山东大学＿＿计算机科学与技术＿＿学院

＿＿云计算技术＿＿课程实验报告

学号：202200130048	姓名： 陈静雯	班级： 6
实验题目：实现 RSA 加解密及破解		
实验学时：2	实验日期： 2025.4.23	

实验目的：实现 RSA 加解密实验，认识 RSA 加解密过程。
具体内容：
1）编程实现 RSA 加密算法，包括密钥生成、加密和解密过程；
2）基于因式分解进行破解，注意为了保证破解的可行性，需要选取合适的秘钥长度，请选择 5 种不同的秘钥长度，并给出对应的破解时间。

硬件环境：
计算机一台

软件环境：
Linux 或 Windows

实验步骤：
1）编程实现 RSA 加密算法，包括密钥生成、加密和解密过程；
2）基于因式分解进行破解，选取合适的秘钥长度，请选择 5 种不同的秘钥长度，并给出对应的破解时间。

RSA 算法原理：

一、核心思想

1. 非对称性
   使用一对密钥：公钥（公开）用于加密，私钥（保密）用于解密，二者数学关联但无法互相推导。
2. 数学基础
   - 大素数分解难题：将一个大合数分解为两个大素数的乘积在计算上不可行。
   - 欧拉定理：若 mm 与 nn 互质，则 mϕ(n)≡1mod n$m\phi(n)\equiv 1\bmod n$，其中 ϕ(n)$\phi(n)$ 为欧拉函数。

二、密钥生成过程

密钥生成是 RSA 的核心，分为以下步骤：

1. 选择两个大素数
- 随机生成两个大素数 p 和 q（例如使用米勒-拉宾素性测试）。
- 代码对应：generate_prime(bit_length) 函数生成素数。

2. 计算模数 n
   n=p×q，n 公开，但其因子 p 和 q 必须保密。

3. 计算欧拉函数 ϕ(n)
   ϕ(n)=(p−1)(q−1)，ϕ(n)$\phi(n)$ 的值必须保密，它决定了后续密钥的生成。

4. 选择公钥指数 e
- 选择一个整数 e，满足：

1. $1<e<\phi(n)$
2. e 与 $\phi(n)$ 互质（即 $\gcd(e,\phi(n))=1$）。

- 常见选择：e=65537（费马素数，二进制仅含两个 1，计算高效）。
- 代码对应：固定使用 e = 65537，若与 $\phi(n)$ 不互质则重新生成密钥。

5. 计算私钥指数 d

$$d\equiv e-1 \bmod \phi(n)$$

- d 是 e 关于模 $\phi(n)$的模逆元，需严格保密。
- 代码对应：通过 modular_inverse(e, phi) 计算 d。

6. 生成密钥对

- 公钥：(e,n)
- 私钥：(d,n)

三、加密与解密过程

1. 加密（使用公钥）

- 明文 m 需满足 m<n。
- 计算密文 c：

$$c\equiv m^e \bmod n$$

- 代码实现：pow(m, e, n)

2. 解密（使用私钥）

- 计算明文 m：

$$m\equiv c^d \bmod n$$

- 代码实现：pow(c, d, n)

四、安全性分析

RSA 的安全性基于以下两点：

1. 大整数分解难题
   攻击者需分解 $n=p\times q$ 才能计算 $\phi(n)$，进而破解私钥 d。当 p 和 q 足够大时（如 2048 位），分解在计算上不可行。
2. 直接计算 $\phi(n)$的困难性
   已知 n 但不知 p 和 q，计算 $\phi(n)=(p-1)(q-1)$的复杂度等同于分解 n。

实验内容：

1.编程实现

1) 素数生成：
   - 使用米勒-拉宾素性测试生成指定位数的大素数
   - 通过 generate_prime 函数确保生成符合要求的素数
2) 密钥生成：
   - 通过 generate_rsa_key 生成 RSA 公私钥
   - 自动处理 e 与 $\varphi(n)$不互质的情况
3) 加解密实现：
   - 直接使用 Python 内置的 pow 函数实现模幂运算
   - 支持任意整数消息的加解密（需满足 m < n）
4) 因式分解破解：
   - 使用 Pollard's Rho 算法进行高效因数分解
   - 递归分解直至获得所有质因数
   - 记录不同密钥长度下的破解时间

```python
import random
import math
import time

def is_prime(n, k=5):
    if n <= 1:
        return False
    elif n <= 3:
        return True
    d = n - 1
    s = 0
    while d % 2 == 0:
        d //= 2
        s += 1
    for _ in range(k):
        a = random.randint(2, min(n-2, 1 << 20))
        x = pow(a, d, n)
        if x == 1 or x == n-1:
            continue
        for __ in range(s-1):
            x = pow(x, 2, n)
            if x == n-1:
                break
        else:
            return False
    return True
```

```python
def generate_prime(bit_length):
    while True:
        p = random.getrandbits(bit_length)
        p |= (1 << (bit_length - 1))
        if p % 2 == 0:
            p += 1
        if is_prime(p):
            return p

def extended_gcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = extended_gcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modular_inverse(a, m):
    g, x, y = extended_gcd(a, m)
    if g != 1:
        raise ValueError('Modular inverse does not exist')
    else:
        return x % m
```

```python
def generate_rsa_key(key_length):
    p_bit = key_length // 2
    q_bit = key_length - p_bit
    p = generate_prime(p_bit)
    q = generate_prime(q_bit)
    while p == q:  # 避免p和q相同
        q = generate_prime(q_bit)
    n = p * q
    phi = (p-1) * (q-1)
    e = 65537
    try:
        d = modular_inverse(e, phi)
    except ValueError:
        return generate_rsa_key(key_length)  # 自动重试
    return (e, n), (d, n)

def pollards_rho(n):
    if n % 2 == 0:
        return 2
    if n % 3 == 0:
        return 3
    if n % 5 == 0:
        return 5
    while True:
        c = random.randint(1, n-1)
        f = lambda x: (pow(x, 2, n) + c) % n
        x, y, d = 2, 2, 1
```

```python
        while d == 1:
            x = f(x)
            y = f(f(y))
            d = math.gcd(abs(x-y), n)
        if d != n:
            return d

def factor(n):
    factors = []
    def _factor(n):
        if n == 1:
            return
        if is_prime(n):
            factors.append(n)
            return
        d = pollards_rho(n)
        _factor(d)
        _factor(n//d)
    _factor(n)
    return sorted(factors)
```

```python
def main():
    key_lengths = [20, 40, 60, 80, 100]  # 修改密钥长度
    print("RSA加解密及破解实验（自定义密钥长度）")
    print("=" * 45)

    # 加解密演示（使用40位密钥）
    print("\n加解密演示:")
    public_key, private_key = generate_rsa_key(40)
    e, n = public_key
    d, _ = private_key
    print(f"[密钥生成]")
    print(f"公钥 (e, n): ({e}, {n})")
    print(f"私钥 (d, n): ({d}, {n})")

    # 生成小明文（确保m < n）
    m = random.randint(1, n//1000)
    c = pow(m, e, n)
    m_dec = pow(c, d, n)
    print(f"\n[加解密过程]")
    print(f"原始消息: {m}")
    print(f"加密后密文: {c}")
    print(f"解密后消息: {m_dec}")
```

```python
    # 破解实验
    print("\n因式分解破解时间测试:")
    for kl in key_lengths:
        print(f"\n密钥长度: {kl} bits")
        public_key, _ = generate_rsa_key(kl)
        _, n = public_key
        print(f"攻击目标公钥 n = {n}")
        start = time.time()
        factors = factor(n)
        elapsed = time.time() - start
        print(f"因数分解结果: {factors}")
        print(f"破解时间: {elapsed:.4f}秒")

if __name__ == "__main__":
    main()
```

2.选择 5 种不同的秘钥长度，并给出对应的破解时间。

```
加解密演示:
[密钥生成]
公钥 (e, n): (65537, 521946083273)
私钥 (d, n): (194722761473, 521946083273)

[加解密过程]
原始消息: 431792029
加密后密文: 237242569330
解密后消息: 431792029

因式分解破解时间测试:

密钥长度: 20 bits
攻击目标公钥 n = 748747
因数分解结果: [751, 997]
破解时间: 0.0000秒

密钥长度: 40 bits
攻击目标公钥 n = 890940175297
因数分解结果: [921887, 966431]
破解时间: 0.0004秒

密钥长度: 60 bits
攻击目标公钥 n = 516046698103693433
因数分解结果: [625344443, 825219931]
破解时间: 0.0115秒

密钥长度: 80 bits
攻击目标公钥 n = 606745680398350573294901
因数分解结果: [643957421629, 942213972569]
破解时间: 0.2042秒

密钥长度: 100 bits
攻击目标公钥 n = 941057125306430211236352327397
因数分解结果: [910941002024219, 1033060454206463]
破解时间: 24.3856秒
```

结论分析与体会：

1.密钥长度选择

（1）推荐使用 2048 位以上 的密钥（演示代码中使用小密钥仅为教学目的）。

2.明文填充方案

（1）直接加密原始消息存在安全风险（如明文猜测攻击）。

（2）实际使用需结合填充方案（如 OAEP）增强安全性。

3.性能优化

（1）加密时选择小公钥指数（如 e=65537*e*=65537）提升效率。

（2）解密时利用中国剩余定理（CRT）加速计算。

4.密钥长度对性能的影响

密钥长度	密钥生成速度	加密/解密速度	破解速度（因式分解）	安全性

16-32 位	极快（<0.1 秒）	极快（微秒级）	瞬时（<0.1 秒）	极低
64 位	快（<1 秒）	极快	秒级	低
128 位	中等（秒级）	快	分钟级	弱
256 位	慢（分钟级）	中等	小时级	中
2048 位	极慢（小时级）	慢	实际不可行	高

指数级增长