**SM2椭圆曲线公钥密码算法**

一、实验目的（包括实验环境、实现目标等等）

实验环境：python3.12.7

实现目标：

1. 理解椭圆曲线密码学的基本原理：实现椭圆曲线上的点加法和标量乘法运算。
2. 对椭圆曲线密钥生成和验证
3. 实现基于椭圆曲线的加密和解密算法

二、方案设计

（包括背景、原理、必要的公式、图表、算法步骤等等）

1. 背景：椭圆曲线加密是一种基于椭圆曲线数学结构的公钥密码。与传统的RSA算法相比，ECC在提供相同安全强度的情况下，使用的密钥长度更短，具有更高的计算效率和更低的存储需求。
2. 原理：

**椭圆曲线上的点运算**

1. **点加法**：给定椭圆曲线上的两个点 和，点加法 的计算规则如下：

* 如果是无穷远点，则。
* 如果是无穷远点，则。
* 如果 且，则。
* 如果，则：
* 如果，则：

**标量乘法**： 给定椭圆曲线上的点 和整数，标量乘法表示将点加次。可以通过重复使用点加法实现。

**密钥生成:**

1. **私钥**：随机选择一个整数作为私钥，满足，其中是椭圆曲线的阶。
2. **公钥**：计算公钥，其中是椭圆曲线上的基点。

**公钥验证:**

1. 验证公钥不是无穷远点。
2. 验证公钥的坐标和是域中的元素。
3. 验证满足椭圆曲线方程。
4. 验证。
5. 算法步骤：

**加密算法过程**

设需要发送的消息为比特串 为的比特长度。

对明文 M 进行加密，作为加密者的用户A应实现以下运算步骤：

A1: 用随机数发生器产生随机数;

A2: 计算椭圆曲线点，按文献公式将​数据类型转换为比特串;

A3: 计算椭圆曲线点，若S是无穷远点，则报错并退出;

A4: 计算椭圆曲线点，按文献中公式，将坐标数据类型转换为比特串;

A5: 计算，若为全比特串，则返回A1;

A6: 计算;

A7: 计算;

A8: 输出密文。

**解密算法**

设为密文中的比特长度。为了对密文进行解密，作为解密者的用户B应实现以下运算步骤：

B1: 从*C* 中取出比特串​，按文献将的数据类型转换为椭圆曲线上的点，验证是否满足椭圆曲线方程，若不满足则报错并退出；

B2: 计算椭圆曲线点​，若是无穷远点，则报错并退出；

B3: 计算，按文献将的数据类型转换为比特串；

B4: 计算，若为全 比特串，则报错并退出；

B5: 从中取出比特串​，计算；

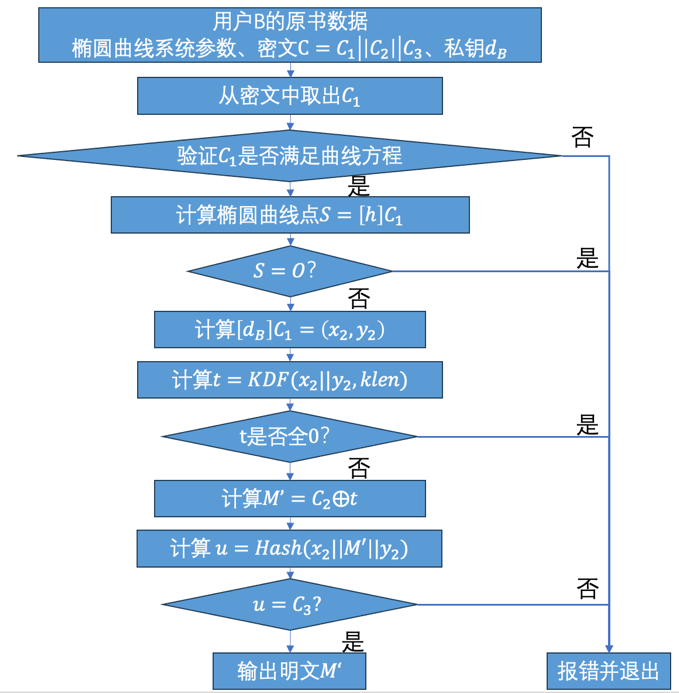
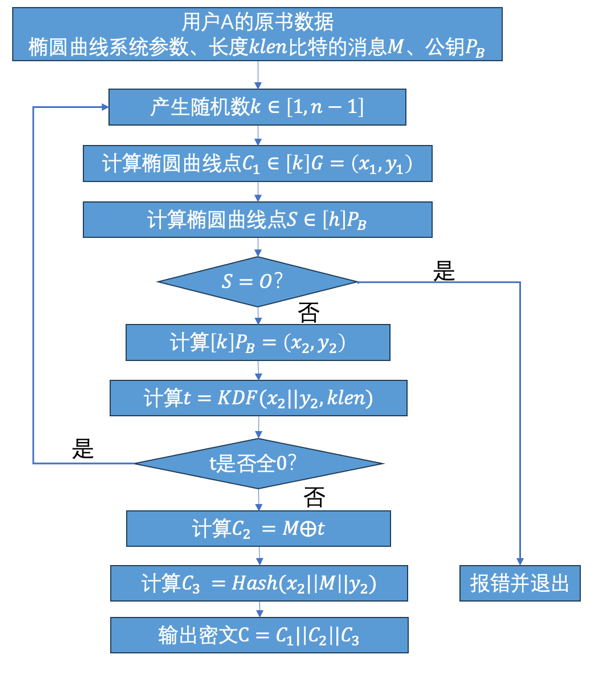
B6: 计算，从中取出比特串，若，则报错并退出；

B7: 输出明文。

三、方案实现

（包括算法流程图、主要函数的介绍、算法实现的主要代码等等）

1. 加解密算法流程图：（左图为加密，右图为解密）



1. 主要函数介绍及实现代码
   1. 倍点加法运算

def add(x1, y1, x2, y2):

if x1 == math.inf:

return x2,y2

if x2 == math.inf:

return x1,y1

if x1 == x2 and y1 == -y2:

return math.inf,math.inf

if x1 == x2 and y1 == y2:

l = (3 \* x1 \* x1 + a) \* egcd(2 \* y1, p)[0] % p

else:

l = (y2 - y1) \* egcd(x2-x1,p)[0] % p

x3 = l \* l - x1 - x2

y3 = l \* (x1 - x3) - y1

return x3 % p, y3 % p

* 1. 乘法运算

def mul(x1, y1, k):

k = bin(k)[2:]

x3,y3 = math.inf,math.inf

for i in k:

x3,y3 = add(x3,y3,x3,y3)

if int(i) == 1:

x3,y3 = add(x3,y3,x1,y1)

return x3,y3

* 1. 公钥私钥生成函数

def keygen(q):

d = random.randint(1,n-2)

# d = 0x1649AB77A00637BD5E2EFE283FBF353534AA7F7CB89463F208DDBC2920BB0DA0

xp,yp = mul(gx,gy,d)

if publickeytest(q,xp,yp) == False:

return keygen(q)

else:

return d,[xp,yp]

* 1. 检验公钥是否符合要求

def publickeytest(q,x1,y1):

if x1 == math.inf or y1 == math.inf:

return False

if x1 < 0 or x1 >= q-1:

return False

if y1 < 0 or y1 >=q-1:

return False

if y1\*y1 % q != (x1\*\*3 + a\*x1 + b) % q:

return False

xx,yy = mul(x1,y1,n)

if xx!= math.inf and yy != math.inf:

return False

return True

* 1. KDF函数

def KDF(Z,klen):

ct = 0x00000001

v = 256

num = klen//v

K = ''

for i in range(num):

ct\_bytes = ct.to\_bytes(4, byteorder='big')

K += sm3.sm3\_hash(func.bytes\_to\_list(Z + ct\_bytes))

ct += 1

if klen % v == 0:

return K

else:

ct\_bytes = ct.to\_bytes(4, byteorder='big')

Ha = sm3.sm3\_hash(func.bytes\_to\_list(Z + ct\_bytes))

K += Ha[:(klen-(v\*math.ceil(klen/v)))//4]

return K

* 1. 通过字节转化为曲线点

def bytes\_to\_point(S,l):

PC = S[:2]

x = int(S[2:l+1],16)

y = int(S[l+1:],16)

if PC != '04':

return False

return x,y

* 1. 从曲线上的点转化为字节串

def point\_to\_bytes(xp,yp):

xp = hex(xp)[2:]

yp\_bit = bin(yp)[2:]

yp = hex(yp)[2:]

ypp = yp\_bit[-1]

PC = '04'

S = PC + xp + yp

return S

* 1. 加密算法

def encrypt(M):

m = ''.join(format(ord(i),'08b') for i in M)

# m = format(int(binascii.hexlify(M.encode()), 16), '8b')

klen = len(m)

print(klen)

print(f'明文的二进制比特串为：{m}')

k = random.randint(1,n-1)

# k = 0x4C62EEFD6ECFC2B95B92FD6C3D9575148AFA17425546D49018E5388D49DD7B4F

print(f'随机数k为：{hex(k)}')

x1,y1 = mul(gx,gy,k)

# print(f'x1为:{hex(x1)}\ny1为:{hex(y1)})')

c1 = point\_to\_bytes(x1,y1)

# print(f'c1为:{c1}')

S = mul(PB[0],PB[1],h)

if S[0] == math.inf or S[1] == math.inf:

return False

x2,y2 = mul(PB[0],PB[1],k)

# print(f'x2为:{hex(x2)}\ny2为:{hex(y2)}')

x2\_to\_bytes = x2.to\_bytes(32, byteorder='big')

y2\_to\_bytes = y2.to\_bytes(32, byteorder='big')

string = x2\_to\_bytes + y2\_to\_bytes

t = KDF(string,klen)

# print(f't = {t}')

# print(hex(int(t,16)))

t = bin(int(t,16))[2:]

if t == '0'\*len(t):

raise ValueError('KDF输出全为0')

c2 = int(m,2) ^ int(t,2)

# print(f'c2为:{hex(c2)}')

m\_to\_bytes = int(m,2).to\_bytes((len(m) + 7) // 8, byteorder='big')

c3 = sm3.sm3\_hash(func.bytes\_to\_list(x2\_to\_bytes + m\_to\_bytes + y2\_to\_bytes))

# print(f'c3为:{c3}')

C = c1 + hex(c2)[2:] + c3

print(f'密文为:{C}')

return C

* 1. 解密算法

def decrypt(C):

l = math.ceil(math.log2(p)/8)

slen = 2\*l + 1

C1 = C[:2\*slen]

print(f'解密C1为:{C1}')

klen = len(C) - 2\*slen - 64

C2 = C[2\*slen:-64]

print(f'解密C2为:{C2}')

C3 = C[-64:]

print(f'解密C3为:{C3}')

xp,yp = bytes\_to\_point(C1,slen)

print(f'解密x1为:{hex(xp)}\ny1为:{hex(yp)}')

if yp \* yp % p != (xp\*\*3 + a\*xp + b) % p:

raise ValueError("解密失败，C1不是椭圆曲线上的点")

S = mul(xp,yp,h)

if S[0] == math.inf or S[1] == math.inf:

return False

x2,y2 = mul(xp,yp,d)

x2\_to\_bytes = x2.to\_bytes(32, byteorder='big')

y2\_to\_bytes = y2.to\_bytes(32, byteorder='big')

string = x2\_to\_bytes + y2\_to\_bytes

t = KDF(string,klen\*4)

if t == '0'\*len(t):

raise ValueError('KDF输出全为0')

print(t)

M = int(C2,16) ^ int(t,16)

print(f'解密明文为:{hex(M)}')

hex\_string = hex(M)[2:] # 将整数转换为十六进制字符串，并去掉前缀 '0x'

if len(hex\_string) % 2 != 0:

hex\_string = '0' + hex\_string # 确保长度为偶数

# 将十六进制字符串转换为字节串

byte\_data = bytes.fromhex(hex\_string)

# 将字节串进行 UTF-8 解码

utf8\_encoded = byte\_data.decode('utf-8')

print(f'解密明文的UTF-8编码为:{utf8\_encoded}')

u = sm3.sm3\_hash(func.bytes\_to\_list(x2\_to\_bytes + byte\_data + y2\_to\_bytes))

if u == C3:

print('解密成功')

else:

print('解密失败')

* 1. 主函数调用

if \_\_name\_\_ == '\_\_main\_\_':

M = input('请输入明文：')

C = encrypt(M)

decrypt(C)

四、数据分析(包括算法测试数据的分析，运行结果截图等等)

1.对6.txt文件测试



2.对7.txt的测试



3.对8.txt的测试



4.对9.txt的测试



5.对10.txt的测试



五、思考与总结

1. 实验过程中遇到了什么问题，如何解决的？

答：实验过程中我遇到了在验证公钥密码的是否合法的时候倍的是否取到无穷远的情况判断失误，后来在老师的帮助下，我选择对横坐标和纵坐标分别验证就解决了这个问题。

1. 通过该实验有何收获和感想？

答：本次实验我更加深刻理解了椭圆曲线SM2加密算法的原理，能够熟练的编写加解密函数，在本次实验中我发现了对输出比特串转换的时候需要格外注意比特串的长度，因为在十六进制数字转化为比特串时如果出现数字较小会丢失比特串前面可能存在的0位。所以说我们需要在进制转换的时候限制比特串的长度这样就不会出错了。