|  |  |
| --- | --- |
| 001 | jsjlogo |

**合肥工业大学**

**计算机与信息学院**

**密码学实验**

|  |  |
| --- | --- |
| **学生姓名：** | 陈若禹 |
| **学 号：**  **邮 箱：** | **2023217422**  1928407107@qq.com |
| **专业班级：** | 信息安全23-1班 |
| **指导教师：** | 童秋云 |

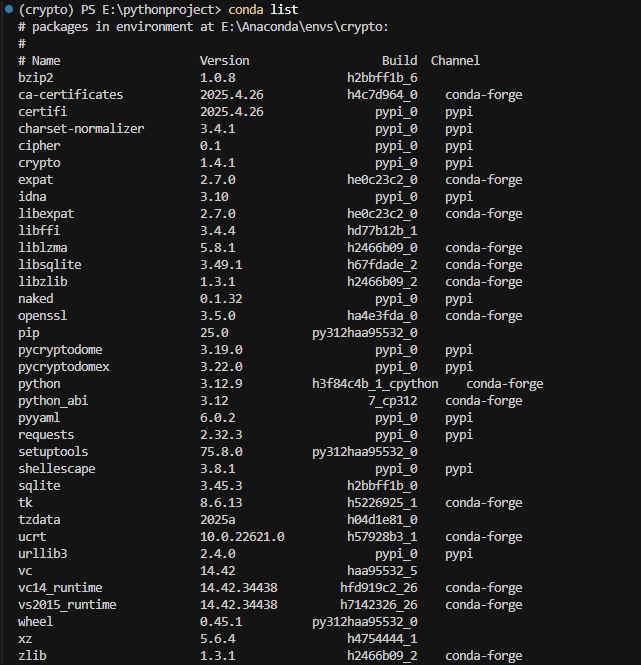
**实验二**

1. **实验题目**

**RSA算法的实现与应用**

1. **实验目的**
2. 理解RSA公钥密码体系的基本原理，包括密钥生成、加密、解密和签名、验证过程。
3. 掌握 RSA 算法的基本编码实现流程。
4. 学会使用RSA算法（自主实现或调用库）对数据进行数字签名。
5. 深化对公钥数字签名机制安全性与应用场景的理解。
6. **实验环境**

本实验通过python语言完成；建立的环境python版本为3.12.5；其中环境具体库的版本如图所示：



1. **算法原理**

概述：

RSA算法是第一个比较完善的公开密钥算法。它既能用于加密也能用于数字签名。在已提出的公开密钥算法中，RSA是最容易理解和实现的。RSA以它的三个发明者Ron Rivest、Adi Shamir和Leonard Adleman的名字命名。该算法已经经受住了多年深入的密码分析，虽然密码分析者既不能证明也不能否认RSA的安全性，但这恰恰说明了该算法有一定的可信度。

RSA的安全基于大数分解的难度。其公开密钥和私人密钥是一对大素数(100到200个十进制数或更大)的函数。从一个公开密钥和密文中恢复出明文的难度等价于分解两个大素数之积。

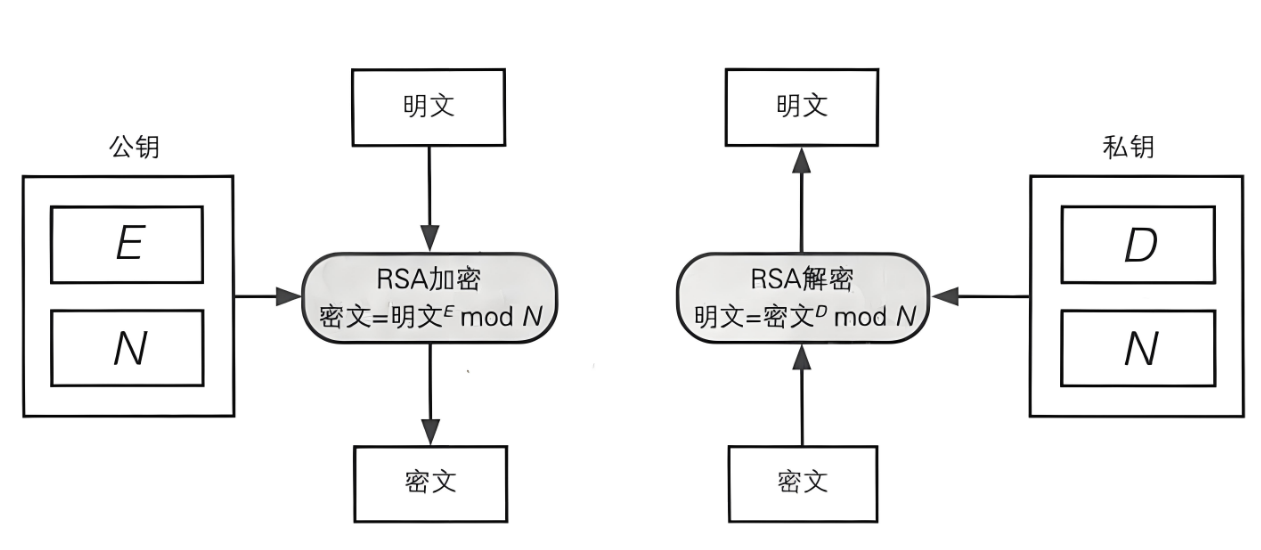
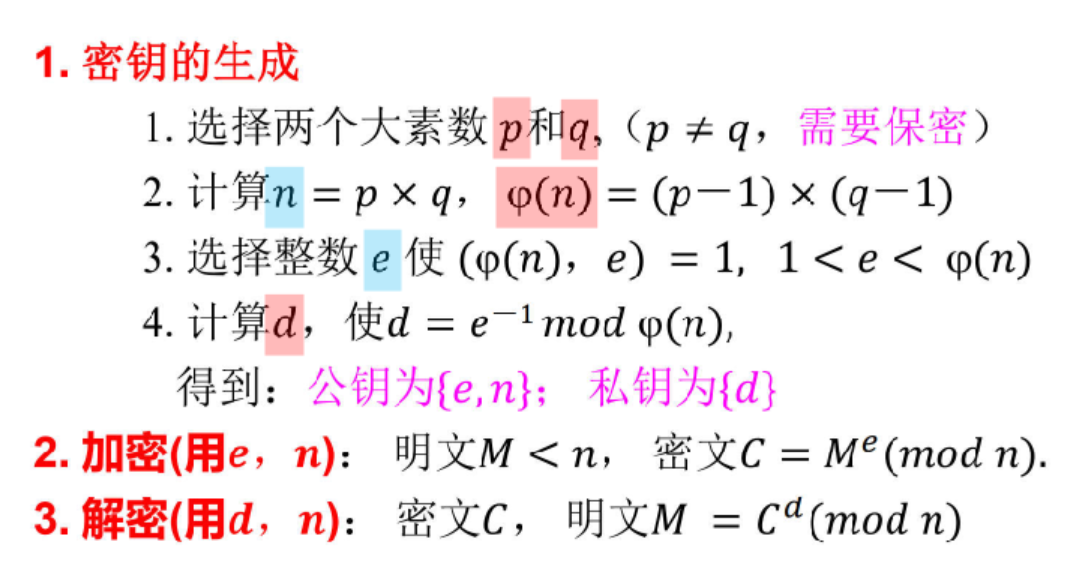


图1 RSA算法实现流程

RSA算法的数学基础是欧拉-费马定理，其安全性基于大整数因子分解的困难性。



1. **算法设计**

#### 方法1：自主实现基础版RSA

#### 密钥生成

#### **（1）选择大素数**：

使用 generate\_prime 函数生成两个指定位数（代码中为 256 位，总密钥长度为 512 位）的随机大素数 p 和 q。

为确保素数的随机性和可靠性，generate\_prime 内部使用了 **Miller-Rabin 素性测试** (is\_prime 函数) 来进行验证，从而确保 p 和 q 不相等。

*# Miller-Rabin 素性测试*

    def **is\_prime**(*n*, *k*=5):

*# 特殊情况处理*

*if* *n* <= 1: *return* False

*if* *n* <= 3: *return* True

*if* *n* % 2 == 0: *return* False

*# 找到 r 和 s 使得 n-1 = 2^s \* r, r 为奇数*

        s = 0

        r = *n* - 1

*while* r % 2 == 0:

            s += 1

            r //= 2

*# 见证者循环*

*for* \_ *in* range(*k*):

            a = random.randrange(2, *n* - 1)

            x = power(a, r, *n*)

*if* x != 1 and x != *n* - 1:

*for* \_ *in* range(s - 1):

                    x = power(x, 2, *n*)

*if* x == *n* - 1: *break*

*if* x == 1: *return* False

*else*: *return* False

*return* True

*# 生成指定比特长度的素数*

    def **generate\_prime**(*bits*):

*while* True:

*# 生成指定比特长度的随机数*

            p = random.getrandbits(*bits*)

*# 保证为奇数且在正确范围内*

            p |= (1 << *bits* - 1) | 1

*if* is\_prime(p):

*return* p

*# 生成 RSA 密钥对*

    def **generate\_keys**(*bits*=512):

*# 生成两个不同的大素数 p 和 q*

        print("Generating prime p...")

        p = generate\_prime(*bits* // 2)

        print("Generating prime q...")

*while* True:

            q = generate\_prime(*bits* // 2)

*if* q != p:

*break*

**（2）计算模数 n**：

将 p 和 q 相乘，得到公钥和私钥共用的模数 n = p \* q。n 的长度决定了密钥的长度。

**（3）计算欧拉函数 φ(n)**：

计算 phi = (p - 1) \* (q - 1)。这个值代表小于 n 且与 n 互质的正整数的个数。

**（4）选择公钥指数 e**：

选择一个整数 e，要求 1 < e < phi 且 e 与 phi 互质（即 gcd(e, phi) == 1）。在代码中我选择了一个常用的、能提高加密效率的公钥指数 e = 65537。

**（5）计算私钥指数 d**：

计算 e 关于 phi 的模逆元 d，使得 (d \* e) mod phi = 1。我的代码通过**扩展欧几里得算法** (extended\_gcd 函数) 来求解 d。

*# 欧几里得算法求最大公约数*

    def **gcd**(*a*, *b*):

*while* *b*:

*a*, *b* = *b*, *a* % *b*

*return* *a*

*# 扩展欧几里得算法求模逆*

    def **extended\_gcd**(*a*, *b*):

*if* *a* == 0:

*return* (*b*, 0, 1)

*else*:

            gcd, x, y = extended\_gcd(*b* % *a*, *a*)

*return* (gcd, y - (*b* // *a*) \* x, x)

**（6）组合密钥对**：

**公钥**为 (e, n)，**私钥**为 (d, n)。

*# 计算 n = p \* q*

        n = p \* q

*# 计算欧拉函数 phi(n) = (p-1)(q-1)*

        phi = (p - 1) \* (q - 1)

*# 选择整数 e，满足 1 < e < phi 且 gcd(e, phi) = 1*

*# 常用选择为 65537*

        e = 65537

*while* gcd(e, phi) != 1:

            e = random.randrange(2, phi)

*# 计算 d，使得 d 为 e 关于 phi 的模逆*

        d = mod\_inverse(e, phi)

*return* ((e, n), (d, n)) *# 公钥（e, n），私钥（d, n）*

#### 2.签名

为了保证签名的消息长度固定且内容不被篡改，需要先对我的原始消息（姓名+学号）进行哈希运算。我在这里使用了 SHA-256 算法，并将得到的十六进制哈希摘要转换成一个大整数，以便进行数学运算。

    def **string\_to\_int\_hash**(*text*):

*import* hashlib

        hash\_obj = hashlib.sha256(*text*.encode('utf-8'))

*# 将十六进制摘要转为整数*

*return* int(hash\_obj.hexdigest(), 16)

在进行哈希处理后，我使用私钥 (d, n) 对消息哈希值进行加密（本质是模幂运算），得到数字签名。

公式为：签名 = (消息哈希 ^ d) mod n

    def **power**(*a*, *b*, *m*):

        res = 1

*a* %= *m*

*while* *b* > 0:

*if* *b* % 2 == 1:

                res = (res \* *a*) % *m*

*a* = (*a* \* *a*) % *m*

*b* //= 2

*return* res

    def **rsa\_sign**(*message\_hash*, *private\_key*):

        d, n = *private\_key*

*# 签名 = hash^d mod n*

        signature = power(*message\_hash*, d, n)

*return* signature

#### 3. 验证

我使用公钥 (e, n) 对数字签名进行解密，得到一个哈希值。然后将这个解密出的哈希值与原始消息的哈希值进行比对。如果两者完全相等，则签名验证成功，说明消息未被篡改且确实由对应的私钥持有者发出。

如果不相等，则验证失败。

    def **rsa\_verify**(*message\_hash*, *signature*, *public\_key*):

        e, n = *public\_key*

        decrypted\_hash = power(*signature*, e, n)

*return* decrypted\_hash == *message\_hash*

    is\_valid = rsa\_verify(message\_hash\_int, signature, public\_key)

*if* is\_valid:

        print("\nSignature verification successful!")

*else*:

        print("\nSignature verification failed.")

#### 方法2：调用现有密码库 (Cryptodome)

#### 密钥生成

我直接调用库函数 RSA.generate()，只需指定密钥的位数（这里是1024位）。库会自动完成所有复杂的素数生成、n、e、d 的计算。

*from* Cryptodome.PublicKey *import* RSA

*# 1. 生成 RSA 密钥对（1024 位）*

key = RSA.generate(1024)

#### 签名

1. 哈希处理

区别于自主实现，我调用库提供的 SHA256.new() 来创建一个哈希对象。注意这里得到的是一个对象，而不是像方式一那样的整数。

*from* Cryptodome.Hash *import* SHA256

plaintext = name + student\_id

hash\_obj = SHA256.new(plaintext.encode('utf-8'))

1. 生成签名

首先，从密钥对象中导入私钥，然后使用 pkcs1\_15.new() 创建一个签名器，之后调用签名器的 sign() 方法，并传入哈希对象。

*from* Cryptodome.Signature *import* pkcs1\_15

*# 导出私钥内容（用于演示或存储）*

    private\_key\_data = key.export\_key()

*# 从数据中重新导入私钥，模拟真实场景*

    loaded\_private\_key = RSA.import\_key(private\_key\_data)

    signer = pkcs1\_15.new(loaded\_private\_key)

signature = signer.sign(hash\_obj)

#### 验证

调用验证器的 verify() 方法，传入原始哈希对象和签名。如果签名无效，该方法会抛出 ValueError 或 TypeError 异常。因此，需要使用 try...except 结构来捕获异常并判断验证结果。

*try*:

        verifier.verify(hash\_obj, signature)

        print("\nSignature verification successful!")

*except* (ValueError, TypeError):

        print("\nSignature verification failed.")

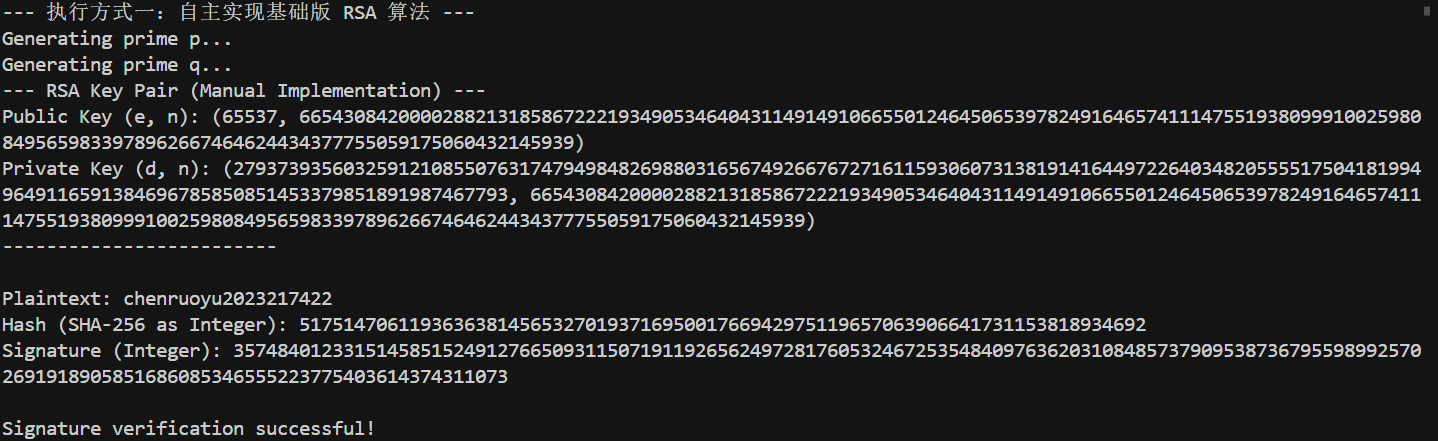
1. **原文内容**

姓名：chenruoyu

学号：2023217422

1. **结果验证**

#### 方法1：自主实现基础版RSA：



程序首先计算出明文 chenruoyu2023217422 的哈希值Hash (SHA-256 as Integer):

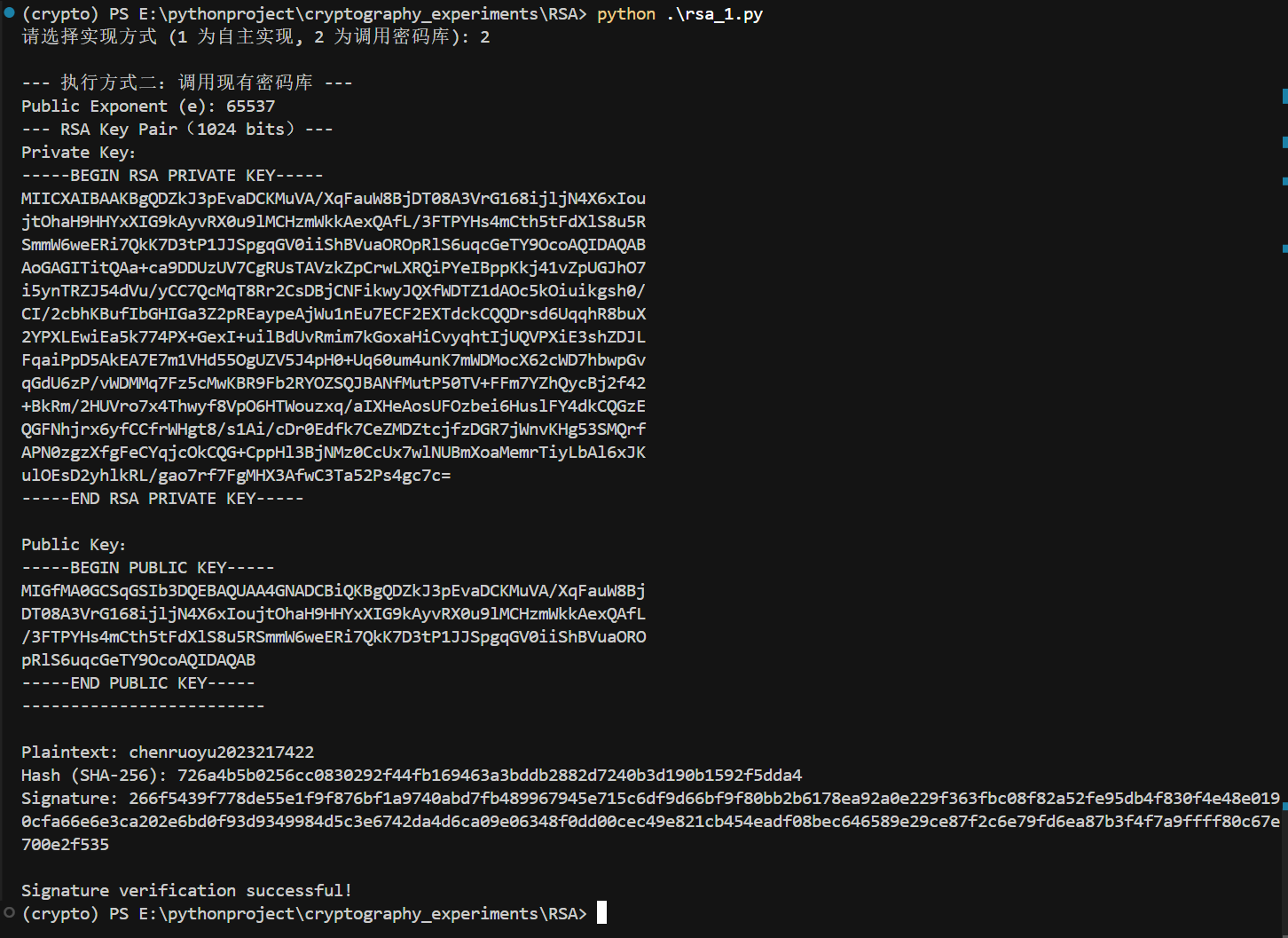
51751470611936363814565327019371695001766942975119657063906641731153818934692

之后程序在后台调用了 rsa\_sign 函数。这个函数接收了上面的 Hash 和我的 Private Key，然后执行了 (Hash ^ d) mod n 的数学运算并生成了最终签名，如上图所示。

同时在验证的时候重新进行将原始哈希和签名进行比较，二者的哈希值相等，故最后程序输出：

Signature verification successful!

#### 方法2：调用现有密码库 (Cryptodome)



方式二同样在验证的时候重新进行将原始哈希和签名进行比较，二者的哈希值相等，故最后程序输出：

Signature verification successful!

在这里的生成签名流程和方式一相似，但是存在不同的几个地方:

1. 密钥的表示方式

方式一中公钥和私钥直接以大整数元组 (Tuples of Integers) 的形式展示；而方式二中密钥被格式化为 PEM (Privacy-Enhanced Mail) 标准格式的文本块。

1. 哈希值的表示方式

方式一中哈希值被转换为一个巨大的整数；而方式二中哈希值以标准的十六进制字符串表示。

1. 签名的表示方式

方式一中签名本身为一个巨大的整数；而方式二中签名为标准的十六进制字符串。

1. **实验提问**
2. 加密与签名的区别

信息加密与数字签名的核心区别在于它们的目的和所使用的密钥完全相反：信息加密是为了保密，确保只有特定的人能看懂内容，它使用接收方的公钥进行加密，只有持有对应私钥的接收方才能解密；而数字签名是为了验证身份和防止篡改，确保信息来源可靠且未被修改，它使用发送方的私钥进行签名，任何人都可以用发送方的公钥来验证该签名的真伪。简而言之，加密是“给信件上锁”，保证内容不外泄；签名是“在信封上盖章”，证明这封信是谁写的。

1. 在RSA中对于素数P和Q选择的要求：

首先，P和Q必须是数学上严格的素数，这是RSA算法能够成立的数学基础。其次，它们必须非常“大”，通常每个都有几百个十进制位（例如，在2048位RSA中，P和Q各自都是1024位的二进制数）。选择巨大素数的目的是为了让它们的乘积 N 同样巨大，使得对 N 进行因式分解在计算上变得不可行。同时，P和Q必须是随机选择的，以防止攻击者通过猜测或缩小范围来找到它们。

除了P和Q本身要大，它们两个数之间的差值 |P-Q| 也必须足够大。如果P和Q非常接近，那么它们的乘积N就会约等于P的平方。攻击者可以利用这一点，通过费马分解法（Fermat's Factorization Method）等特殊算法，从N的平方根附近开始搜索，从而快速地找到P和Q。因此，为了抵御此类攻击，P和Q不能是相邻或相近的素数，必须在数值上有相当大的距离。

P-1和Q-1的素因子不能都是很小的数。这是为了防范一种名为“Pollard's p-1”的分解算法。该算法的攻击效率依赖于N的某个素因子P所对应的P-1能否被小的素数（或它们的幂）整除。如果P-1的所有素因子都很小，那么这个算法就能高效地分解出P。因此，在生成P和Q时，需要确保P-1和Q-1各自都至少包含一个足够大的素因子，这样的P和Q有时也被称为“强素数”（Strong Primes）。

1. 为什么e的选取为65537呢？

加密性能高：65537 的二进制表示非常特别，作为2的16次方加一，写成二进制就是 10000000000000001。这个数只有两位是 1。在进行 RSA 加密运算（即计算消息的 e 次方再取模）时，这个特性会大大提高运算效率。使用“平方-乘法”（square-and-multiply）算法时，计算m的e次方只需要进行 16 次平方和 1 次乘法，速度远快于选择一个随机的、拥有很多 1 的二进制数作为 e。

安全性好：虽然使用更小的 e（比如 3）会使加密更快，但过小的 e 在某些特定情况下可能会导致安全漏洞（例如，当加密的明文本身也很小时，或者同一个消息用不同密钥但相同的e=3加密后广播给多人时）。65537 在性能和安全性之间提供了一个很好的平衡，目前没有已知的针对这个值的有效攻击。

简化密钥生成：选择 e 的一个关键要求是它必须与 phi = (p-1)\*(q-1) 互质（即 gcd(e, phi) = 1）。因为 65537 本身是一个素数，所以要让它和 phi 不互质，phi 必须是 65537 的倍数。在 p 和 q 是随机生成的大素数的情况下，(p-1) 或 (q-1) 恰好是 65537 的倍数的概率极低。因此，选择 65537 几乎总能满足互质条件，简化了密钥对的生成过程。

1. **实验收获**

在本次实验中，我分别实现了自主版 RSA 算法和调用密码库的 RSA 实现。我通过实验可以明显感受到，密钥长度对运行时间有显著影响。以自主实现为例，生成 512 位素数时，素性测试和大数生成的过程相对较快，通常几秒内可以完成；而如果将密钥长度提升到 1024 位或更高，素数生成的时间会大幅增加，甚至可能需要几十秒甚至更久。这是因为大数素性测试（如 Miller-Rabin 算法）在位数增加时，随机性和复杂度都大幅提升。相比之下，调用密码库（如 Cryptodome）时，虽然底层也做了大量优化，但密钥长度越大，生成和签名/验签的耗时也会增加。因此，在实际应用中需要在安全性和性能之间做权衡：密钥越长安全性越高，但效率越低。

在实验中我遇到了以下调整，并实现了解决：

1.大素数生成与素性测试：在自主实现 RSA 时，最具挑战性的部分是大素数的生成。初期采用简单的随机数加素性测试，发现效率极低。后来引入了 Miller-Rabin 素性测试算法，并通过调整测试轮数（k 值）来平衡安全性和效率。此外，为了避免生成相同的 p、q，我还需在生成 q 时确保其与 p 不同。

2.模逆运算的实现：我计算私钥 d 时，需要求 e 关于 φ(n) 的模逆。刚开始直接用欧几里得算法，发现无法处理所有情况。后来查阅资料，采用了扩展欧几里得算法，成功解决了模逆不存在或计算错误的问题。

3.哈希与签名格式统一：我在签名时，需将消息先做哈希处理。自主实现时，需将哈希值转为整数参与运算，且要保证与库函数的哈希方式一致（如都用 SHA-256）。否则会导致签名和验签不一致。

4.与密码库的对比：我在调用 Cryptodome 库时，发现其对密钥长度有最低要求（1024 位），且签名格式为字节串。与自主实现的整数签名不同，需要注意数据类型的转换和输出格式的统一。故在这里我在自主实现方法中密钥位数为512位，调用密码库方法中密钥位数为1024位。

1. **参考文献**

[1] 杨波, 现代密码学(第4版), 清华大学出版社, 2017  
[2] 陈恭亮. 简明信息安全数学基础. 高等教育出版社，2011  
[3] Douglas R.Stinson. 密码学原理与实践. 电子工业出版社, 2009