**西安电子科技大学网信院**

**信息安全基础与密码学**

**综合实验**

**实 验 报 告（四）**

**ElGamal公钥密码算法**

**班级：2118021**

**姓名：盖乐**

**学号：21009200991**

**日期：2024.10.19**

一、实验目的（包括实验环境、实现目标等等）

1. 实验环境
   1. Windows11
   2. Python3.12
2. 实现目标
   1. 通过编写代码实现EIGamal公钥密码算法，加深对EIGamal算法的理解，体会该算法在解决实际问题的价值；
   2. 将密码学和数学知识相联系，并灵活运用到密码学的设计方案中；
   3. 提高实践能力和逻辑思维能力。

二、方案设计

1. 背景

ElGamal算法是由Tather ElGamal在1985年提出的，它是一种基于离散对数难题的加密体系，与RAS算法一样，既能用于数据加密，也能用于数字签名。ElGamal算法是基于因数分解，而ElGamal算法是基于离散对数问题。与RSA算法相比，ElGamal算法哪怕是使用相同的私钥，对相同的明文进行加密，每次加密后得到的签名也各不相同，有效的防止了网络中可能出现的重放攻击。

1. 离散对数困难问题

设g是模p的一个原根，任一整数h：

给定整数x，计算元素很容易；

给定整数h，计算整数x，0 ≤ x ≤ P - 2，使得非常困难。

1. ElGamal加密算法
   1. 密钥生成
      1. 随机产生一个大素数p以及模p的一个原根g
      2. 随机选取整数a, 计算ga(mod p)

此时，公钥是(p, g, ga),私钥是a。

* 1. 加密过程
     1. 随机选取一个整数k，1≤k≤p-2
     2. 计算，

此时，密文为()

* 1. 解密过程
     1. 计算V
     2. 计算m

1. 本次实验主要实现ElGamal的全流程，因此分为生成密钥、加密、解密三个过程。

三、方案实现

1. 算法流程图

图示

描述已自动生成

1. 主要函数
   1. 快速幂取模算法,这里使用位操作，相比于直接使用乘法的快速幂取模算法，更加高效，这很重要，内置的pow效率不高

def fast\_power\_mod(x, n, Mod):  
 res = 1  
 x %= Mod  
 while n != 0:  
 if n & 1:  
 res = (res \* x) % Mod  
 n >>= 1  
 x = (x \* x) % Mod  
 return res

* 1. 生成强素数p，使用sympy库中的randprime函数生成素数。

但是**2q+1不一定是素数**，所以需要再次判断，这里**调用了exp1中的费马素性检验**。

此处寻找强素数会需要**大量时间**，这是程序时间开销无法保证的根本原因。

这里q的范围在10^149到150之间，其实是为了保证p是150位的，有些死板了。

def generate\_p():  
 while True:  
 q = randprime(pow(10, 149), pow(10, 150))  
 p = 2 \* q + 1  
 if len(str(p)) == 150 and fpt.fermat\_primality\_test(p, 3, True):  
 break  
 return p

* 1. 生成p的一个原根g

随机选取g看上去时间开销无法接受，但其实，对于2\*q+1而言，其原根有q-1个，所以**每次随机选取到原根的概率有1/2**，是可以接受的。

def generate\_g(p):  
 while True:  
 g = random.randint(2, p - 2)  
 q = (p - 1) // 2  
 if fast\_power\_mod(g, 2, p) != 1 and fast\_power\_mod(g, q, p) != 1:  
 return g

* 1. 加密算法

这里不使用python内置的pow函数，而是使用自己实现的快速幂取模算法，前者效率太低

def encrypt(p, g, g\_a, m):  
 k = random.randint(1, p - 2)  
 print("k =", k)  
 c1 = fast\_power\_mod(g, k, p)  
 print("c1 =", c1)  
 temp = fast\_power\_mod(g\_a, k, p)  
 c2 = (m \* temp) % p  
 print("c2 =", c2)  
 return c1, c2

* 1. 解密算法

计算m时，是pow(v,-1,p),指的是v模p的逆元

def decrypt(p, c1, c2, a):  
 v = fast\_power\_mod(c1, a, p)  
 m = (c2 \* pow(v, -1, p)) % p  
 return m

* 1. 主函数

主要是遍历读取文件，以及实现密钥生成、加密、解密函数的调用。

if \_\_name\_\_ == "\_\_main\_\_":  
 folder\_path = r"test\_data"  
 files = os.listdir(folder\_path)  
 for file\_name in files:  
 file\_path = os.path.join(folder\_path, file\_name)  
 print(file\_name)  
 with open(file\_path, 'r') as f:  
 m = int(f.read())  
 p = generate\_p()  
 print("p =", p)  
 g = generate\_g(p)  
 print("g =", g)  
 a = random.randint(pow(10, 149), pow(10, 150))  
 g\_a = fast\_power\_mod(g, a, p)  
 print("g^a =", g\_a)  
 c1, c2 = encrypt(p, g, g\_a, m)  
 result = decrypt(p, c1, c2, a)  
 print("m =", m)  
 print("decrypted\_result =", result)  
 if m == result:  
 print("成功解密")  
 else:  
 print("解密失败")  
 print("------------------------------------------------------")

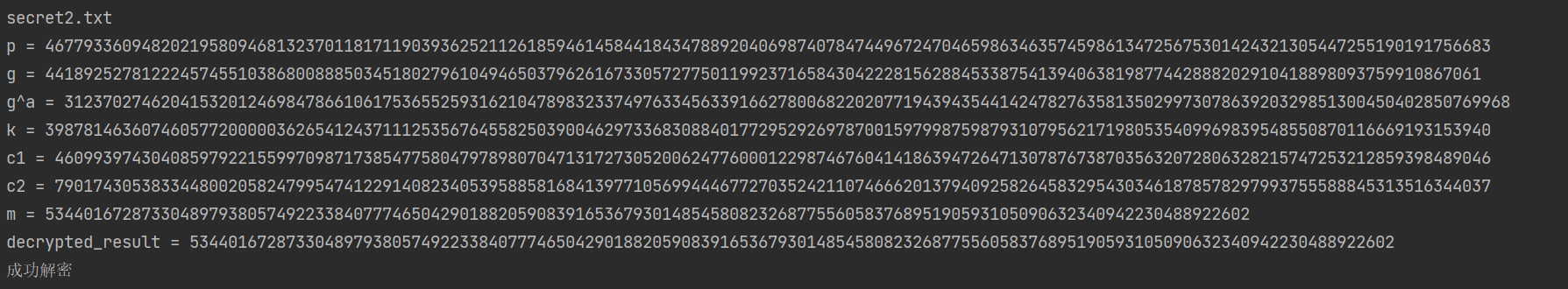
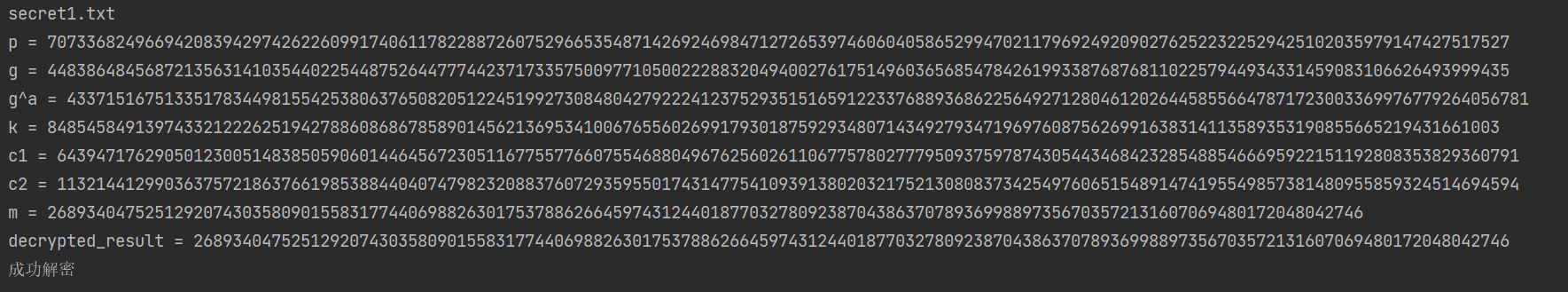
1. 完整代码

import os  
  
from sympy import randprime  
import random  
import fermat\_primality\_test as fpt  
  
  
# 快速幂取模算法  
# 这里使用位操作，相比于直接使用乘法，更加高效  
def fast\_power\_mod(x, n, Mod):  
 res = 1  
 x %= Mod  
 while n != 0:  
 if n & 1:  
 res = (res \* x) % Mod  
 n >>= 1  
 x = (x \* x) % Mod  
 return res  
  
  
# 生成强素数p  
# 使用sympy库中的randprime函数生成素数  
# 但是2q+1不一定是素数，所以需要再次判断，这里调用了exp1中的费马素性检验  
# 此处寻找强素数会需要大量时间，主要原因是2q+1不一定是素数，所以需要多次尝试  
def generate\_p():  
 while True:  
 q = randprime(pow(10, 149), pow(10, 150))  
 p = 2 \* q + 1  
 if len(str(p)) == 150 and fpt.fermat\_primality\_test(p, 3, True):  
 break  
 return p  
  
  
# 计算一个原根g  
# 随机选取g看上去时间开销无法接受，但其实，对于2\*q+1而言，其原根有q-1个，所以每次随机选取到原根的概率有1/2，是可以接受的  
def generate\_g(p):  
 while True:  
 g = random.randint(2, p - 2)  
 q = (p - 1) // 2  
 if fast\_power\_mod(g, 2, p) != 1 and fast\_power\_mod(g, q, p) != 1:  
 return g  
  
  
# 加密算法  
# 这里不使用python内置的pow函数，而是使用自己实现的快速幂取模算法，前者效率太低  
def encrypt(p, g, g\_a, m):  
 k = random.randint(1, p - 2)  
 print("k =", k)  
 c1 = fast\_power\_mod(g, k, p)  
 print("c1 =", c1)  
 temp = fast\_power\_mod(g\_a, k, p)  
 c2 = (m \* temp) % p  
 print("c2 =", c2)  
 return c1, c2  
  
  
# 解密算法  
# 计算m时，是pow(v,-1,p),指的是v模p的逆元  
def decrypt(p, c1, c2, a):  
 v = fast\_power\_mod(c1, a, p)  
 m = (c2 \* pow(v, -1, p)) % p  
 return m  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 folder\_path = r"test\_data"  
 files = os.listdir(folder\_path)  
 for file\_name in files:  
 file\_path = os.path.join(folder\_path, file\_name)  
 print(file\_name)  
 with open(file\_path, 'r') as f:  
 m = int(f.read())  
 p = generate\_p()  
 print("p =", p)  
 g = generate\_g(p)  
 print("g =", g)  
 a = random.randint(pow(10, 149), pow(10, 150))  
 g\_a = fast\_power\_mod(g, a, p)  
 print("g^a =", g\_a)  
 c1, c2 = encrypt(p, g, g\_a, m)  
 result = decrypt(p, c1, c2, a)  
 print("m =", m)  
 print("decrypted\_result =", result)  
 if m == result:  
 print("成功解密")  
 else:  
 print("解密失败")  
 print("------------------------------------------------------")

四、数据分析

每个文件输出11行，分别是：

1. 读入的文件名称
2. 强素数p
3. 强素数p的一个原根g
4. ga(mod p)
5. 随机选取的k
6. 加密后的C1
7. 加密后的C2
8. 明文m
9. 解密后的结果
10. 是否解密成功
11. 分隔符



五、思考与总结

1. 请简述什么是本原根，给定素数P，如何求其本原根？。

设m是正整数，a是整数，若a模m的阶等于φ(m)，则称a为模m的一个本原根。每个素数p都有本原根，而且刚好有φ(p-1)个模p的本原根。

求素数P的原根：对p-1素因子分解，求出x-1所有不同的质因子p1,p2...pm，对于任何2<=a<=x-1,判定a是否为x的原根，只需要检验a^((x-1)/p1),a^((x-1)/p2),...a^((x-1)/pm)这m个数中，是否存在一个数mod x为1，若存在，a不是x的原根，否则就是x的原根。

但是这里P很大，计算出全部原根很费时间，实际上，我们并不需要给出全部的原根，我们只需要找出一个原根即可，因此我们可以采用**检测**的方法，随机生成一个数，判断其是否为P的原根。

* + 1. 随机生成大质数q，直到p=2×q+1也是素数
    2. 随机选取1<g<p-1,那么g的阶必然是p-1=2q的因数，也就是g的阶只有2，q，2q三种可能
    3. 若或，说明g不是p的原根。否则就是p的原根。

这里随机选取g的形式虽然看起来时间开销无法保障，但是对于p=2×q+1，其原根的个数有q-1之多，所以每次随机选取有1/2的概率选到原根，不会花费很久。

1. 如果𝑘与𝑝−1不互素，可能会发生什么情况？

在ElGamal加密算法中，k与p-1不互素意味着k与p-1有公共因子。如果攻击者知道k与p-1的公共因子，那么攻击者就可以利用这一公共因子来求解密钥x，进而破解ElGamal加密算法。

具体来说，攻击者可以利用以下公式来求解密钥x：

x = (k \* (p-1) / gcd(k, p-1)) % p

其中，gcd(k, p-1)表示k与p-1的最大公因子。

例如，假设p=11，g=2，k=4，则p-1=10。gcd(k, p-1)=2，因此x=(4 \* 10 / 2) % 11 = 20 % 11 = 8。

攻击者可以利用计算出的密钥x来解密密文，从而获取明文。

因此，在ElGamal加密算法中，k与p-1必须互素，否则会导致加密算法的安全性降低。

为了确保k与p-1互素，可以采用以下方法：

在生成公钥时，随机选择k，并检查k与p-1是否互素。如果不互素，则重新生成k。

在加密时，随机选择k，并检查k与p-1是否互素。如果不互素，则重新生成k。

此外，也可以在ElGamal加密算法中引入一个新的参数r，并将k替换为r。其中，r与p-1互素，且1≤r≤p-2。这种方法可以进一步提高ElGamal加密算法的安全性。

1. 实验过程中还遇到了什么问题，如何解决的？通过该实验有何收获？
   1. 得到p=2\*q+1时，没有判断p是否为素数，导致后续v模p的逆元不存在，增加对p的判断，普通判断十分缓慢，使用了实验1的费马素性检验。
   2. 调用smypy的求一个数最小原根的方法，耗时十分夸张，寻找到了新的方法。
      1. 随机生成大质数q，直到p=2×q+1也是素数
      2. 随机选取1<g<p-1,那么g的阶必然是p-1=2q的因数，也就是g的阶只有2，q，2q三种可能
      3. 若或，说明g不是p的原根。否则就是p的原根。

这里随机选取g的形式虽然看起来时间开销无法保障，但是对于p=2×q+1，其原根的个数有q-1之多，所以每次随机选取有1/2的概率选到原根，不会花费很久。

* + 1. 这里花费时间长的地方在于第一步，“随机生成大质数q，直到p=2×q+1也是素数”