**汇总报告**

Project1：

实验原理

生日攻击的目的是寻求一个基于sm3哈希值的弱碰撞。

这里我使用了类似查表攻击似的数据结构，一边存表一边查表 以便可以在较短时间内找到一个前16bit的hash弱碰撞。

sm3的实现\*\*

代码前半部分是sm3的实现，其中seg\_func（）函数是主体，体现了sm3的逻辑，包括了消息填充，消息扩展，压缩函数等函数的调用，最后只需要调用seg\_func（）就能得出hash值。

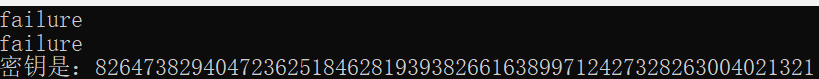
朴素生日攻击,即在主函数里随机选取两个信息，进行hash值的比对，是没有任何方法的随机选择的消息。通过Index变量控制要比对的碰撞的位数，是4\*Index位。

通过时间测试函数，最后输出找到碰撞的时间

如果寻找更长比特的碰撞，寻找时间也会相应变长。

硬件环境：Intel [Pentium 4](https://zhidao.baidu.com/search?word=Pentium%204&fr=iknow_pc_qb_highlight)  CPU:2.0GHz ，2G内存,500G硬盘  
软件环境：开发环境：linux

实验结果



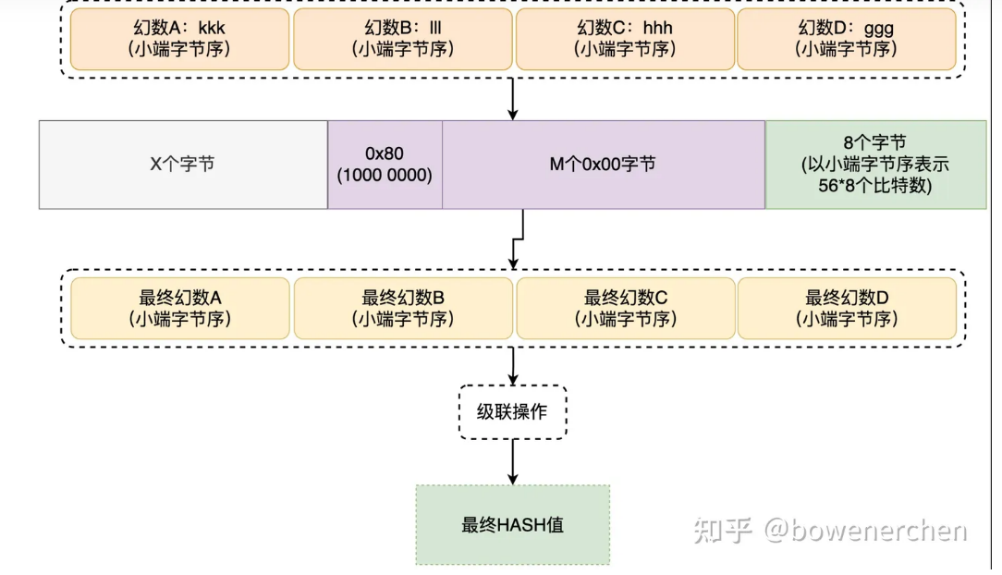
Project2

具体而言，rho方法首先在原始消息的末尾添加一个1比特，然后添加若干个0比特，以使消息的总长度满足以下两个条件之一：  
  
消息的长度模512余数为448。  
消息的长度模512余数为0，但是原始消息的末尾64位用于存储原始消息的比特长度。  
接下来，rho方法将填充后的消息分为若干个512比特（64字节）的块。如果消息长度超过了512比特，则将其分成多个块。  
  
最后，对每个块应用SM3算法的压缩函数进行处理，以生成哈希值。

硬件环境： PU:2.0GHz ，5G内存,500G硬盘  
软件环境：开发环境：windows

Project3

实验原理



首先由前半部分信息data1经过hash得到一个杂凑值。

\* 可以用该杂凑值替换hash函数的初始IV，把任意的后半部分信息data2进行相同的hash计算。

\* 由上述方法得到的hash值，与直接使用(data1+data\_padding+data2)该信息进行hash得到的结果相同。

\* 从上述原理我们可以看出，第一种计算并没有知道原始消息内容，已获知信息是原消息的杂凑值，由此便可构造出新消息的杂凑值

硬件环境： PU:2.0GHz ，5G内存,500G硬盘  
软件环境：开发环境：windows

Project4

实验原理

使用位运算代替乘法和除法，利用并行计算，合理地对数据进行对齐，使用表格预计算，以减少运行时的计算量。使用SIMD指令的部分是sm3中的消息扩展部分。在消息扩展部分同样也使用循环展开，进一步提速

实验环境

硬件环境： PU:2.0GHz ，5G内存,500G硬盘  
软件环境：开发环境：windows

实验结果



Project5

实验原理

默克尔树（Merkle tree）是一种数据结构，以它的提出者默克尔命名，根据默克尔树的性质也可以叫哈希树，是一种典型的二叉树。

默克尔树由根，分支（中间的非叶节点），叶节点组成。

\*\*叶子节点\*\*：Merkle树的叶子节点是数据的哈希值。每个叶子节点都对应一条数据。

2. \*\*内部节点\*\*：除叶子节点外的其他节点都是内部节点。内部节点的值通过将其两个子节点的哈希值连接并进行哈希运算来计算得到。

3. \*\*根节点\*\*：Merkle树的顶部节点称为根节点。它是所有其他节点的父节点，代表整个树的哈希值。

它有两个重要的特点：

第一个是叶节点包含的是相关数据的哈希值（而不是相关数据本身），这也是它叫哈希树的原因，因为它的所有节点都是存储的哈希值

第二个是非叶节点的内容是孩子的哈希值的哈希。

实验环境

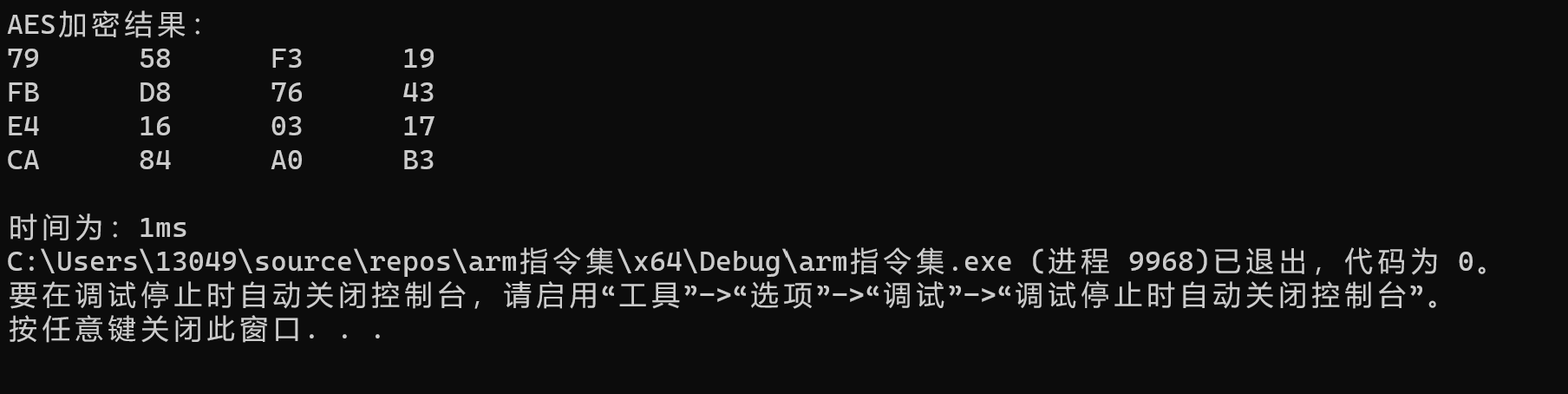
硬件环境： PU:2.0GHz ，5G内存,500G硬盘  
软件环境：开发环境：windows

Project9

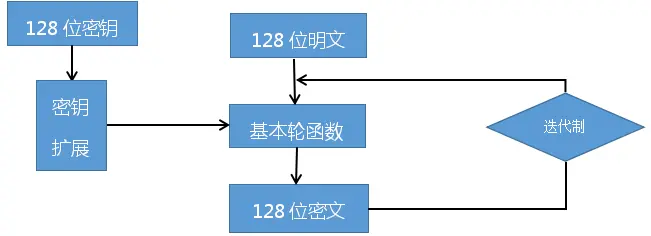
原理：AES加密过程涉及到 4 种操作，分别是**字节替代、行移位、列混淆和轮密钥加**。解密过程分别为对应的逆操作。由于每一步操作都是可逆的，按照相反的顺序进行解密即可恢复明文。加解密中每轮的密钥分别由初始密钥扩展得到。算法中 16 个字节的明文、密文和轮密钥都以一个 4x4 的矩阵表示。



# 实现结果



# SM4算法

SM4算法的基本步骤如下：

密钥扩展：根据输入的密钥生成轮密钥，用于后续的加密和解密操作。

轮函数：SM4算法使用32位的字作为数据块，轮函数包括四个运算：非线性变换、线性变换、循环左移和异或运算。

a. 非线性变换：通过S盒代替输入的字节，S盒是固定的非线性变换表。

b. 线性变换：通过固定的置换操作，改变字的位置。

c. 循环左移：将字中的位进行循环左移操作。

d. 异或运算：将变换后的字与轮密钥进行按位异或运算。

加密/解密轮处理：SM4算法采用多轮迭代处理来加密和解密数据。每一轮的操作包括以下四个步骤：

a. 轮函数运算：使用轮函数对输入数据块进行变换操作。

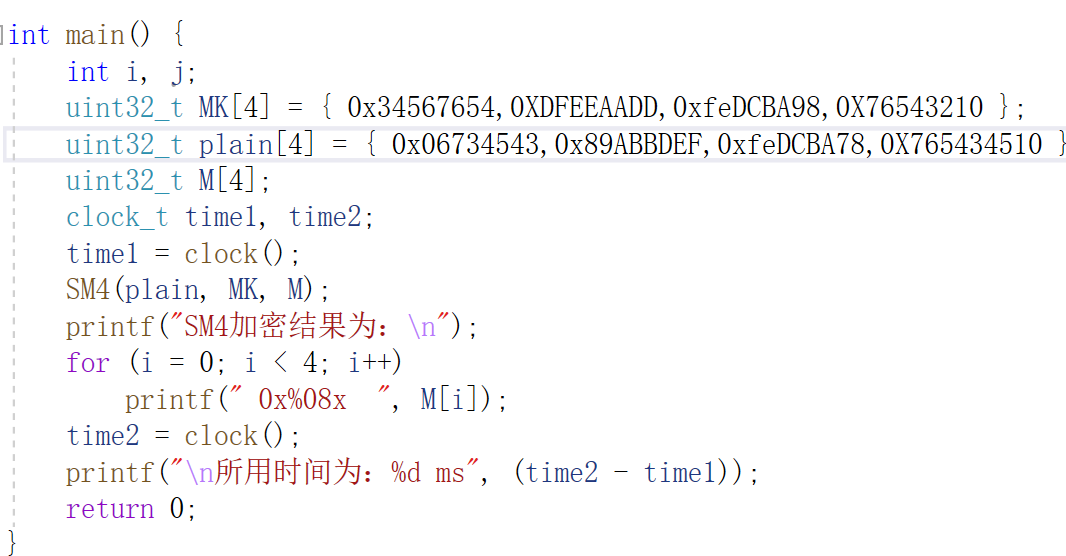
b. 轮密钥加：将当前轮的轮密钥与数据块进行按位异或运算。

c. 列混淆：对输入数据块的列进行线性变换。

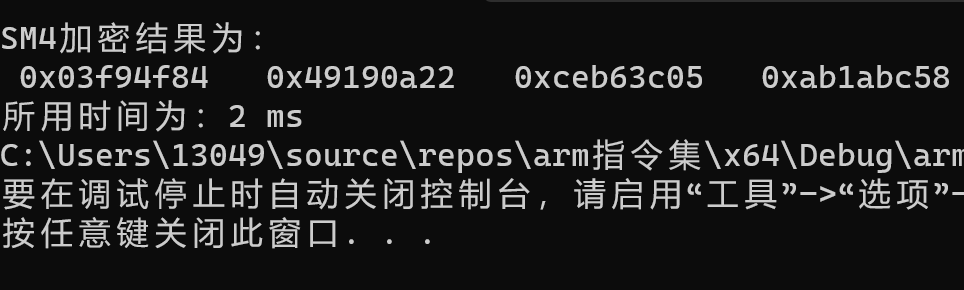
d. 轮密钥更新：生成下一轮使用的轮密钥。

最后一轮处理：最后一轮处理与其他轮不同，它不包括列混淆。

密文输出（加密）/明文输出（解密）：经过多轮迭代后，最后一轮的输出即为加密后的密文或解密后的明文



**实现结果**



Project10

1、密钥生成

传统的ECDSA密钥生成过程包括选择一个随机数k，并计算公钥Q=k\*G，其中G是基点。然而，选择合适的随机数k是一个困难的问题，因为k的选择会影响到私钥的安全性。推导技术通过优化随机数的选择过程，可以减少私钥被猜测的风险，从而提高密钥生成的效率和安全性。

2、签名验证

传统的ECDSA签名验证过程包括计算一个点R=r\*G，并将其x坐标与签名中的r进行比较。然而，这个比较操作需要进行一次椭圆曲线点的加法运算，导致了一定的计算开销。推导技术通过优化签名验证过程中的计算步骤，可以减少计算开销，提高签名验证的效率。



Project11

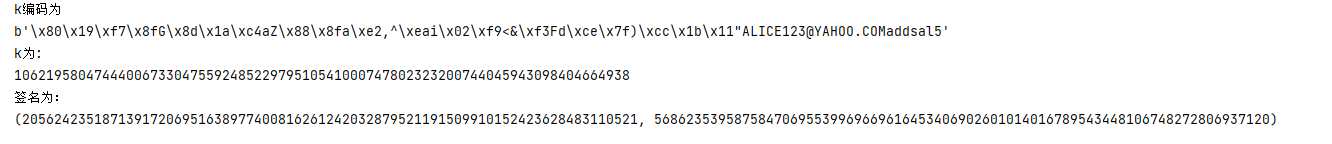
Sm2的PGP方案

按照规定，k随机。故对其进行补字和哈希。

所以我们规定k=（d||ID||"addsalt"）其中d为私钥，ID为ID，再接一个混淆的字符串，用sm3算法进行hash处理，最后得到一个随机的数。这就导致k相同的概率很小很小，不会由于k相同而产生安全问题

结合sm2算法完成该实验

最后运行程序可得结果：



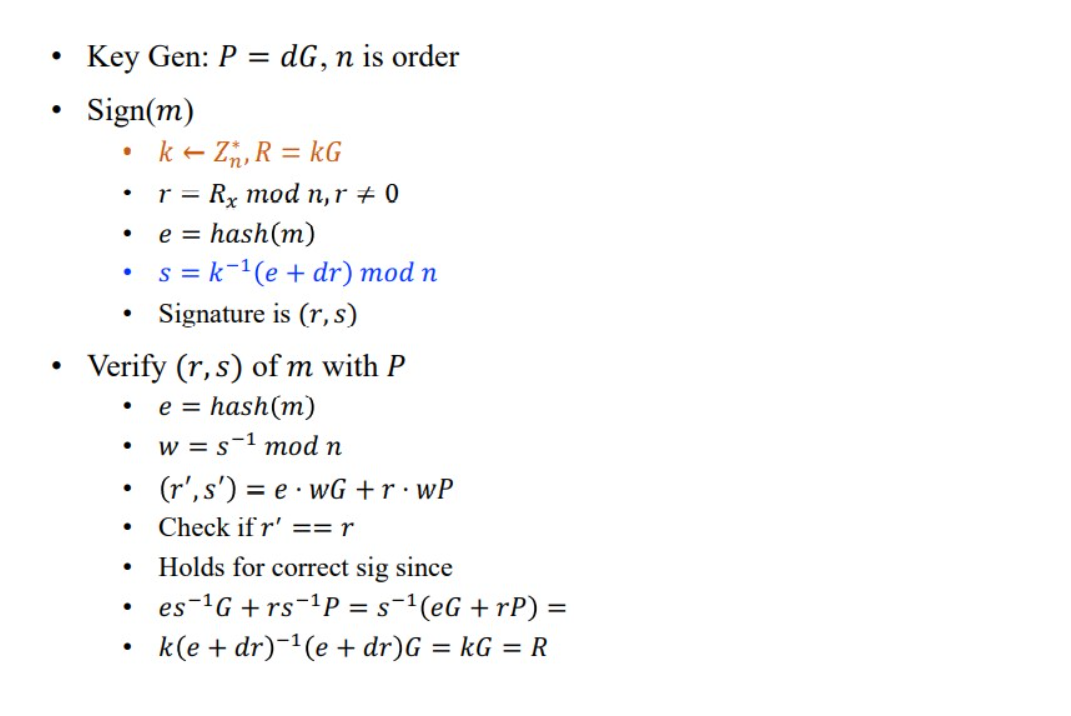
最后可得耗时：



Project12

一：实验原理

课件中资料如下：



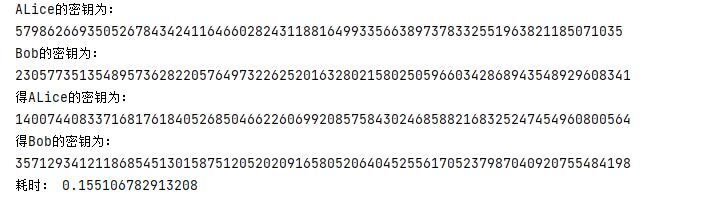
按照这种算法可以还原出私钥，前提是选取了一致的k

通过实现椭圆曲线算法，可以很实现算法中的各种运算是比较容易的 。

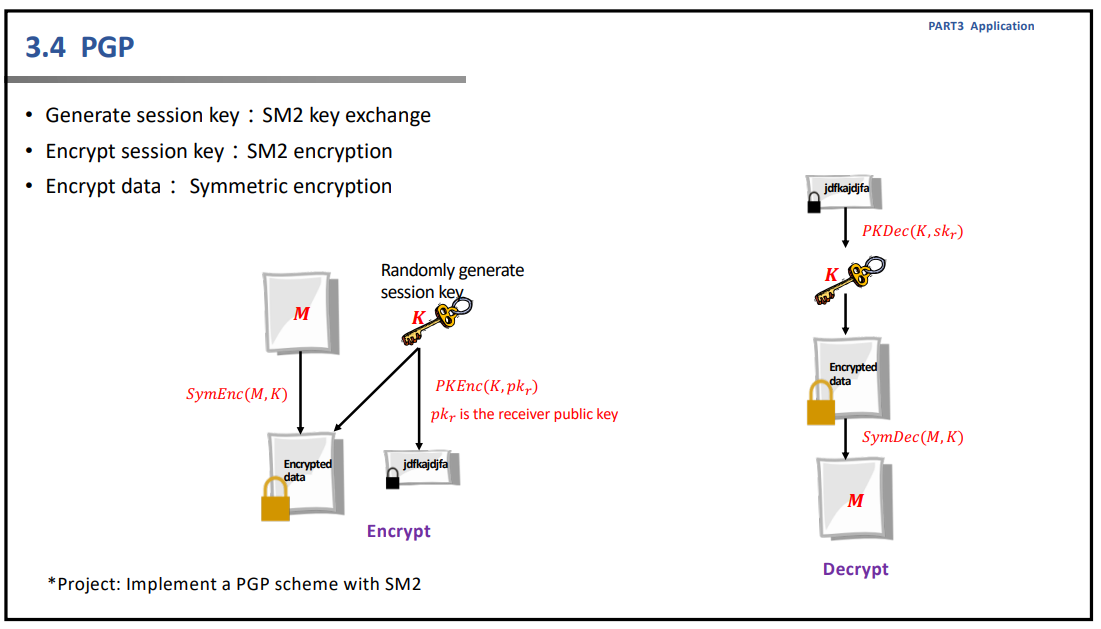
二：实验结果：

可以发现还原成功了。

同时测量了时间



Project14



按照此结构，实现PGP方案

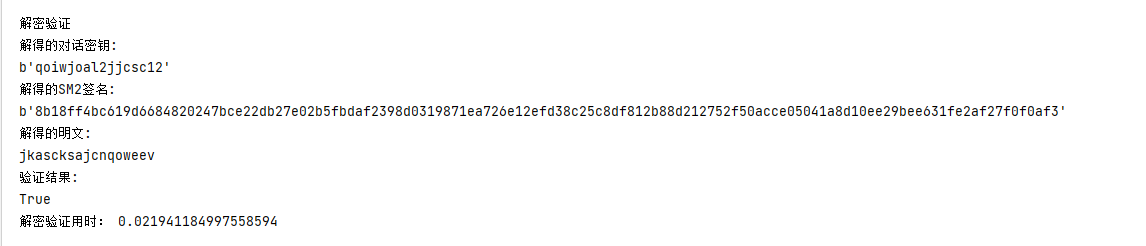
于是在代码中可以体现出:

在加密的过程中，我们生成一个随机数 作为密钥k，对data进行hash等操作后进行sm4对称加密。然后用sm2对k进行加密后一起拼接发送。

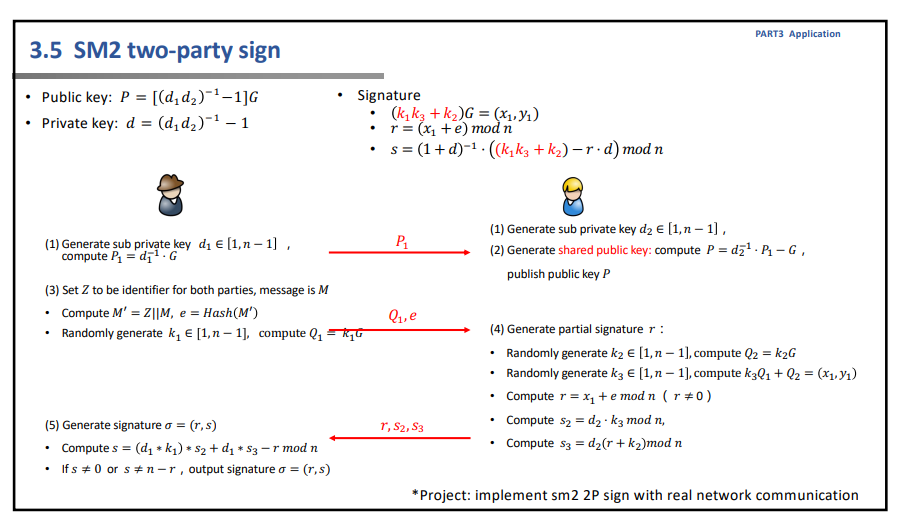
在解密函数decrypt中，我们只需要按照顺序分开解密，验证公钥即可

最后结果为：





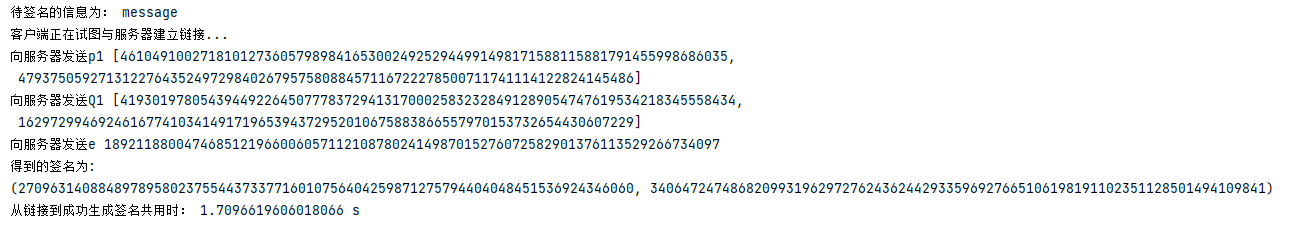
Project15



按照这种签名方案，定义并实现椭圆曲线的各种操作函数，实现起来是较为简单的。

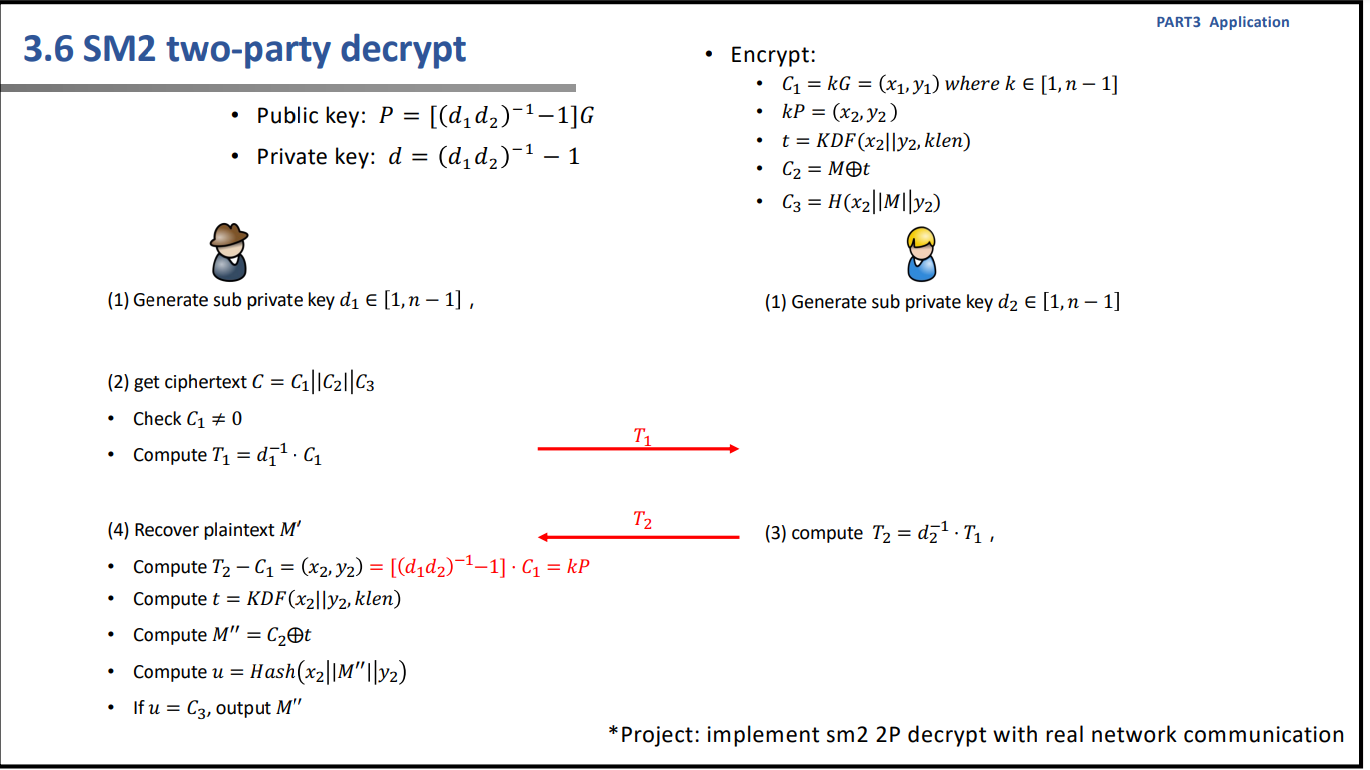
由于要求模拟真实网络，我们调用socket库，在本机网络上实现了两方签名。

可得结果：



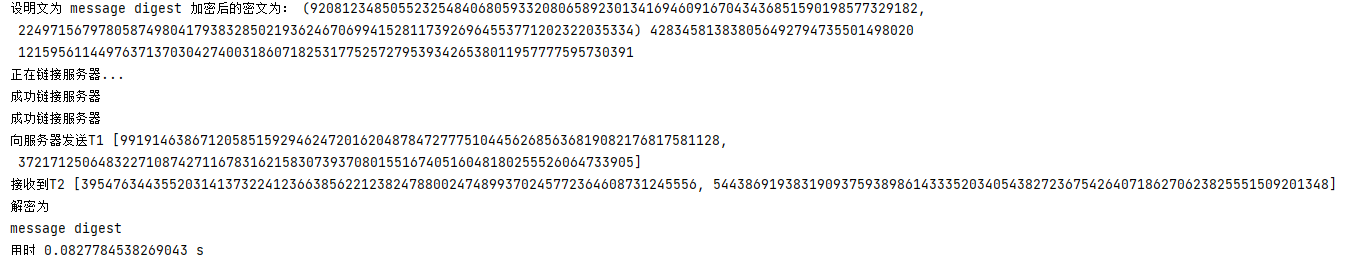
观察数据发现验证成功

Project16



按照这种方案，依据椭圆曲线运算和网络通信的原理在真实网络实现sm2解密方案

实验结果为：



Project17

此题为比较探究类体型，详情请见project17

Project18

实验内容

先下载[electrum]([Electrum Bitcoin Wallet](https://electrum.org/#download)) 比特币钱包，然后通过注册收获了30个比特币地址以及私钥。

比特币交易流程：

创建钱包：首先，您需要选择一个合适的数字钱包来存储您的比特币。可以选择在线钱包、硬件钱包或者软件钱包。创建钱包后，会生成一个独特的地址用于接收和发送比特币。

获取比特币：要获取比特币，您可以通过购买、挖掘或接受比特币作为支付。购买比特币可以在交易所进行，挖掘则需要算力和硬件设备，接受比特币可以通过与他人进行交易。

发起交易：当您想要将比特币发送给他人时，需要知道对方的比特币地址。在您的钱包中选择发送选项，并输入收款人的比特币地址和要发送的数量。

确认交易：在发送比特币之前，您需要确认交易细节，例如发送数量和收款地址是否正确。此外，还需要支付相应的交易费用，以确保交易被快速验证并添加到区块链中。

交易广播和确认：完成交易确认后，您的钱包会将交易广播到比特币网络中的节点。然后，矿工会将您的交易包含在新的区块中，并开始进行工作量证明的计算。一旦有足够的矿工验证并确认您的交易，它就会被添加到区块链中。

交易完成：一旦您的交易被添加到区块链中，您和对方就可以在区块链上查看交易记录。此时，比特币所有权已从您转移到了接收人手中，交易即完成。

可以在 <https://live.blockcypher.com/btc-testnet> 中查看详细的交易。

但之后没有使用上面的比特币钱包软件，而是获得了比特币测试地址

https://www.bitaddress.org)

//登录后还需要在网址后加上?testnet=true可获取测试地址

该交易可以在线 <https://live.blockcypher.com/btc-testnet/tx/bdd6a952621cc7653c43ebd6eb8e4d1e6b8b233d820e87aec2db8174675c291f/> 看到。

取得测试地址后可以在水龙头领取测试的比特币

之后便会有一笔交易，打入比特币进这个比特币地址。

在[比特币测试网交易查询]([BlockCypher Testnet Block Explorer |BlockCypher](https://live.blockcypher.com/bcy/)) 网址里面可以查询比特币地址，以及通过txid（交易id，唯一标识某笔交易的交易号）也可以找到某笔交易。

通过查询我的比特币地址：可以看到上面的余额情况：

再通过查询一笔交易的txid可以查到该交易。

交易哈希（Transaction Hash）：每笔比特币交易都有一个唯一的交易哈希，用于标识该交易。通过交易哈希可以在区块链上查找到该交易的详细信息。

输入（Inputs）：交易的输入是之前交易的输出。输入包含了之前某些比特币交易的输出信息，即发送方的地址和发送金额。可以通过解析输入找到发送方的地址和发送的比特币数量。

输出（Outputs）：交易的输出是指交易接收方的地址和接收的金额。可以通过解析输出找到接收方的地址和接收的比特币数量。

手续费（Fee）：交易手续费是发送方支付给矿工的费用，以便将交易添加到区块链中。通过解析交易数据，可以确定交易手续费的大小。

签名（Signatures）：交易数据还包含用于验证交易有效性的数字签名。签名是发送方使用其私钥对交易进行加密的结果，接收方使用发送方的公钥对签名进行解密和验证。

比如我们查询刚才进行的把比特币打入账户的交易，查到这笔交易的

我们的任务是爬取该部分信息并且解析出来，所以可以把网址交给自己写的爬虫程序，它从网址上爬取之后再解析出文本格式。

爬虫程序为parse.py，部分如下：

# 指定目标网址

url = 'https://live.blockcypher.com/btc-testnet/tx/f96ba3e152b89a7318f88566d192ef37fcd44dd25b5305b5e1dedcd6b2c7861e/'

# 发送GET请求获取网页内容

response = requests.get(url)

html\_content = response.text

# 使用BeautifulSoup解析HTML

soup = BeautifulSoup(html\_content, 'html.parser')

# 获取网页文本内容（去除HTML标签）

text\_content = soup.get\_text()

# 将文本内容保存到.txt文件中

with open('output after parse.txt', 'w', encoding='utf-8') as file:

file.write(text\_content)

即该笔交易的信息

## Project18-send a tx on Bitcoin testnet, and parse the tx data down to every bit

由于比特币交易未成功，无法得到tx文件

Project19

1，实验内容

通过中本聪的公钥，在没有密钥情况下伪造中本聪签名并通过验证

首先可以查到中本聪的公钥和签名：

```python

P\_Satoshi=(26877259512020005462763638353364532382639391845761963173968516804546337027093,48566944205781153898153509065115980357578581414964392335433501542694784316391)

sig\_Satoshi=(41159732757593917641705955129814776632782548295209210156195240041086117167123, 57859546964026281203981084782644312411948733933855404654835874846733002636486)

```

伪造签名的原理如下：

只需要中本聪的公钥便可伪造其签名（在验签过程中不重新计算e，而是直接使用给出的e进行验证）

验签过程如下：

from base import \*

import random

if \_\_name\_\_ == "\_\_main\_\_":

prikey, pubkey = Key\_Generation()

print("生成的 Satoshi 的公钥为:", pubkey)

u = random.randint(1, N - 1)

v = random.randint(1, N - 1)

R = add\_point(mul\_point(u, G), mul\_point(v, pubkey), P)

r = R[0] % N

s = (r \* calc\_inverse(v, N)) % N

sign = (r, s)

print("生成的签名为:", sign[0])

e = (r \* u \* calc\_inverse(v, N)) % N

r\_ = (mul\_point(calc\_inverse(s, N), add\_point(mul\_point(e, G), mul\_point(r, pubkey), P)))[0] % N

print("伪造的签名问:", r\_)

if r\_ == r:

print("签名伪造成功！")

else:

print("伪造失败！")```

额外传入了参数e，直接用e进行验证，而不是传入消息m，用消息m重新计算e再验证。

Project21

此题并未完全弄明白，在同学帮助下大致做出，

实验内容

收集待验证的签名和相关信息：

获得要验证的一系列Schnorr签名（每个签名包括公钥、消息、签名本身）。

收集对应的公钥。

计算批量验证的挑战值（Challenge Calculation）：

将收集到的每个签名的公钥串联起来，并对其进行哈希运算，生成挑战值。

确保使用相同的哈希算法和消息数据对每个公钥进行串联和哈希运算。

计算批量验证的伴随元素（Accumulated Point Calculation）：

对收集到的每个签名的公钥进行标量乘法运算，使用挑战值作为标量。

将每次乘法运算的结果累加，得到一个累积点（accumulated point）。

进行批量验证：

对收集到的每个签名的签名本身进行标量乘法运算，同样使用挑战值作为标量。

将每次乘法运算的结果累加，得到一个累积签名（accumulated signature）。

验证累积签名：

计算累积签名的验证方程式，检查是否满足等式：[累积签名] = [挑战值] × [累积点]。

如果验证方程式成立，则表明所有签名均有效；如果不成立，则至少有一个签名无效。

实现Schnorr签名的批量验证

(C:\Users\Mr.smile\AppData\Roaming\marktext\images\2023-08-02-17-28-18-image.png)

Schnorr signature batch verify的原理如下：

![](C:\Users\Mr.smile\AppData\Roaming\marktext\images\2023-08-02-17-29-07-image.png)

代码实现中，batch verify部分的实现：

```python

def verify\_sig(sig,M,P):

s\_sum=0

R\_sum=None

eiPi\_sum=None

for i in range(0,Index):

s\_sum+=sig[i][1]

R\_sum=point\_add(R\_sum,sig[i][0])

eiPi\_sum=point\_add(eiPi\_sum,point\_mul(P[i],e[i]))

sG=point\_mul(G,s\_sum)

ReP=point\_add(R\_sum,eiPi\_sum)

if(sG==ReP):

print('签名验证成功')

else:

print('签名验证失败')

```

### 2，运行指导

直接运行代码即可，需要给出消息，默认的Index=4，需要输入四条消息，然后批量验证后输出测试时间。

### 3，运行结果

运行代码结果如下：

> \*\*Index=4\*\* :

>

> \* Batch verify：

>

> ![](C:\Users\Mr.smile\AppData\Roaming\marktext\images\2023-08-02-17-32-45-image.png)

> \*\*Index=4\*\* :

>

> \* single verify time\*Index:

>

> ![](C:\Users\Mr.smile\AppData\Roaming\marktext\images\2023-08-02-17-35-27-image.png)

> \*\*Index=8\*\* :

>

> \* Batch verify:

>

> ![](C:\Users\Mr.smile\AppData\Roaming\marktext\images\2023-08-02-17-36-35-image.png)

> \*\*Index=8\*\* :

>

> \* single verify time\*Index:

>

> ![](C:\Users\Mr.smile\AppData\Roaming\marktext\images\2023-08-02-17-37-40-image.png)

上述通过比较普通验签和批量验签的效率，可以看出批量验证的效率要高于单次的验签。

批量验证在在区块链中，每个区块通常包含多个交易，并需要验证每个交易的签名。通过使用批量验证技术，可以在处理区块时同时验证所有交易的签名，提高整个区块链的吞吐量和效率。

综上，batch verify可以提高验签效率。

Project22，此题为探究题，主要内容请见project22