**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（五）**

**SM2椭圆曲线公钥密码算法**

**班级：2118021**

**姓名：盖乐**

**学号：21009200991**

**日期：2024.10.26**

一、实验目的

1. 实验环境
   1. Windows11
   2. Python3.12
2. 实现目标
   1. 通过编写代码实现SM2椭圆曲线公钥密码算法，加深对SM2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. SM2算法简介

SM2算法是中国国家密码局推出的国产化算法，是基于椭圆曲线的非对称算法，相对于RSA算法，SM2具有密钥更小，运算速度更快，相同密钥长度下具有更高安全性等优势。

1. SM2详解

SM2算法定义了两条椭圆曲线，一条基于F§上的素域曲线，一条基于F2^m上的拓域曲线，目前使用最多的曲线为素域曲线，这次实验我使用的算法基于素域曲线上的运算，素域曲线方程定义如下

此时，

* + 1. 加法单位元是整数0；
    2. 乘法单位元是整数1；
    3. 域元素的加法是整数的模p加法，即若则 a+b = (a+b)mod p;
    4. 域元素的乘法是整数的模p乘法，即若则 a·b = (a·b)mod p;
    5. 非互逆的不同点相加规则：

* + 1. 倍点规则：
    2. 要求：

1. SM2加解密过程
   1. 加密过程

文本

描述已自动生成

1. 解密过程

文本

描述已自动生成

三、方案实现

1. 算法流程图

图片包含 日历

描述已自动生成图示

描述已自动生成图示

描述已自动生成

1. 主要函数
   1. 类型转换：

def bytes\_to\_bits(*bytes\_string*: bytes):  
 bits = ''  
 for byte in *bytes\_string*:  
 for i in range(8):  
 bit\_value = (byte >> (7 - i)) & 1  
 bits += str(bit\_value)  
 return bits  
  
  
def bits\_to\_bytes(*bits\_string*: str):  
 global l  
 byte\_string = b''  
 for i in range(0, len(*bits\_string*), 8):  
 byte = int(*bits\_string*[i:i + 8], 2)  
 byte\_string += bytes([byte])  
 # 如果字节串长度不足l，则左填充补0  
 if len(byte\_string) < l:  
 byte\_string = byte\_string.rjust(l, b'\x00')  
 return byte\_string  
  
  
def int\_to\_bytes(*x*: int):  
 global l  
 try:  
 byte\_string = *x*.to\_bytes(l, byteorder='big')  
 except OverflowError:  
 print("int\_to\_bytes: l设定太小，溢出")  
 return  
 return byte\_string  
  
  
def bytes\_to\_int(*byte\_string*: bytes):  
 integer = int.from\_bytes(*byte\_string*, byteorder='big')  
 return integer  
  
  
def field\_to\_bytes(*x*, *form*: int):  
 if *form* == 0:  
 return int\_to\_bytes(*x*)  
 else:  
 return bits\_to\_bytes(*x*)  
  
  
def bytes\_to\_field(*x*, *form*: int):  
 if *form* == 0:  
 return bytes\_to\_int(*x*)  
 else:  
 return bytes\_to\_bits(*x*)  
  
  
def hex\_to\_bits(*h*):  
 b\_list = []  
 for i in *h*:  
 b = bin(eval('0x' + i))[2:].rjust(4, '0')  
 b\_list.append(b)  
 b = ''.join(b\_list)  
 return b  
  
  
def bits\_to\_hex(*bits*):  
 decimal\_value = int(*bits*, 2)  
 hex\_num = hex(decimal\_value)  
 return hex\_num  
  
  
def mod\_inverse(*a*, *p*):  
 return pow(*a*, -1, *p*)  
  
  
def fraction\_to\_int(*numer*, *denom*):  
 global p  
 return (*numer* % p \* mod\_inverse(*denom*, p)) % p  
  
  
def point\_to\_bytes(*P*, *form*: int):  
 x, y = *P*[0], *P*[1]  
 x1 = int\_to\_bytes(x)  
 if *form* == 1:  
 yp = y & 1  
 if yp == 1:  
 pc = b'\x03'  
 else:  
 pc = b'\x02'  
 s = pc + x1  
 elif *form* == 0:  
 y1 = int\_to\_bytes(y)  
 pc = b'\x04'  
 s = pc + x1 + y1  
 else:  
 y1 = int\_to\_bytes(y)  
 yp = y & 1  
 if yp == 1:  
 pc = b'\x07'  
 else:  
 pc = b'\x06'  
 s = pc + x1 + y1  
 return s  
  
  
def bytes\_to\_point(*s*, *form*):  
 global a, b  
 PC = *s*[0]  
 if *form* == 1:  
 x = *s*[1:]  
 if PC == 0x02:  
 y\_hat = 0  
 elif PC == 0x03:  
 y\_hat = 1  
 else:  
 raise Exception("bytes\_to\_point: PC错误")  
 elif *form* == 0:  
 if PC == 0x04:  
 x\_length = (len(*s*) - 1) // 2  
 x = *s*[1: l + 1]  
 y = *s*[l + 1: 2 \* l + 1]  
 xp = bytes\_to\_field(x, 0)  
 yp = bytes\_to\_field(y, 0)  
 if (xp \*\* 3 + a \* xp + b - yp \*\* 2) % p != 0:  
 raise Exception("bytes\_to\_point: 点不在椭圆曲线上")  
 P = (xp, yp)  
 print("C1满足椭圆曲线方程")  
 return P  
 elif *form* == 2:  
 if PC == 0x06:  
 y\_hat = 0  
 elif PC == 0x07:  
 y\_hat = 1  
 else:  
 raise Exception("bytes\_to\_point: PC错误")

* 1. 在椭圆曲线上执行标量加法

def EC\_scalar\_add(*P*, *Q*):  
 global p  
 if *P* == 0:  
 return *Q* if *Q* == 0:  
 return *P* x1, y1, x2, y2 = *P*[0], *P*[1], *Q*[0], *Q*[1]  
 lmd = fraction\_to\_int((y2 - y1), (x2 - x1))  
 x3 = (lmd \*\* 2 - x1 - x2) % p  
 y3 = (lmd \* (x1 - x3) - y1) % p  
 result = (x3, y3)  
 return result

* 1. 在椭圆曲线上执行标量倍增

def EC\_scalar\_double(*P*):  
 global p, a  
 if *P* == 0:  
 return *P* x, y = *P*[0], *P*[1]  
 lmd = fraction\_to\_int((3 \* x \*\* 2 + a), (2 \* y))  
 x3 = (lmd \*\* 2 - 2 \* x) % p  
 y3 = (lmd \* (x - x3) - y) % p  
 result = (x3, y3)  
 return result

* 1. 在椭圆曲线上执行标量倍数运算

def EC\_scalar\_multiple(*P*, *k*: int):  
 global p  
 Q = 0  
 k\_bits = bin(*k*)[2:]  
 for i in k\_bits:  
 Q = EC\_scalar\_double(Q)  
 if i == '1':  
 Q = EC\_scalar\_add(*P*, Q)  
 return Q

* 1. 密钥派生函数kdf

def KDF(*Z*, *klen*: int):  
 v = 256  
 if *klen* >= (pow(2, 32) - 1) \* v:  
 raise Exception("密钥派生函数KDF出错，请检查klen的大小！")  
 ct = 0x00000001  
 t = ceil(*klen* / v)  
 H\_a = []  
 for i in range(t):  
 ct\_bytes = int\_to\_bytes(ct)  
 ct\_bits = bytes\_to\_bits(ct\_bytes)  
 # 上面填充多了，这里截取后32位  
 ct\_bits = ct\_bits[-32:]  
 s\_bits = *Z* + ct\_bits  
 s\_bytes = bits\_to\_bytes(s\_bits)  
 s\_list = [i for i in s\_bytes]  
 hash\_hex = sm3.sm3\_hash(s\_list)  
 hash\_bin = hex\_to\_bits(hash\_hex)  
 H\_a.append(hash\_bin)  
 ct += 1  
 if *klen* % v != 0:  
 H\_a[-1] = H\_a[-1][:*klen* - v \* floor(*klen* / v)]  
 K = ''.join(H\_a)  
 return K

* 1. SM2加密函数

def SM2\_encryption(*plaintext*):  
 # 明文：  
 M\_str = *plaintext* M\_bytes = M\_str.encode('utf-8')  
 M\_int = int(M\_bytes.hex(), 16)  
  
 # 用随机数发生器产生随机数k∈[1,n-1]  
 k = secrets.randbelow(n)  
 print("随机数k:", k)  
  
 # 计算椭圆曲线点C1=[k]G=(x1,y1)  
 C1\_point = EC\_scalar\_multiple(G, k)  
  
 # 将C1的数据类型转换为比特串  
 C1\_bytes = point\_to\_bytes(C1\_point, 0)  
 C1\_bits = bytes\_to\_bits(C1\_bytes)  
 C1 = bits\_to\_hex(C1\_bits)[2:]  
 print("C1\_x:", hex(C1\_point[0]))  
 print("C1\_y:", hex(C1\_point[1]))  
  
 # 计算椭圆曲线点[k]PB=(x2,y2)  
 S1 = EC\_scalar\_multiple(P\_B, k)  
 print("[k]PB\_x:", hex(S1[0]))  
 print("[k]PB\_y:", hex(S1[1]))  
 x2 = S1[0]  
 y2 = S1[1]  
  
 # 将坐标x2、y2 的数据类型转换为比特串  
 x2\_bytes = field\_to\_bytes(x2, 0)  
 x2\_bits = bytes\_to\_bits(x2\_bytes)  
 y2\_bytes = field\_to\_bytes(y2, 0)  
 y2\_bits = bytes\_to\_bits(y2\_bytes)  
  
 klen = 4 \* len(M\_bytes.hex())  
  
 # 计算t=KDF(x2 ∥ y2, klen)，若t为全0比特串，则返回A1  
 t = KDF(x2\_bits + y2\_bits, klen)  
 if all(bit == '0' for bit in t):  
 raise Exception("KDF返回了全0比特串，需重新生成k")  
 t\_hex = bits\_to\_hex(t)  
 print("加密中的t:", t\_hex)  
  
 # 计算C2 = M ⊕ t  
 C2\_int = M\_int ^ int(t\_hex, 16)  
 C2 = hex(C2\_int)[2:]  
 print("C2:", C2)  
 s = x2\_bytes + M\_bytes + y2\_bytes  
 hash\_list = [i for i in s]  
 M\_bits = bytes\_to\_bits(M\_bytes)  
 print("(x2 || M || y2):", bits\_to\_hex(x2\_bits + M\_bits + y2\_bits))  
  
 # 计算C3 = Hash(x2 ∥ M ∥ y2)  
 C3 = sm3.sm3\_hash(hash\_list)  
 print("C3:", C3)  
  
 # 输出密文C = C1 ∥ C2 ∥ C3  
 C = C1 + C2 + C3  
 print("密文:", C)  
 return C

* 1. SM2解密函数

def SM2\_decryption(*C*):  
 # 从C中取出比特串C1，将C1的数据类型转换为椭圆曲线上的点，验证C1是否满足椭圆曲线方程，若不满足则报错并退出  
 global l  
 h = 1  
 C\_pad = ''  
 if len(*C*) % 2 != 0:  
 C\_pad = "0" + *C* print("C\_pad:", C\_pad)  
 C\_bytes = bytes.fromhex(C\_pad)  
 print("C\_bytes:", C\_bytes)  
 C1\_length = 2 \* l + 1  
 C1\_hex\_length = 2 \* C1\_length  
 C1\_bytes = C\_bytes[:C1\_length]  
 print("C1\_HEX:", C\_pad[:C1\_hex\_length])  
 C1\_point = bytes\_to\_point(C1\_bytes, 0)  
  
 # 计算椭圆曲线点S=[h]C1，若S是无穷远点，则报错并退出；  
 S = EC\_scalar\_multiple(C1\_point, h)  
 if EC\_scalar\_add(S, G) == G:  
 raise Exception("S是无穷远点")  
  
 # 计算[dB]C1=(x2,y2)，将坐标x2、y2的数据类型转换为比特串  
 point = EC\_scalar\_multiple(C1\_point, dB)  
 print("[dB]C1\_x2:", hex(point[0]))  
 print("[dB]C1\_y2:", hex(point[1]))  
 x2 = point[0]  
 y2 = point[1]  
 x2\_bytes = field\_to\_bytes(x2, 0)  
 x2\_bits = bytes\_to\_bits(x2\_bytes)  
 y2\_bytes = field\_to\_bytes(y2, 0)  
 y2\_bits = bytes\_to\_bits(y2\_bytes)  
  
 C3\_hex\_length = 64  
 C2\_hex\_length = len(*C*) - C1\_hex\_length - C3\_hex\_length + 1  
 klen = 4 \* C2\_hex\_length  
  
 # 计算t = KDF(x2 ∥ y2, klen)，若t为全0比特串，则报错并退出  
 t = KDF(x2\_bits + y2\_bits, klen)  
 if all(bit == '0' for bit in t):  
 raise Exception("KDF返回了全0比特串")  
 t\_hex = bits\_to\_hex(t)  
 print("解密中的t:", t\_hex)  
  
 # 从C中取出比特串C2，计算M′= C2 ⊕ t；  
 C2\_int = int(*C*[C1\_hex\_length - 1:-C3\_hex\_length], 16)  
 print("C2:", *C*[C1\_hex\_length - 1:-C3\_hex\_length])  
 M\_m\_int = C2\_int ^ int(t\_hex, 16)  
 M\_m\_str = hex(M\_m\_int)[2:]  
 M\_m\_bytes = bytes.fromhex(M\_m\_str)  
  
 # 计算u = Hash(x2 ∥ M′ ∥ y2)，从C中取出比特串C3，若u ̸= C3，则报错并退出  
 s = x2\_bytes + M\_m\_bytes + y2\_bytes  
 hash\_list = [i for i in s]  
 u = sm3.sm3\_hash(hash\_list)  
 print("u:", u)  
 C3 = *C*[-C3\_hex\_length:]  
 print("C3:", C3)  
 if u != C3:  
 raise Exception("u != C3")  
 print("解密结果为：", M\_m\_bytes.decode())

四、数据分析

**输出格式：**

文件名

公钥：

私钥：

本次加密的明文：

随机数k:

C1\_x:

C1\_y:

[k]PB\_x:

[k]PB\_y:

加密中的t:

C2:

(x2 || M || y2):

密文:

C\_pad:

C\_bytes:

C1\_HEX:

C1满足椭圆曲线方程

[dB]C1\_x2:

[dB]C1\_y2:

解密中的t:

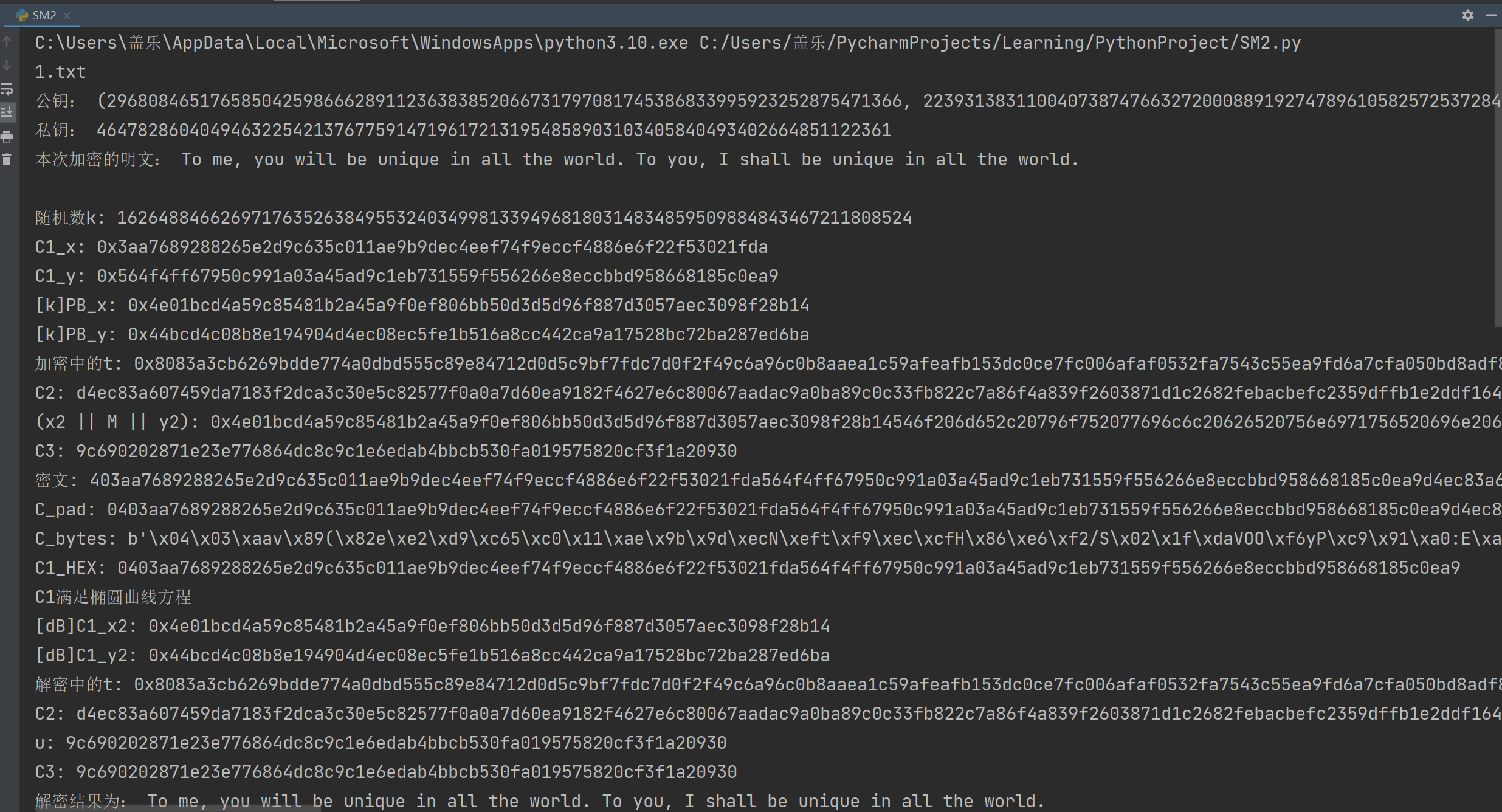
C2:

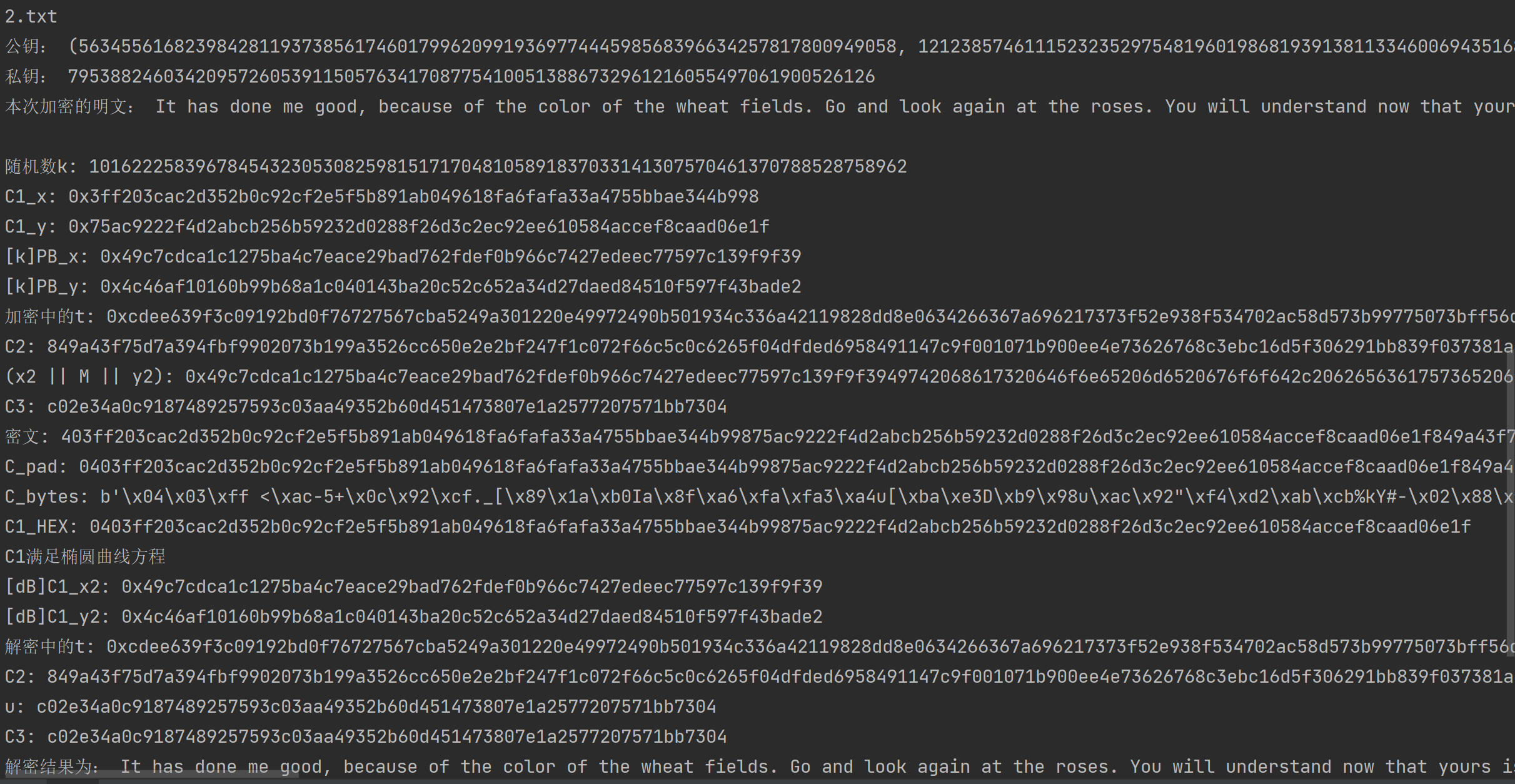
u:

C3:

解密结果为：

**运行结果截图：**





五、思考与总结

1. 实验过程中遇到了什么问题，如何解决的？
   * 1. 对于无穷远点的判断，我们使用P+S=P来判断S是无穷远点。体现在代码中：

# 计算椭圆曲线点S=[h]C1，若S是无穷远点，则报错并退出；  
S = EC\_scalar\_multiple(C1\_point, h)  
if EC\_scalar\_add(S, G) == G:  
 raise Exception("S是无穷远点")

* + 1. 对于密钥对，我们本来使用的是固定公钥的私钥的，但是经过改进，我们使用随机的密钥对，原理如图：

文本

描述已自动生成

代码如下：

dB = secrets.randbelow(n-2)  
P\_B = EC\_scalar\_multiple(G, dB)  
print("公钥：", P\_B)  
print("私钥：", dB)

1. 通过该实验有何收获和感想？

SM2 国密算法属于非对称加密算法，理解起来并不是很难。但是需要完整的把整个代码写出来是不容易的，需要清晰的思路，主要涉及多种类型的形式转换，如图：

图示, 示意图

描述已自动生成

写一个转换验证一个转换是否正确，防止后续debug非常费劲；另外就是实现椭圆曲线上的加法、倍增、倍数运算的实现，他们是在模p下的运算，套用公式的λ是分数，要化成整数；另一个重点就是密钥派生函数KDF的实现了，他其实就是分块，哈希运算，最后连接起来，主要注重是对于最后一个块的处理；

然后对于整个程序的实现就没有问题了，按照给到的加解密算法一步步写程序就好了。