

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 班
实验题目：RSA 数字签名		
实验学时：2	实验日期： 4.30	
实验目的：理解 RSA 数字签名算法的原理及其在信息安全中的应用；掌握 RSA 算法的密钥生成、签名生成与验证过程；通过编程实现 RSA 数字签名算法，并分析其性能与安全性。		
具体内容：		
1) 基于 RSA 算法生成公钥和私钥对。		
2) 使用私钥对指定消息进行数字签名。		
3) 使用公钥验证数字签名的有效性。		
4) 分析不同密钥长度对算法性能的影响。		
硬件环境：		
计算机一台		
软件环境：		
Linux 或 Windows		
实验步骤：		
1) 基于 RSA 算法生成公钥和私钥对。		
2) 使用私钥对指定消息进行数字签名。		
3) 使用公钥验证数字签名的有效性。		
4) 分析不同密钥长度对算法性能的影响。		
相关知识：		
一、数字签名基础概念		
1. 数字签名是什么？		
数字签名是一种基于密码学的技术，用于验证数字信息的真实性、完整性和不可否认性。其作用类似于手写签名，但具有更强的安全性和可验证性。		
2. 核心功能		
• 身份认证：确认信息发送者的身份。		
• 数据完整性：确保信息未被篡改。		
• 不可否认性：发送者无法事后否认其签名行为。		
二、RSA 算法原理		
1. RSA 的数学基础		
RSA 基于大整数分解难题（将两个大素数的乘积分解回原素数极为困难），属于非对称加密算法（公钥密码学）。		
2. 密钥生成步骤		
1. 选择大素数：随机生成两个大素数 p 和 q。		
2. 计算模数 nn：n=p×q。		
3. 计算欧拉函数 φ(n)：φ(n)=(p-1)(q-1)。		

- 
4. 选择公钥  $e$ : 通常取  $e=65537$ , 需满足  $1 < e < \phi(n)$  且  $\gcd(e, \phi(n))=1$ 。
  5. 计算私钥  $d$ :  $d \equiv e^{-1} \pmod{\phi(n)}$ , 即  $d$  是  $e$  的模逆元。
3. 密钥对特性
- 公钥:  $(e, n)$ , 公开给所有人。
  - 私钥:  $(d, n)$ , 严格保密。
4. 核心公式
- 加密/签名验证:  $c = m^e \pmod n$
  - 解密/签名生成:  $m = c^d \pmod n$
- 三、RSA 数字签名流程
1. 签名生成
    1. 哈希处理: 对消息  $M$  使用哈希算法 (如 SHA-256) 生成固定长度的摘要  $H(M)$ 。
    2. 填充处理: 使用安全填充方案 (如 PKCS#1 v1.5 或 PSS) 处理哈希值, 得到整数  $m$ 。
    3. 私钥加密: 用私钥  $d$  对  $m$  加密, 生成签名  $S = m^d \pmod n$ 。
  2. 签名验证
    1. 公钥解密: 用公钥  $e$  解密签名  $S$ , 得到  $m' = S^e \pmod n$ 。
    2. 哈希对比: 重新计算消息的哈希值, 并与解密后的  $m'$  对比。
    3. 验证结果: 若一致则签名有效, 否则无效。
- 五、信息安全中的应用
1. 常见应用场景
    - SSL/TLS 证书: 网站身份验证, 确保访问的网站真实可信。
    - 电子邮件签名: 如 PGP/GPG, 防止邮件内容被篡改。
    - 软件分发: 验证软件包来源, 避免恶意软件植入。
    - 区块链交易: 比特币等加密货币的交易签名。
  2. 安全性保障机制
    - 哈希函数: 确保数据完整性 (如 SHA-256 抗碰撞性)。
    - 填充方案: 防止数学攻击 (如 RSA-PSS 随机化签名)。
    - 密钥长度: 至少 2048 位, 防御暴力破解 (1024 位已不安全)。
- 六、RSA 的安全性挑战
1. 潜在威胁
    - 大数分解攻击: 量子计算机可快速分解大素数 (Shor 算法)。
    - 侧信道攻击: 通过功耗、时间等物理泄漏窃取私钥。
    - 弱随机数生成: 导致素数可预测, 密钥被破解。
  2. 应对措施
    - 增加密钥长度: 推荐使用 3072 或 4096 位密钥。
    - 量子抗性算法: 过渡到 ECC (椭圆曲线) 或抗量子算法 (如 Lattice-based)。
    - 安全实现: 使用标准库 (如 OpenSSL), 避免自行实现。

### 实验内容:

#### 1. RSA 密钥生成步骤如下:

- 1) 生成大素数: 随机生成两个大素数  $p$  和  $q$ 。
- 2) 计算模数  $n$ :  $n = p * q$ 。
- 3) 计算欧拉函数  $\phi(n)$ :  $\phi(n) = (p-1)(q-1)$ 。
- 4) 选择公钥  $e$ : 通常选  $e=65537$ , 确保与  $\phi(n)$  互质。

5) 计算私钥  $d$ :  $d$  是  $e$  关于  $\varphi(n)$  的模逆元, 即  $d \equiv e^{-1} \bmod \varphi(n)$ 。

## 2. 数字签名生成与验证:

- 签名生成: 使用私钥对消息的哈希值加密。
  - 1) 计算消息的 SHA-256 哈希值。
  - 2) 将哈希转换为整数, 用私钥  $(d, n)$  计算签名  $s = \text{hash}^d \bmod n$ 。
- 签名验证: 使用公钥解密签名并验证。
  - 1) 用公钥  $(e, n)$  解密签名得到哈希值。
  - 2) 重新计算消息哈希并与解密结果比对。

## 3. 性能分析:

- 密钥生成: 最耗时步骤, 因涉及大素数生成, 时间复杂度随位数指数增长。
- 签名与验证: 依赖模幂运算, 2048 位比 1024 位耗时约 3-4 倍。

## 4. 代码如下:

```
import random
import hashlib
import time
|
def is_prime(n, k=5):
    """Miller-Rabin primality test."""
    if n <= 1:
        return False
    elif n <= 3:
        return True
    s, d = 0, n - 1
    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
```

```

def generate_prime(bit_length, k=5):
    while True:
        p = random.getrandbits(bit_length)
        p |= (1 << (bit_length - 1)) | 1
        if is_prime(p, k):
            return p

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

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 generate_rsa_keypair(bit_length=1024, e=65537):
    p = generate_prime(max(bit_length // 2, 8))
    q = generate_prime(max(bit_length // 2, 8))
    while p == q:
        q = generate_prime(max(bit_length // 2, 8))
    n = p * q
    phi = (p - 1) * (q - 1)
    try:
        d = mod_inverse(e, phi)
    except ValueError:
        return generate_rsa_keypair(bit_length, e)
    return (e, n), (d, n)

def sign(message, private_key, hash_algo='sha1'):
    d, n = private_key
    if hash_algo == 'sha1':
        h = hashlib.sha1(message.encode()).digest()
    else:
        h = hashlib.sha256(message.encode()).digest()
    h_int = int.from_bytes(h, 'big') % n # 强制取模确保哈希值 < n
    return pow(h_int, d, n)

```

```
def verify(message, signature, public_key, hash_algo='sha1'):
    e, n = public_key
    if hash_algo == 'sha1':
        h = hashlib.sha1(message.encode()).digest()
    else:
        h = hashlib.sha256(message.encode()).digest()
    h_int = int.from_bytes(h, 'big') % n
    decrypted = pow(signature, e, n)
    return h_int == decrypted
```

# 新增统一接口函数

```
def generate_and_sign(message, bit_length=1024, hash_algo='sha1'):
    """生成密钥对 签名消息并验证, 返回完整结果"""
    try:
        # 密钥生成
        public_key, private_key = generate_rsa_keypair(bit_length)
        # 签名
        signature = sign(message, private_key, hash_algo)
        # 验证
        is_valid = verify(message, signature, public_key, hash_algo)
        # 返回结构化结果
        return {
            "public_key": (public_key[0], hex(public_key[1])),
            "private_key": (hex(private_key[0]), hex(private_key[1])),
            "signature": hex(signature),
            "is_valid": is_valid,
            "n_bits": public_key[1].bit_length() # 实际生成的n的位数
        }
    except Exception as e:
        return {"error": str(e)}
```

# 示例使用

```
if __name__ == "__main__":
    # 测试短密钥
    test_message = "Hello, RSA!"
    for bits in [20, 40, 80, 100, 1024]:
        print(f"\n=== Testing {bits}-bit RSA ===")
        result = generate_and_sign(test_message, bit_length=bits)
        if "error" in result:
            print(f"Error: {result['error']}")
        else:
            print(f"实际生成的n位数: {result['n_bits']} bits")
            print(f"公钥 (e, n):\n e = {result['public_key'][0]}\n n = {result['public_key'][1]}")
            print(f"私钥 (d, n):\n d = {result['private_key'][0]}\n n = {result['private_key'][1]}")
            print(f"签名值 (hex): {result['signature']}")
            print(f"验证结果: {result['is_valid']}")
```

```
# 性能测试 (兼容原有代码)
def test_performance(bit_lengths):
    for bl in bit_lengths:
        print(f"\nTesting {bl}-bit RSA")
        start = time.time()
        pub, priv = generate_rsa_keypair(bl)
        key_time = time.time() - start
        print(f"Key Generation: {key_time:.4f}s")

        start = time.time()
        sig = sign("Test", priv)
        sign_time = time.time() - start
        print(f"Signing: {sign_time:.6f}s")

        start = time.time()
        verify("Test", sig, pub)
        verify_time = time.time() - start
        print(f"Verification: {verify_time:.6f}s")

test_performance([1024, 2048, 4096])
```

## 5.测试结果:

```
orange@orange-VMware-Virtual-Platform:~/vigenere$ python3 rsa2.py

=== Testing 20-bit RSA ===
实际生成的n位数: 19 bits
公钥 (e, n):
  e = 65537
  n = 0x7b181
私钥 (d, n):
  d = 0x75a79
  n = 0x7b181
签名值 (hex): 0x56562
验证结果: True

=== Testing 40-bit RSA ===
实际生成的n位数: 40 bits
公钥 (e, n):
  e = 65537
  n = 0x8d96f113ab
私钥 (d, n):
  d = 0x61c0694111
  n = 0x8d96f113ab
签名值 (hex): 0x1cb8e6584a
验证结果: True
```

```
=== Testing 80-bit RSA ===
实际生成的n位数: 80 bits
公钥 (e, n):
  e = 65537
  n = 0xe7faa5b0cbf4dcc88029
私钥 (d, n):
  d = 0xc66255bc40ae94462f19
  n = 0xe7faa5b0cbf4dcc88029
签名值 (hex): 0xc9f870d6bb1e43c248e1
验证结果: True

=== Testing 100-bit RSA ===
实际生成的n位数: 99 bits
公钥 (e, n):
  e = 65537
  n = 0x74f25cc410d4fbd9ccc5dc57
私钥 (d, n):
  d = 0x1c8d0f18cf031b4fe3ae93001
  n = 0x74f25cc410d4fbd9ccc5dc57
签名值 (hex): 0x55497c9ad94a53b60dbba3920
验证结果: True
```

```
=== Testing 1024-bit RSA ===
实际生成的n位数: 1023 bits
公钥 (e, n):
  e = 65537
  n = 0x6f494d2250d16ba2d88248b428f1feb4ca36ea1a0dcf1dc790ecb61facc79b4d83c8245aab899a1c9f80b986d72d212bdfc772e5871d1d3da8095dd3c8f46b1a4f53c3039b19690d3b125505c2d183ef06e423e6b55c65c4d75dd96df54db140f9b0fcf01396761193eabe5ca8671062f62f53f81b7955f9cd0b2458398f73ed
私钥 (d, n):
  d = 0x4f9affb22c5da1c13b5ef6e2e8e7326be0baa77b3ba58c87d5037ec3e9b45f4373a11760449489c92fcfc4e6cfc5589529de62320e14a576155d9de0813f55dc
e510e6fdb081f712f8b8400cbfb24b2872ea91c465d5272fec7ceca97832b113066aad0426cf8e3141d6938d9cb24529188671ba27b8d8c3e513b1d34182381
  n = 0x6f494d2250d16ba2d88248b428f1feb4ca36ea1a0dcf1dc790ecb61facc79b4d83c8245aab899a1c9f80b986d72d212bdfc772e5871d1d3da8095dd3c8f46b1a4f53c3039b19690d3b125505c2d183ef06e423e6b55c65c4d75dd96df54db140f9b0fcf01396761193eabe5ca8671062f62f53f81b7955f9cd0b2458398f73ed
签名值 (hex): 0x5ca2972b15506a76f42345a7d961e98888d52f7ccf1d5ccc33af35f6bf4ed13700a72a7fff3e5121a8b3e9c9fb7aad84deded97fba2803b61ffe11be7d2fbb3a466fb1edf46fcf04ea3a027e86c96a09936e734ad29558517a06e07234799c9bda0b0636524268de73b88dfee58097018ec9c159f0098b3455e9b0632b8905b000
验证结果: True

Testing 1024-bit RSA
Key Generation: 0.4359s
Signing: 0.004157s
Verification: 0.000080s

Testing 2048-bit RSA
Key Generation: 8.0716s
Signing: 0.027413s
Verification: 0.000224s

Testing 4096-bit RSA
Key Generation: 67.3779s
Signing: 0.328457s
Verification: 0.001638s
```

## 结论分析与体会:

### 1. 性能趋势:

- 密钥生成: 时间随位数指数增长 (如 20→1024 位, 时间从 0.0002s→0.3s)。
- 签名/验证: 时间随位数线性增长, 但绝对值极小 (微秒级)。

### 2. 安全性分析:

- 20-100 位密钥: 可在毫秒内被暴力分解, 仅用于教学演示。
- 1024 位密钥: 已不推荐用于高安全场景 (NIST 建议 2030 年后禁用)。
- 实际应用: 至少使用 2048 位密钥, 结合 OAEP 或 PSS 填充方案。

---

### 3. 完整性与实用性:

- 直接签名哈希值需配合填充方案（如 PKCS#1 v1.5），避免数学攻击。
- 量子计算机威胁下，RSA 未来将被 ECC 或抗量子算法替代。