (c) *base* \leftarrow (*base* \times *base*) mod *modulus*
4. **return** *result*

3.2 Prime Generation

Generating large primes is essential for RSA security:

- Use probabilistic primality tests (Miller-Rabin)
- Ensure p and q are of similar bit length
- Avoid primes with special forms that are easier to factor

4 Security Analysis

4.1 Common Attacks

Attack Type	Description
Factorization	Attempt to factor n into p and q
Small e attacks	When e is too small, certain messages can be recovered
Timing attacks	Measure computation time to deduce secret information
Side-channel attacks	Use power consumption, electromagnetic leaks, etc.

Table 1: Common attacks on RSA

4.2 Security Recommendations

1. Use key sizes of at least 2048 bits (3072 or 4096 for long-term security)
2. Use proper padding schemes (OAEP)
3. Implement constant-time algorithms to prevent timing attacks
4. Regularly update cryptographic libraries

5 Performance Analysis

5.1 Computational Complexity

5.2 Optimization Techniques

- Use Chinese Remainder Theorem for faster decryption

Operation	Complexity	Typical time (2048-bit)
Key generation	$O(k^3)$	1-10 seconds
Encryption	$O(k^2)$	< 1 ms
Decryption	$O(k^3)$	10-100 ms

Table 2: Computational complexity of RSA operations

- Implement windowing methods for exponentiation
- Use hardware acceleration (AES-NI, etc.)

6 Mathematical Extensions

6.1 RSA with Multiple Primes

Instead of two primes, use k primes: $n = p_1 p_2 \cdots p_k$

Advantages:

- Faster decryption using multi-prime CRT
- Same security with smaller total modulus

6.2 Blinding Techniques

To prevent timing attacks:

$$C' = C \cdot r^e \pmod{n}$$

$$M' = (C')^d \pmod{n}$$

$$M = M' \cdot r^{-1} \pmod{n}$$

7 Practical Considerations

7.1 Key Management

- Secure storage of private keys
- Regular key rotation
- Certificate management

7.2 Compliance Standards

- FIPS 140-2/3 for government use
- Common Criteria evaluation
- Industry-specific regulations

8 Conclusion

RSA remains a fundamental public-key cryptosystem despite the emergence of newer algorithms. Its security relies on the hardness of integer factorization, which remains computationally infeasible for properly chosen key sizes.

8.1 Future Directions

- Post-quantum cryptography alternatives
- Homomorphic encryption extensions
- Improved side-channel resistance

A Appendix A: Mathematical Background

A.1 Euler's Theorem

For any integers a and n with $\gcd(a, n) = 1$:

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

A.2 Chinese Remainder Theorem

If n_1, n_2, \dots, n_k are pairwise coprime, then the system of congruences:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

has a unique solution modulo $N = n_1 n_2 \cdots n_k$.

B Appendix B: Sample Code

```
# Simple RSA implementation in Python
import random
from math import gcd

def generate_keypair(bits=1024):
    p = generate_prime(bits//2)
    q = generate_prime(bits//2)
    n = p * q
    phi = (p-1) * (q-1)

    e = 65537
    d = modinv(e, phi)

    return ((e, n), (d, n))

def encrypt(pk, plaintext):
    key, n = pk
    cipher = pow(plaintext, key, n)
    return cipher

def decrypt(pk, ciphertext):
    key, n = pk
    plain = pow(ciphertext, key, n)
    return plain
```