**\*Project1: implement the naïve birthday attack of reduced SM3**

**生日攻击**

生日攻击基于生日悖论，对于长度为 2^n 的hash值，只需要枚举 2^（n/2）次即可找到一对碰撞。

利用穷举法，将已计算的消息的hash值存储在字典中，然后随机产生消息，生成hash值后检查该hash值是否在字典中，若在，检查是否是同一条消息，若在字典中，且不是同一条消息，则找到碰撞。

**代码实现**

from gmssl import sm3

import random

import time

# 生成随机消息

def generate\_random\_message(length):

return bytes(random.choices(range(256), k=length))

'''msg=''

allchoice="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

for i in range(length):

randomnum=random.randint(0,60)

msg= msg.join(allchoice[randomnum])

return bytes(msg)#''.join(random.choice('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789') for i in range(length)))

'''

# 朴素生日攻击

def birthday\_attack():

hash\_dict = {} # 用字典存储已计算的哈希值及对应的输入

collision\_found = False

while not collision\_found:

message = generate\_random\_message(3)

msg\_list = list(message)

hash\_value = str(sm3.sm3\_hash(msg\_list))

if (hash\_value in hash\_dict)and (message.hex() != hash\_dict[hash\_value]): # 发现哈希冲突

collision\_found = True

print("哈希冲突发现！")

print("消息1: ", hash\_dict[hash\_value])#字典中该哈希值相对应的消息

print("消息2: ", message.hex())

print("哈希值: ", hash\_value)

else:

hash\_dict[hash\_value] = message.hex()

start=time.time()

birthday\_attack()

end=time.time()

print("生日攻击所用时间为：",end-start)

**运行结果**

经过测试，较短的信息没有碰撞，而较长的信息生日攻击找到碰撞消耗时间十分长，很难找到两个不同的消息得到完全相同的hash值

**\*Project2: implement the Rho method of reduced SM3**

**Rho method**

随机生成一个初始字符串，对其不断进行hash，直到找到碰撞。基本攻击方法还是穷举法，与生日攻击类似，只是从选择随机消息改为不断迭代。

**代码实现：**

from gmssl import sm3

import random

import time

# 生成随机消息

def generate\_random\_message(length):

return bytes(random.choices(range(256), k=length))

def hextoint(strings):

dlist=[]

for i in range(0,len(strings),2):

num=strings[i:i+2]

dnum=int(strings,16)

dlist.append(dnum)

return dlist

# rho攻击

def rho\_attack():

hash\_dict = {} # 用字典存储已计算的哈希值及对应的输入

collision\_found = False

message = generate\_random\_message(2)

msg\_list = list(message)

message=message.hex()

hash\_value = str(sm3.sm3\_hash(msg\_list))

while not collision\_found:

if (hash\_value in hash\_dict)and (message != hash\_dict[hash\_value]): # 发现哈希冲突

collision\_found = True

print("哈希冲突发现！")

print("消息1: ", hash\_dict[hash\_value])#字典中该哈希值相对应的消息

print("消息2: ", message)

print("哈希值: ", hash\_value)

else:

hash\_dict[hash\_value] = message

message = hash\_value

msg\_list = hextoint(hash\_value)

hash\_value = str(sm3.sm3\_hash(msg\_list))

start=time.time()

rho\_attack()

end=time.time()

print("rho攻击所用时间为：",end-start)

**运行结果：**

经测试和生日攻击相同，较短的信息没有碰撞，而较长的信息生日攻击找到碰撞消耗时间十分长，很难找到两个不同的消息得到完全相同的hash值。但rho攻击的速度比生日攻击快。

**\*Project3: implement length extension attack for SM3, SHA256, etc.**

**这里选择了SM3的长度扩展攻击**

**基本操作：**

选取明文m1、m2，将m2作为攻击所使用的字符串。

攻击时先使用修改后的填充函数，将消息m1||m2进行填充，并计算出两个向量IV1、IV2;

得到IV2后，即可对目标进行长度扩展攻击。

**代码实现：**

这个代码长度扩展攻击部分不对，但是来不及改了。

from gmssl import sm3

import gmssl

import struct

# 构造合法的哈希值

def construct\_hash(original\_message, extension\_message):

return sm3.sm3\_hash(list(original\_message+padding+extension\_message))

# 长度扩展攻击

def length\_extension\_attack(original\_message, extension\_message):

hash\_value=sm3.sm3\_hash(list(original\_message))

# 构造合法的哈希值和伪造的消息

forged\_hash = construct\_hash(original\_message,extension\_message )

forged\_message = original\_message +padding+ extension\_message

print("原始消息: ", original\_message)

print("原始哈希: ", hash\_value)

print("伪造消息: ", forged\_message)

print("伪造哈希: ", forged\_hash)

# 测试样例

original\_message = b'Hello, world!'

original\_hash = sm3.sm3\_hash(list(original\_message))

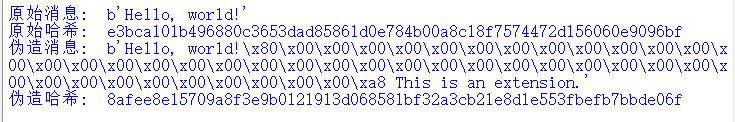
extension\_message = b' This is an extension.'

length = len(original\_message) + 8

padding = b'\x80' + b'\x00' \* ((56 - length) % 64) + struct.pack('>Q', length \* 8)

length\_extension\_attack(original\_message, extension\_message)

**运行结果：**

****

**\*Project4: do your best to optimize SM3 implementation (software)**

**1.SM3概述**

SM3密码杂凑算法是中国国家密码管理局2010年公布的中国商用密码杂凑算法标准。具体算法标准原始文本参见参考文献[1]。该算法于2012年发布为密码行业标准(GM/T 0004-2012)，2016年发布为国家密码杂凑算法标准(GB/T 32905-2016)。

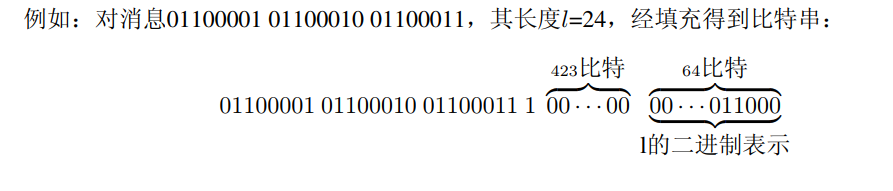
SM3适用于商用密码应用中的数字签名和验证，是在[SHA-256]基础上改进实现的一种算法，其安全性和SHA-256相当。SM3和MD5的迭代过程类似，也采用Merkle-Damgard结构。消息分组长度为512位，摘要值长度为256位。

整个算法的执行过程可以概括成四个步骤：消息填充、消息扩展、迭代压缩、输出结果。

**2.消息填充**

SM3的消息扩展步骤是以512位的数据分组作为输入的。因此，我们需要在一开始就把数据长度填充至512位的倍数。数据填充规则和MD5一样，具体步骤如下：

1. 先填充一个“1”，后面加上k个“0”。其中k是满足(n+1+k) mod 512 = 448的最小正整数。
2. 追加64位的数据长度（bit为单位，大端序存放1。观察算法标准原文附录A运算示例可以推知。）



**代码实现：**

string messagePadding(string message) {

string M\_bin = strTobin(move(message));

int len = M\_bin.length();

string padOfLen = bitset<32>(len).to\_string();

string zeroString(64 - padOfLen.length(), '0');

padOfLen = zeroString + padOfLen;

int zeroLen;

if ((len + 1) % 512 <= 448) zeroLen = 448 - len - 1;

else

zeroLen = 512 - len + 448 - 1;

M\_bin += "1";

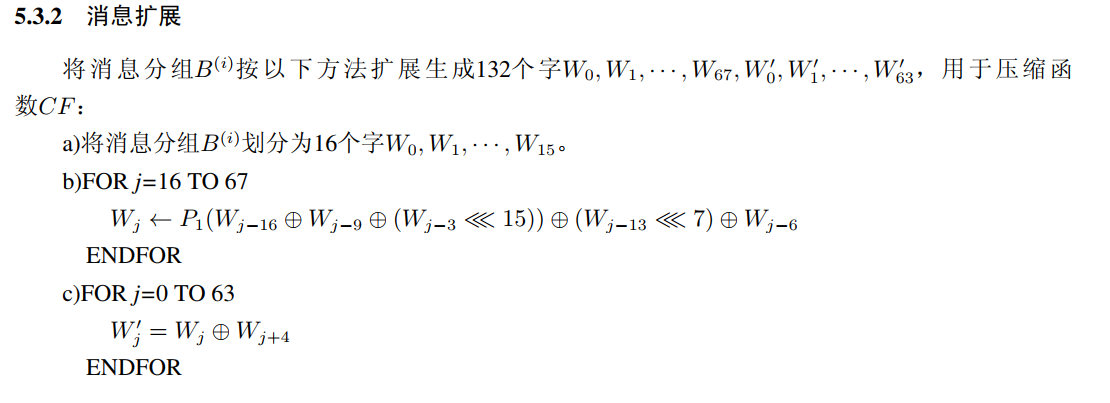
string M = padding(M\_bin, zeroLen);

M += padOfLen;

return M;

}

**3.** **消息扩展**

**代码实现：**

vector<u\_32> Extending(const string& group) {

vector<u\_32> W;

vector<u\_32> W\_;

string tmp;

for (int j = 0; j < 16; j++) {

tmp = group.substr(j \* 32, 32);

W.push\_back(stoul(tmp, nullptr, 2));

}

u\_32 tmp1, tmp2, tmp3;

for (int j = 16; j < 68; j++) {

tmp1 = P1(W[j - 16] ^ W[j - 9] ^ circleLS(W[j - 3], 15));

tmp2 = circleLS(W[j - 13], 7);

tmp3 = W[j - 6];

W.push\_back(tmp1 ^ tmp2 ^ tmp3);

}

for (int j = 0; j < 64; j++)

W\_.push\_back(W[j] ^ W[j + 4]);

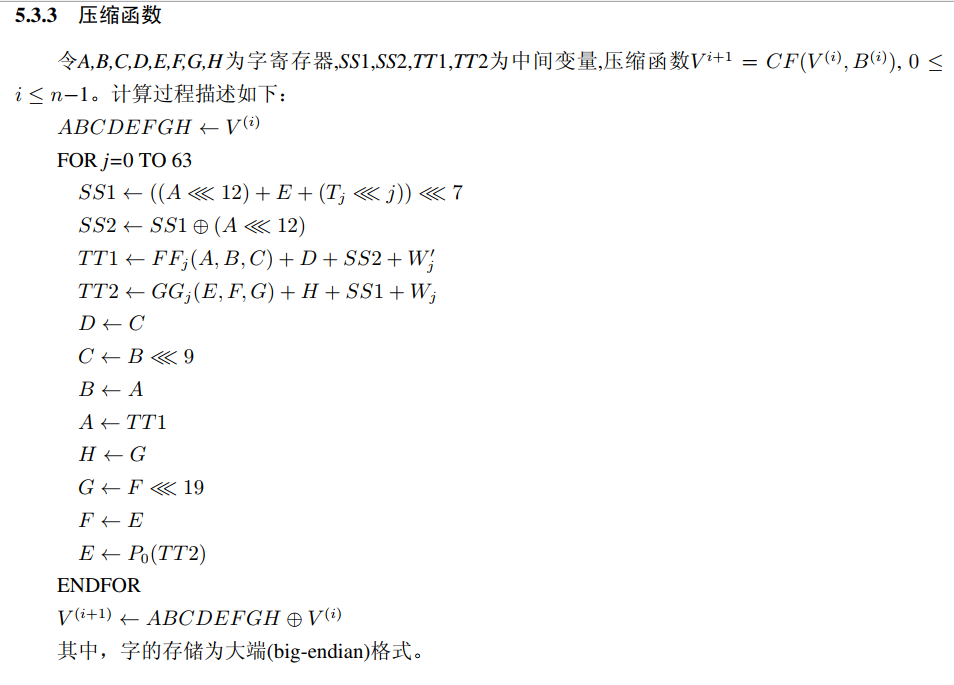
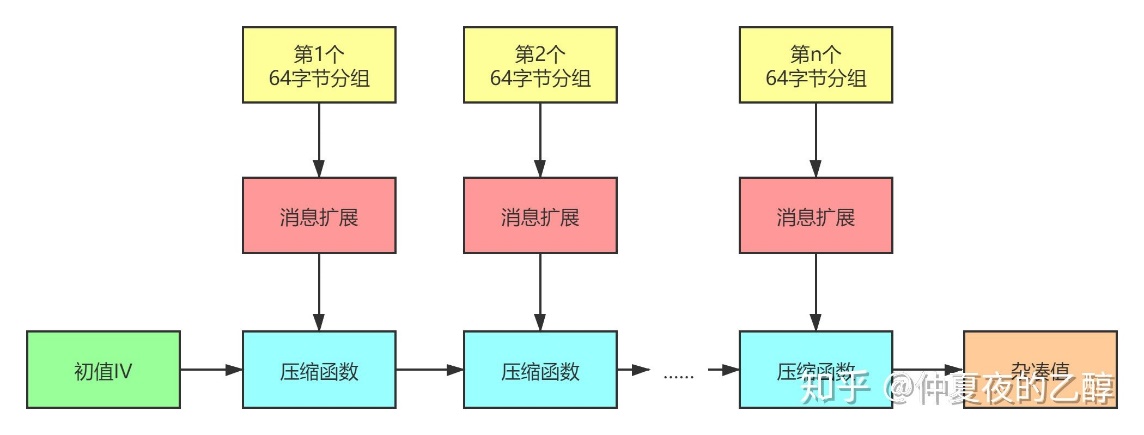
W.insert(W.end(), W\_.begin(), W\_.end());

return W;

}

**4.迭代压缩**

SM3的迭代过程和MD5类似，也是Merkle-Damgard结构。但和MD5不同的是，SM3使用消息扩展得到的消息字进行运算。这个迭代过程可以用这幅图表示：先填充一个“1”，后面加上k个“0”。其中k是满足(n+1+k) mod 512 = 448的最小正整数。



**代码实现：**

string CF(const string& V\_i, vector<u\_32> B\_i) {

u\_32 A = u\_32(stoul(V\_i.substr(0, 8), nullptr, 16));

u\_32 B = u\_32(stoul(V\_i.substr(8, 8), nullptr, 16));

u\_32 C = u\_32(stoul(V\_i.substr(16, 8), nullptr, 16));

u\_32 D = u\_32(stoul(V\_i.substr(24, 8), nullptr, 16));

u\_32 E = u\_32(stoul(V\_i.substr(32, 8), nullptr, 16));

u\_32 F = u\_32(stoul(V\_i.substr(40, 8), nullptr, 16));

u\_32 G = u\_32(stoul(V\_i.substr(48, 8), nullptr, 16));

u\_32 H = u\_32(stoul(V\_i.substr(56, 8), nullptr, 16));

u\_32 A\_ = u\_32(stoul(V\_i.substr(0, 8), nullptr, 16));

u\_32 B\_ = u\_32(stoul(V\_i.substr(8, 8), nullptr, 16));

u\_32 C\_ = u\_32(stoul(V\_i.substr(16, 8), nullptr, 16));

u\_32 D\_ = u\_32(stoul(V\_i.substr(24, 8), nullptr, 16));

u\_32 E\_ = u\_32(stoul(V\_i.substr(32, 8), nullptr, 16));

u\_32 F\_ = u\_32(stoul(V\_i.substr(40, 8), nullptr, 16));

u\_32 G\_ = u\_32(stoul(V\_i.substr(48, 8), nullptr, 16));

u\_32 H\_ = u\_32(stoul(V\_i.substr(56, 8), nullptr, 16));

u\_32 SS1, SS2, TT1, TT2;

for (int j = 0; j < 64; j++) {

SS1 = circleLS(circleLS(A, 12) + E + circleLS(T(j), j), 7);

SS2 = SS1 ^ circleLS(A, 12);

TT1 = FF(A, B, C, j) + D + SS2 + B\_i[68 + j];

TT2 = GG(E, F, G, j) + H + SS1 + B\_i[j];

D = C;

C = circleLS(B, 9);

B = A;

A = TT1;

H = G;

G = circleLS(F, 19);

F = E;

E = P0(TT2);

}

string a = dtoh(A ^ A\_);

string b = dtoh(B ^ B\_);

string c = dtoh(C ^ C\_);

string d = dtoh(D ^ D\_);

string e = dtoh(E ^ E\_);

string f = dtoh(F ^ F\_);

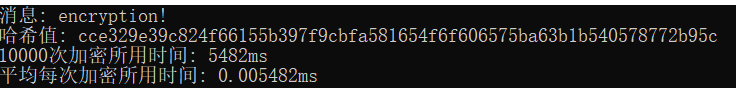
string g = dtoh(G ^ G\_);

string h = dtoh(H ^ H\_);

return a + b + c + d + e + f + g + h;

}

**运行结果：**

****

**\*Project9: AES / SM4 software implementation**

**基本步骤：**

要实现 AES-128 算法，需要以下几个过程：

（1）轮密钥加 （2）字节代换 （3）行移位 （4）列混合 （5）密钥扩展

**轮密钥加**

将 128 位轮密钥 Ki 同状态矩阵中的数据进行逐位异或操作（每个字节与轮密钥的对应字节进行异或运算）；轮密钥加的逆运算同正向的轮密钥加运算完全一致，这是因为异或的逆操作是其自身。轮密钥加非常简单，但却能够影响 S 数组中的每一位

**字节代换**

AES 的字节代换其实就是一个简单的查表操作。AES 定义了一个 S 盒和一个逆 S 盒。把每一个字节的高 4 位作为行值，低 4 位作为列值，取出 S 盒或者逆 S 盒中对应的行的元素作为输出。例如，加密时，输出的字节 S1 为 0x23, 则查 S 盒的第 0x02 行和 0x03 列，得到值替换 S1 原有的0x23。经过代换后得到新的状态矩阵。

**行移位**

行移位操作最为简单，它是用来将输入数据作为一个 4×4 的字节矩阵进行处理的，然后将这个矩阵的字节进行位置上的置换，目的是将单个位上的变换扩散到影响整个状态，从而达到雪崩效应。在加密时行移位处理与解密时的处理相反。它之所以称作行移位，是因为它只在 4×4 矩阵的行间进行操作，每行 4 字节的数据。在加密时，保持矩阵的第一行不变，第二行向左移动 8Bit(一个字节)、第三行向左移动 2 个字节、第四行向左移动 3 个字节。而在解密时恰恰相反，依然保持第一行不变，将第二行向右移动一个字节、第三行右移 2 个字节、第四行右移 3 个字节。

**代码实现：**

vector<string> RowMove(vector<string>& s)//行移位

{

vector<string> str = s;

for (int i = 0; i < 4; i++)

{

for (int j = 0; j < 4; j++)

{

str[j][2\*i] = s[(j + i) % 4][2\*i];

str[j][2\*i + 1] = s[(j + i) % 4][2\*i + 1];

}

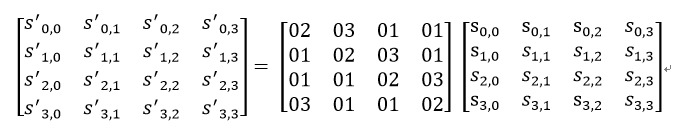
}

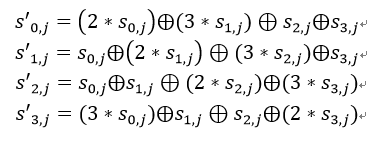
return str;

4.1.4 列混合

列混合变换是通过矩阵相乘来实现的，经行移位后的状态矩阵与固定的矩阵相乘，得到混淆后

的状态矩阵，如下图所示。



状态矩阵中的第 j 列 的列混合可以表示为下图所示：

**代码实现：**

vector<string> ColConfuse(vector<string>& s)//列混淆

{

vector<string> str = s;

for (int i = 0; i < 4; ++i)

{

auto temp = split\_s(s[i]);

int s0 = strInt(temp[0]);

int s1 = strInt(temp[1]);

int s2 = strInt(temp[2]);

int s3 = strInt(temp[3]);

string t0 = Char(LeftShift(s0) ^ LeftShift(s1) ^ s1 ^ s2 ^ s3);

if (t0.length() < 2)

t0 = "0" + t0;

string t1 = Char(s0 ^ LeftShift(s1) ^ LeftShift(s2) ^ s2 ^ s3);

if (t1.length() < 2)

t1 = "0" + t1;

string t2 = Char(s0 ^ s1 ^ LeftShift(s2) ^ s3 ^ LeftShift(s3));

if (t2.length() < 2)

t2 = "0" + t2;

string t3 = Char(s0 ^ LeftShift(s0) ^ s1 ^ s2 ^ LeftShift(s3));

if (t3.length() < 2)

t3 = "0" + t3;

str[i] = t0+t1+t2+t3;

}

return str;

}

密钥扩展

AES 首先将初始密钥输入到一个 4×4 的状态矩阵中，这个 4×4 矩阵的每一列的 4 个字节组成 一个字，矩阵 4 列的 4 个字依次命名为 W[0]、W[1]、W[2] 和 W[3]，它们构成一个以字为单位的数 组 W。 接着，对 W 数组扩充 40 个新列，以如下的递归方式产生：

(1) 如果 i 不是 4 的倍数，那么第 i 列：W[i] = W[i − 4] ⊕ W[i − 1]

(2) 如果 i 是 4 的倍数，那么第 i 列：W[i] = W[i − 4] ⊕ G(W[i − 1])

其中，函数 G 由 3 部分组成：字循环、字节代换和轮常量异或，这 3 部分的作用分别如下：

I. 字循环：将 1 个字中的 4 个字节循环左移 1 个字节。即将输入字 [b0,b1,b2,b3] 变换成

[b1,b2,b3,b0]。

II. 字节代换：对字循环的结果使用 S 盒进行字节代换。

III. 轮常量异或：将前两步的结果同轮常量 Rcon[j] 进行异或，其中 j 表示轮数。

**代码实现：**

vector<string> KeyExtend(string& key0)//计算十轮拓展密钥

{

vector<string> key = MakeGroup(key0);

for (int i = 4; i < 44; i++)

{

string k = "";

string temp = key[i - 1];

if (i % 4 == 0)

temp = G(temp, i / 4 - 1);

k = strXor(temp, key[i - 4]);

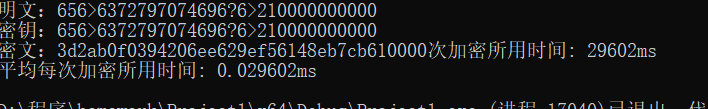
key.push\_back(k);

}

return key;

}

**运行结果：**

****

**\*Project10: report on the application of this deduce technique in Ethereum with ECDSA**

**\*Project11: impl sm2 with RFC6979**

**\*Project12: verify the above pitfalls with proof-of-concept code**

椭圆曲线加密算法，简称ECC，是基于椭圆曲线数学理论实现的一种非对称加密算法。相比RSA，ECC优势是可以使用更短的密钥，来实现与RSA相当或更高的安全，RSA加密算法也是一种非对称加密算法，在公开密钥加密和电子商业中RSA被广泛使用。

**处理过程：**

1.参与数字签名的所有通信方都使用相同的全局参数，用于定义椭圆曲线以及曲线上的基点。

2.签名者首先生成一对公私钥。对于私钥，选择一个随机数或者伪随机数作为私钥，利用随机数和基点算出另一点，作为公钥。

3.对消息计算Hash值，用私钥、全局参数和Hash值生成签名，包括两个整数r和s。

4.验证者用签名者的公钥、全局参数等验证。

**密钥生成：**

每个签名者都要生成一对公私钥，假设签名者是Bob，通过以下步骤产生上述两个密钥：

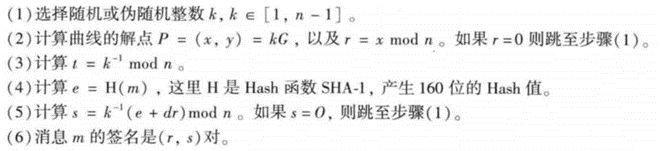
1.选择随机整数d，。

2.计算Q=dG。得到一个曲线上的解点。

3.Bob的公钥是Q，私钥是d。

**数字签名的产生与认证：**

**产生：**



**代码实现：**

import sys

import struct

import random

from math import gcd

from math import ceil

from math import floor

from gmssl import sm3

q = "8542D69E4C044F18E8B92435BF6FF7DE457283915C45517D722EDB8B08F1DFC3"

a = "787968B4FA32C3FD2417842E73BBFEFF2F3C848B6831D7E0EC65228B3937E498"

b = "63E4C6D3B23B0C849CF84241484BFE48F61D59A5B16BA06E6E12D1DA27C5249A"

x\_G = "421DEBD61B62EAB6746434EBC3CC315E32220B3BADD50BDC4C4E6C147FEDD43D"

y\_G = "0680512BCBB42C07D47349D2153B70C4E5D7FDFCBFA36EA1A85841B9E46E09A2"

G = (x\_G, y\_G)

n = "8542D69E4C044F18E8B92435BF6FF7DD297720630485628D5AE74EE7C32E79B7"

def int\_hex(str):#字符串转16进制

return int(str, 16)

def sm3\_hash(message):#求字符串的hash值

message = message.encode('utf-8')

msg\_list = [i for i in message]

hash\_hex = sm3.sm3\_hash(msg\_list)

return hash\_hex

def inv(a, m):#求逆元

if gcd(a, m) != 1:

return None

u1, u2, u3 = 1, 0, a

v1, v2, v3 = 0, 1, m

while v3 != 0:

q = u3 // v3

v1, v2, v3, u1, u2, u3 = (u1 - q \* v1), (u2 - q \* v2), (u3 - q \* v3), v1, v2, v3

return u1 % m

def add\_ECC(P, Q):#椭圆曲线加法

if P == 0:

return Q

if Q == 0:

return P

x1, y1, x2, y2 = int\_hex(P[0]), int\_hex(P[1]), int\_hex(Q[0]), int\_hex(Q[1])

q\_int = int\_hex(q)

tmp1, tmp2 = y2 - y1, inv(x2 - x1 % q\_int, q\_int)

l = tmp1 \* tmp2 % q\_int

x = (l \* l - x1 - x2) % q\_int

y = (l \* (x1 - x) - y1) % q\_int

res = (hex(x)[2:], hex(y)[2:])

return res

def double\_ECC(P):#二倍点

if P == 0:

return P

x1, y1 = int\_hex(P[0]), int\_hex(P[1])

a\_int, q\_int = int\_hex(a), int\_hex(q)

tmp1 = 3 \* x1 \* x1 + a\_int

tmp2 = inv(2 \* y1, q\_int)

l = (tmp1 \* tmp2) % q\_int

x = (l \* l - 2 \* x1) % q\_int

y = (l \* (x1 - x) - y1) % q\_int

Q = (hex(x)[2:], hex(y)[2:])

return Q

def mul\_ECC(P, k):#点乘

k\_bin = bin(k)[2:]

i = len(k\_bin) - 1

Q = P

if i > 0:

k = k - 2\*\*i

while i > 0:

Q = double\_ECC(Q)

i -= 1

if (k > 0):

Q = add\_ECC(Q, mul\_ECC(P, k))

return Q

def check\_ECC(P):#检查是否在椭圆曲线上

x, y = int\_hex(P[0]), int\_hex(P[1])

q\_int, a\_int, b\_int = int\_hex(q), int\_hex(a), int\_hex(b)

if (y \* y) % q\_int == (x \* x \* x + a\_int \* x + b\_int) % q\_int:

return True

else:

return False

def KDF(Z, klen):#密钥扩展函数

ct = int\_hex("00000001")

ceil\_val = ceil(klen / v)

floor\_val = floor(klen / v)

hash = b""

for i in range(1, ceil\_val):

ct\_hex = hex(ct)[2:]

hash += sm3\_hash(Z + (ct\_hex).zfill(8))

ct += 1

ct\_hex = hex(ct)[2:]

if klen % v == 0:

hash\_x = sm3\_hash(Z + (ct\_hex).zfill(8))

else:

hash\_x = sm3\_hash(Z + (ct\_hex).zfill(8))

hash\_x = hash\_x[:((klen - v \* floor\_val) // 4)]

K = hash.hex() + hash\_x

return K

def sm2\_enc(M, P\_B):#加密函数

klen = len(M) \* 4

k = hex(random.randint(1,int\_hex(n)-1))[2:]

k\_int = int\_hex(k)

C1 = mul\_ECC(G, k\_int)

x2, y2 = mul\_ECC(P\_B, k\_int)

t = KDF(x2 + y2, klen)

C2 = hex(int\_hex(M) ^ int\_hex(t))[2:]

C3 = sm3\_hash(x2 + M + y2)

C = C1[0] + C1[1] + C2 + C3

len\_x = len(C1[0])

len\_y = len(C1[1])

len\_C2 = len(C2)

return C, len\_x, len\_y, len\_C2

def sm2\_dec(CT, d\_B, len\_x, len\_y, len\_C2, klen):#解密函数

x1 = CT[:len\_x]

y1 = CT[len\_x:len\_x + len\_y]

C1 = (x1, y1)

if check\_ECC(C1) == False:

print("False!")

return ""

x2, y2 = mul\_ECC(C1, int\_hex(d\_B))

t = KDF(x2 + y2, klen)

C2 = CT[len\_x + len\_y:len\_x + len\_y + len\_C2]

M = hex(int\_hex(C2) ^ int\_hex(t))[2:]

u = sm3\_hash(x2 + M + y2)

C3 = CT[len\_x + len\_y + len\_C2:]

if u != C3:

print("False!")

return ""

return M

#参数设定

#私钥d\_B

d\_B = hex(random.randint(pow(2,127),pow(2,128)))[2:]

P\_B = mul\_ECC(G, int\_hex(d\_B))

v = 256

PT = hex(random.randint(pow(2,127),pow(2,128)))[2:]

print("PT: " + PT + '\n')

klen = len(PT) \* 4

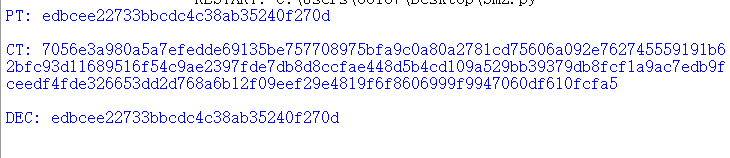
CT, len\_x, len\_y, len\_C2 = sm2\_enc(PT, P\_B)

print("CT: " + CT + '\n')

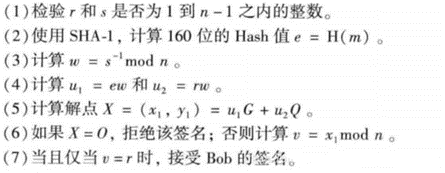
M = sm2\_dec(CT, d\_B, len\_x, len\_y, len\_C2, klen)

print("DEC: " + M + '\n')

**运行结果：**

****

**验证：**



**代码实现：**

def SM2\_sign(M):

tmp = str(len(ID\_A)) + ID\_A + a + b + x\_G + y\_G + P\_A[0] + P\_A[1]

Z\_A = sm3\_hash(tmp)

M\_ = Z\_A + M

e = sm3\_hash(M\_)

#k = random.randint(1, int\_hex(n) - 1)

kG = mul\_ECC(G, k)

r = (int\_hex(e) + int\_hex(kG[0])) % int\_hex(n)

s = (inv(1 + int\_hex(d\_A), int\_hex(n)) \*

(k - r \* int\_hex(d\_A))) % int\_hex(n)

return r, s

def ECDSA\_sign(M):

kG = mul\_ECC(G, k)

e = sm3\_hash(M)

r = int\_hex(kG[0]) % int\_hex(n)

s = (int\_hex(e) + r \* int\_hex(d\_A)) \* inv(k, int\_hex(n)) % int\_hex(n)

return r, s

def verify\_1(k, r, s):

t1 = inv(s + r, int\_hex(n))

t2 = k - s

return (t1 \* t2) % int\_hex(n)

def verify\_2(k, r\_1, r\_2, s\_1, s\_2):

t1 = s\_2 - s\_1

t2 = s\_1 - s\_2 + r\_1 - r\_2

return (t1 \* inv(t2, int\_hex(n))) % int\_hex(n)

def verify\_3(k, r\_2, s\_2):

t1 = inv(s\_2 + r\_2, int\_hex(n))

t2 = k - s\_2

return (t1 \* t2) % int\_hex(n)

def verify\_4(k, r\_1, r\_2, s\_1, s\_2):

e\_1 = (k \* s\_1 - int\_hex(d\_A) \* r\_1) % int\_hex(n)

t1 = s\_1 \* s\_2 - e\_1

t2 = (r\_1 - s\_1 \* s\_2 - s\_1 \* r\_2) % int\_hex(n)

return (t1 \* inv(t2, int\_hex(n))) % int\_hex(n)

k = 123456

print(f'd = {int\_hex(d\_A)}')

r, s = SM2\_sign(M)

t1 = verify\_1(k, r, s)

print(f'If k is leaked, we can get d = {t1}')

r\_1, s\_1 = SM2\_sign(M\_1)

r\_2, s\_2 = SM2\_sign(M\_2)

t2 = verify\_2(k, r\_1, r\_2, s\_1, s\_2)

print(f'If k is reused, we can get d = {t2}')

t3 = verify\_3(k, r\_2, s\_2)

print(f'If k is reused by two users, we can get another d = {t3}')

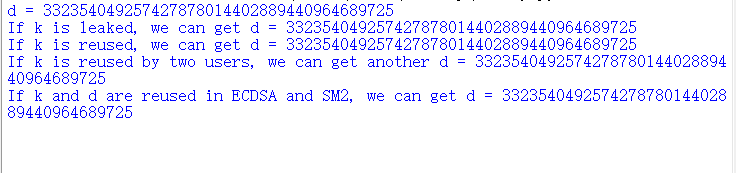
r\_1, s\_1 = ECDSA\_sign(M\_1)

r\_2, s\_2 = SM2\_sign(M\_2)

t4 = verify\_4(k, r\_1, r\_2, s\_1, s\_2)

print(f'If k and d are reused in ECDSA and SM2, we can get d = {t4}')

**运行结果：**

****

**以太坊中的ECDSA**

Secp256k1是指比特币中使用的ECDSA(椭圆曲线数字签名算法)曲线的参数，并且在高效密码学标准（Certicom Research，http://www.secg.org/sec2-v2.pdf）中进行了定义。以太坊也使用了 Secp256k1。

对以太坊一笔交易进行签名的大致步骤如下（简化后）：

1.对交易数据进行 RLP 编码

2.对第一步得到的编码进行哈希

3.将哈希与标识以太坊的特定字符串拼接在一起，再次哈希。这一步是为了保证该签名仅在以太坊上可用

4.用ECDSA算法对第三步得到的哈希进行签名，得到 (r, s, v)，v 是一个字节

5.将第四步得到的签名与交易数据拼接，再次进行RLP编码，得到最终的签名消息。

**\*Project14: Implement a PGP scheme with SM2**

**PGP的介绍：**

PGP（Pretty Good Privacy）加密，由一系列散列、数据压缩、对称密钥加密，以及公钥加密的算法组合而成，每个步骤支持几种算法。

PGP支持消息认证和完整性检测：

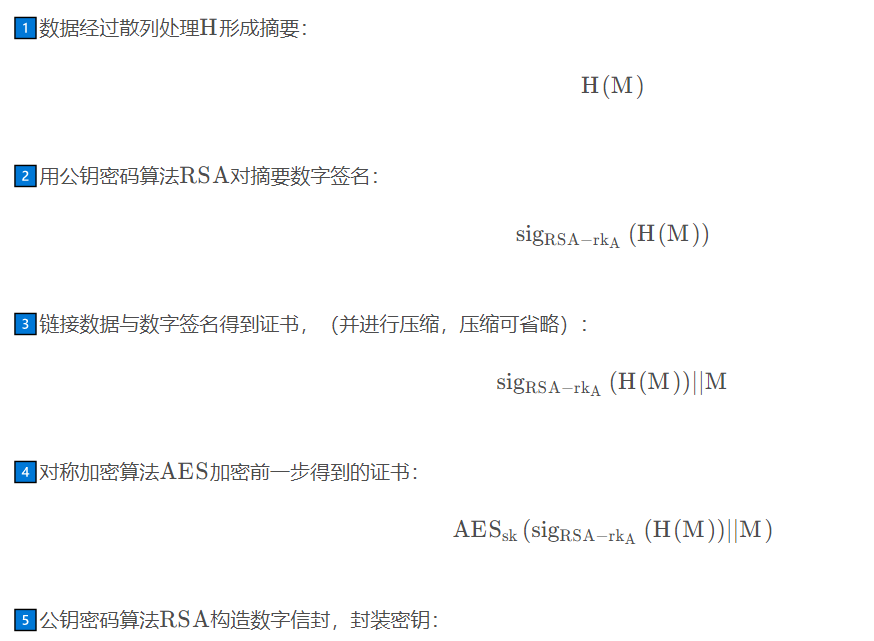
①完整性检测被用来检查消息在传输过程中是否变更过（即验证消息完整性）；

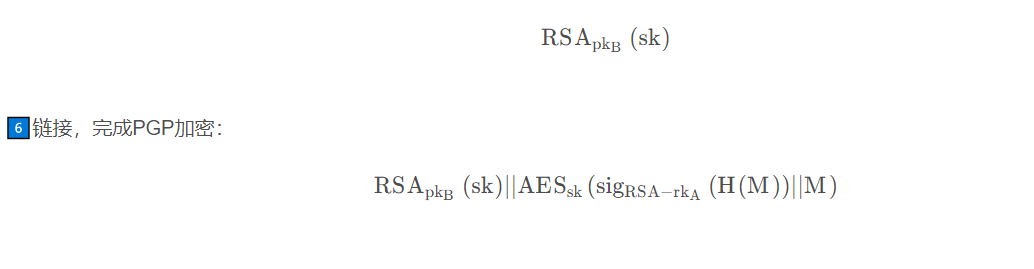
②消息认证则是被用来决定消息是否确由某特定的人或实体发出（即数字签名验证）。

在PGP中，这些特性默认是和消息加密同时开启的，而且同样可以被应用到明文的验证。

发送者只需使用PGP为消息创建一个数字签名，即以数据或信息创建一个散列，然后使用发送者的私钥利用散列生成数字签名。

**基本操作：**





**根据以上步骤，本题中我采用sm2来代替RSA对会话密钥进行加密，构建了一个利用H、sm2、AES的PGP加密。**

**加密形式为sm2(sk)||AES(SHA256(M)||M)**

**以下为各部分基本代码：**

首先用hash函数对消息进行压缩，并连接H(M)||M：

hash\_data = hashlib.sha256(data).digest()+data #H(M)||M

然后用AES加密H(M)||M，会话密钥作为加密密钥：

def AES\_enc(hash\_data,session\_key):

iv = get\_random\_bytes(16)#初始值

cipher = AES.new(session\_key, AES.MODE\_CBC, iv) #确定AES加密的模式和iv，用会话密钥作为密钥

return cipher.encrypt(pad(hash\_data, AES.block\_size)) #填充为128bit的倍数

然后用sm2加密会话密钥：

def sm2\_enc(session\_key):#sm2加密会话密钥

sm2\_crypt = sm2.CryptSM2(public\_key=PK, private\_key=RK)

return sm2\_crypt.encrypt(session\_key)

最后将加密后的会话密钥和文本数据进行链接，得到加密结果：

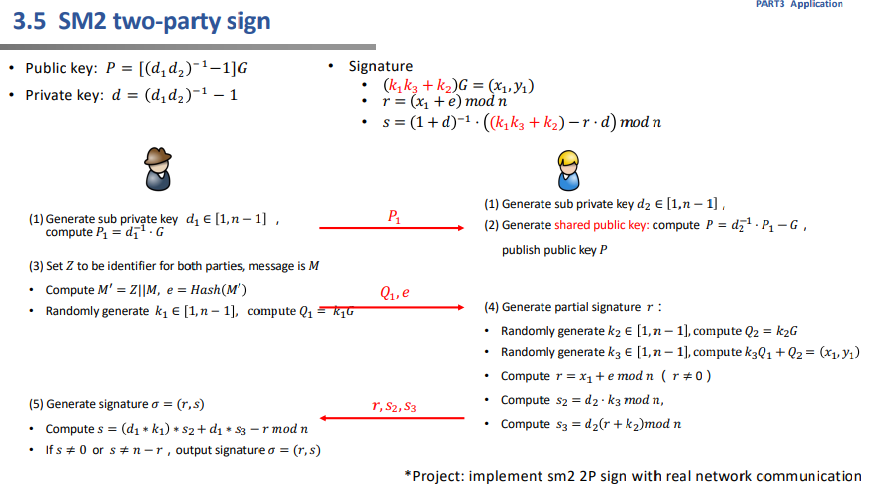
#拼合并转为文本数据

ciphertext=(ciphertext\_sm2+ciphertext\_aes).hex().encode('utf-8')#sm2(sk)||AES(H(M)||M)

**\*Project15: implement sm2 2P sign with real network communication**

**主要流程：**

如下图所示



**Alice**

1.首先与bob建立连接；

alice = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

alice\_addr = ('localhost', 50007)

alice.bind(alice\_addr)

alice.listen(1)

print('connecting...')

bob, addr = alice.accept()

print('Bob has connected!')

2.首先产生随机数d1，计算p1，并将p1发送给Bob；

d1=random.randint(1,n-1)

#p1=d1^(-1)·G

p1=mul\_ECC(X,Y,invert(d1,p))

x,y=hex(p1[0]),hex(p1[1])

# p1

bob.sendto(x.encode(),addr)

bob.sendto(y.encode(),addr)

3.然后计算M'=Z||M和e=Hash(M')，随机生成k1，并计算q1，再将(q1, e)发送给Bob；

m="message"

z="\_bob"

e=sm3\_hash(m+z)

k1=random.randint(1,n-1)

# q1=k1·G

q1=mul\_ECC(X,Y,k1)

x,y=hex(q1[0]),hex(q1[1])

# q1,e

bob.sendto(x.encode(),addr)

bob.sendto(y.encode(),addr)

bob.sendto(e.encode(),addr)

4.最后接收(r, s\_2, s\_3)，用这些数据计算s，并得到签名(r, s)。

# 接受 r,s2,s3

r,addr=bob.recvfrom(1024)

r=int(r.decode(),16)

s2,addr=bob.recvfrom(1024)

s2=int(s2.decode(),16)

s3,addr=bob.recvfrom(1024)

s3=int(s3.decode(),16)

print("r:",r)

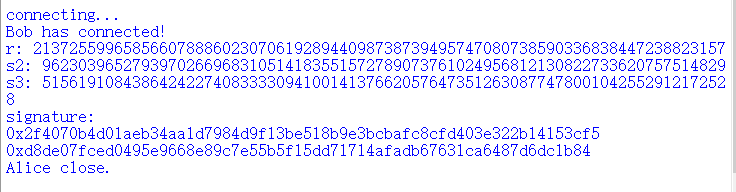
print("s2:",s2)

print("s3:",s3)

# (生成签名 s=(d1\*k1)\*s2+d1\*s3-r mod n

s=((d1\*k1)\*s2+d1\*s3-r)%n

**5.运行结果**

****

**Bob**

1.首先与Alice建立连接；

alice = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

addr = ('localhost', 50007)

print('Connecting to Alice...')

try:

alice.connect(addr)

print('successfully!')

except Exception:

print('Failed!')

sys.exit()

2.生成随机数d2，并根据接收的p1计算公钥P；

d2=random.randint(1,n-1)

# 接受 p1

x,addr=alice.recvfrom(1024)

x=int(x.decode(),16)

y,addr=alice.recvfrom(1024)

y=int(y.decode(),16)

#p = d2^(-1)·p1-G

p1=(x,y)

ptem=mul\_ECC(p1[0],p1[1],invert(d2,p))

ptem=add\_ECC(ptem[0],ptem[1],X,-Y)

3.根据接受的(q1, e)计算(r, s2, s3)，并发给Alice。

# 接受 q1,e

x,addr=alice.recvfrom(1024)

x=int(x.decode(),16)

y,addr=alice.recvfrom(1024)

y=int(y.decode(),16)

q1=(x,y)

e,addr=alice.recvfrom(1024)

e=int(e.decode(),16)

k2=random.randint(1,n-1)

k3=random.randint(1,n-1)

#q2=k2·G

q2=mul\_ECC(X,Y,k2)

#k3·q1+q2=(x1,y1)

x11,y11=mul\_ECC(q1[0],q1[1],k3)

x11,y11=add\_ECC(x11,y11,q2[0],q2[1])

# r=x1+e mod n

r=(x11+e)%n

# s2=d2\*k3 mod n

s2=(d2\*k3)%n

# s3=d2\*(r+k2) mod n

s3=(d2\*(r+k2))%n

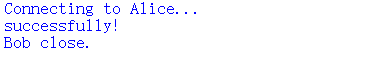
# r,s2,s3

alice.sendall(hex(r).encode())

alice.sendall(hex(s2).encode())

alice.sendall(hex(s3).encode())

**4.运行结果**

****

**\*Project17：比较Firefox和谷歌的记住密码插件的实现区别**

Firefox和谷歌的记住密码插件在实现上有一些区别。

它们的共同点：都是基于浏览器的自动填充功能，用于记住用户在网页上输入的用户名和密码，并在下次访问相同网站时自动填充。

然而，它们的具体实现方式有所不同。在Firefox中，记住密码的功能是由Firefox内置的密码管理器来实现的。当用户在网页上输入用户名和密码时，Firefox会弹出一个提示框询问是否要记住密码。如果用户选择记住密码，Firefox会将用户名和密码保存到浏览器的密码数据库中。当用户下次访问相同网站时，Firefox会自动填充保存的用户名和密码。Firefox保存密码文件中存储了密码的加密形式，以及一些其余的与密码相关的信息。

而在谷歌浏览器中，记住密码的功能是通过谷歌账号来实现的。当用户在网页上输入用户名和密码时，谷歌浏览器会自动检测并弹出一个提示框询问是否要保存密码到谷歌账号。chrome保存密码文件本身就是加密过的，使用文本编辑器打开无法辨认具体内容，只能看到一些不完整的url信息等等。

Firefox和chrome保存密码的方式有很大不同，Firefox的密码存储文件可以直接打开并查看内容，而chrome则不能。Firefox将加密后的密码保存到一个json格式的文件中，其他信息仍为明文；而chrome则将保存密码的文件进行加密，无法得到有效信息。

**\*Project22: research report on MPT**

**这里只写了大概，详见project22文件里**

**1.概述**

Merkle Patricia Tree（又称为Merkle Patricia Trie）是一种经过改良的、融合了默克尔树和前缀树两种树结构优点的数据结构，是以太坊中用来组织管理账户数据、生成交易集合哈希的重要数据结构。

MPT树有以下几个作用：

1. 存储任意长度的key-value键值对数据；
2. 提供了一种快速计算所维护数据集哈希标识的机制；
3. 提供了快速状态回滚的机制；
4. 提供了一种称为默克尔证明的证明方法，进行轻节点的扩展，实现简单支付验证；

**2.** **结构设计**

**节点分类**

MPT树中，树节点可以分为以下四类：

空节点、分支节点、叶子节点、扩展节点。

**key值编码**

在以太坊中，MPT树的key值共有三种不同的编码方式，以满足不同场景的不同需求。

三种编码方式分别为：

1. Raw编码（原生的字符）；
2. Hex编码（扩展的16进制编码）；
3. Hex-Prefix编码（16进制前缀编码）；

**3. 基本操作**

**Get、Insert、Delete、Update、Commit**