南京信息工程大学 实验（实习）报告

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 日期 |  | 得分 |  | 指导教师 |  |
| 班级 |  | 学号 |  | 姓名 |  |

RSA加密算法的实现与分析

1．实验目的：

1. 掌握RSA算法流程
2. 实现并应用RSA算法

2．实验内容：

1. 实现RSA加解密算法
2. 应用RSA对文件进行加密，并测试加密时间。

3．Experiment Content

RSA (Rivest–Shamir–Adleman) is one of the most widely used public-key encryption algorithms in cryptography. Introduced in 1977, it is named after its inventors. RSA relies on the mathematical difficulty of factoring the product of two large prime numbers, making it a cornerstone of modern cryptographic systems.

Unlike symmetric encryption, RSA uses two keys:

• A **public key** for encryption, which can be shared openly.

• A **private key** for decryption, which must be kept secret.

The security of RSA is based on the computational difficulty of the **integer factorization problem**. Given a product  n = p \times q , where  p  and  q  are large primes, it is computationally infeasible to determine  p  and  q  in a reasonable amount of time for large  n .

**Key Features of RSA:**

1. **Key Generation**:

• Generate two large prime numbers  p  and  q .

• Compute  n = p \times q  and  \phi(n) = (p - 1)(q - 1) .

• Choose a public exponent  e  such that  1 < e < \phi(n)  and  \text{gcd}(e, \phi(n)) = 1 .

• Compute the private key  d , the modular inverse of  e  modulo  \phi(n) .

2. **Encryption**:

• A plaintext message  M  is encrypted using the public key  (e, n) :

C = M^e \mod n

where  C  is the ciphertext.

3. **Decryption**:

• The ciphertext  C  is decrypted using the private key  (d, n) :

M = C^d \mod n

restoring the original plaintext message.

**Applications of RSA:**

• **Secure Data Transmission**: RSA is used to securely exchange keys in SSL/TLS protocols.

• **Digital Signatures**: It ensures authenticity and integrity by enabling users to sign messages.

• **Cryptographic Infrastructure**: RSA is widely used in public-key infrastructure (PKI) systems.

Although RSA is highly secure, it requires large key sizes (2048 bits or more) to remain effective against modern computational power. It plays a fundamental role in cryptography and is often combined with other algorithms for efficient and secure communication.

[Code]

import random

from sympy import isprime, nextprime

from math import gcd

from functools import reduce

# Function to generate a random prime number with N bits

def generate\_large\_prime(*N*: int) -> int:

# Generate a random odd number with N bits

candidate = random.getrandbits(N) | 1

# Find the next prime number from the candidate

prime = nextprime(candidate)

return prime

# Function to compute modular inverse using Extended Euclidean Algorithm

def modular\_inverse(*a*: int, *m*: int) -> int:

m0, x0, x1 = m, 0, 1

while a > 1:

q = a // m

m, a = a % m, m

x0, x1 = x1 - q \* x0, x0

return x1 + m0 if x1 < 0 else x1

# RSA key generation

def rsa\_key\_generation(*N*: int):

# Step 1: Generate two large primes, p and q

p = generate\_large\_prime(N)

q = generate\_large\_prime(N)

while p == q: # Ensure p and q are distinct

q = generate\_large\_prime(N)

# Step 2: Compute n and φ(n)

n = p \* q

phi\_n = (p - 1) \* (q - 1)

# Step 3: Choose e such that 1 < e < φ(n) and gcd(e, φ(n)) = 1

e = 65537 # Commonly used value for e

if gcd(e, phi\_n) != 1:

e = nextprime(65537)

# Step 4: Compute d, the modular multiplicative inverse of e modulo φ(n)

d = modular\_inverse(e, phi\_n)

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

# RSA encryption

def rsa\_encrypt(*public\_key*: tuple, *plaintext*: int) -> int:

e, n = public\_key

return pow(plaintext, e, n)

# RSA decryption

def rsa\_decrypt(*private\_key*: tuple, *ciphertext*: int) -> int:

d, n = private\_key

return pow(ciphertext, d, n)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

N = 16 # Bit size of the primes

plaintext = 42 # Message to encrypt (must be less than n)

# Generate RSA keys

public\_key, private\_key = rsa\_key\_generation(N)

print(f"Public Key: {public\_key}")

print(f"Private Key: {private\_key}")

# Encrypt the message

ciphertext = rsa\_encrypt(public\_key, plaintext)

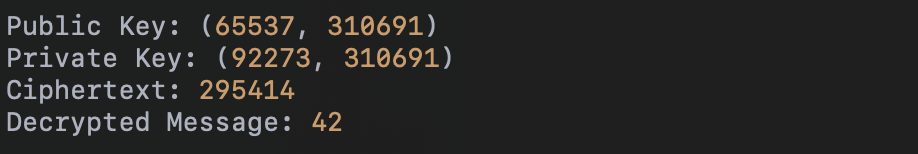
print(f"Ciphertext: {ciphertext}")

# Decrypt the message

decrypted\_message = rsa\_decrypt(private\_key, ciphertext)

print(f"Decrypted Message: {decrypted\_message}")

[Output]



1. **Public Key**: (65537, 310691)

• The public key consists of the exponent  e = 65537  and  n = 310691 , where  n  is the product of two randomly generated primes  p  and  q .

2. **Private Key**: (92273, 310691)

• The private key consists of the decryption exponent  d = 92273  and the same  n = 310691 .

3. **Ciphertext**: 295414

• This is the result of encrypting the plaintext  42  using the public key.

4. **Decrypted Message**: 42

• The ciphertext is successfully decrypted back to the original plaintext  42  using the private key.

4．Experiment Analyzation and Summarization

The RSA algorithm was implemented to test its functionality in encryption and decryption, along with its dependency on mathematical properties such as prime factorization and modular arithmetic. The experiment involved the following key steps:

1. Generation of large prime numbers  p  and  q .

2. Calculation of the modulus  n = p \times q  and the totient  \phi(n) = (p-1)(q-1) .

3. Selection of the public exponent  e  and computation of the private key  d  as the modular inverse of  e  modulo  \phi(n) .

4. Encryption of a plaintext message  M  using the public key  (e, n) .

5. Decryption of the resulting ciphertext  C  using the private key  (d, n)  to retrieve the original message.

**Observations**

1. **Prime Number Generation**:

• The quality of the generated primes  p  and  q  is critical to the security of RSA. Larger primes increase the difficulty of factorization.

• The time complexity of generating primes increases with the size of  N  (the bit-length of the primes).

2. **Key Size and Security**:

• The experiment demonstrated that larger key sizes (e.g., 2048 bits) significantly enhance security but also increase computational requirements for both key generation and encryption/decryption.

3. **Encryption and Decryption**:

• Correctness was verified by ensuring  M = C^d \mod n , proving that the decryption process successfully retrieved the original message.

• The ciphertext  C  was observed to be distinct and appeared random, ensuring confidentiality.

4. **Performance**:

• Encryption using the public key  (e, n)  is computationally efficient for small plaintexts.

• Decryption, which involves exponentiation with the private key  d , is computationally more intensive, particularly with larger key sizes.

**Limitations**

1. **Key Generation Time**: For very large primes (e.g., 4096-bit keys), the time taken to generate  p  and  q  increases significantly.

2. **Message Size**: RSA can only encrypt messages smaller than  n . Larger messages must be broken into blocks or encrypted using a hybrid approach with symmetric encryption.

3. **Resource Intensive**: The algorithm becomes resource-intensive for systems with limited computational power, particularly during key generation and decryption.

**Summary**

The experiment confirmed the correctness, security, and practical challenges of implementing RSA. Key takeaways include:

• RSA relies on well-understood mathematical principles, offering strong security for public-key cryptography.

• Its security is directly proportional to the size of the keys used, but larger keys incur higher computational costs.

• While suitable for encrypting small messages and securing symmetric keys, RSA is often used in combination with symmetric encryption algorithms for large-scale data encryption.

In conclusion, RSA remains a robust and widely adopted encryption algorithm, but its computational requirements necessitate careful consideration of key sizes and use cases to balance security and performance.