1.代码说明

生日攻击的目的是寻求一个基于sm3哈希值的弱碰撞，原理是一定长度和hash值结果2^32长度，在2^16密文空间中可以以50%以上的概率找到一个hash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（可以使用多线程进一步优化脚本性能），以便可以在较短时间内找到一个前16bit的hash弱碰撞。

如果寻找更长bit的碰撞，寻找时间也会相应变长。

运行指导

将源码clone到本地运行main函数即可运行生日攻击脚本。

软件环境：Visual Studio 2019

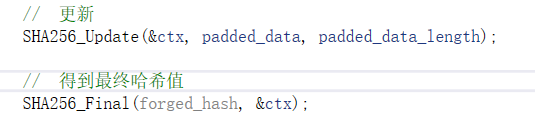
硬件环境：PC机

2. Rho方法用于消息扩展。它将输入的消息按照特定规则进行填充和扩展。

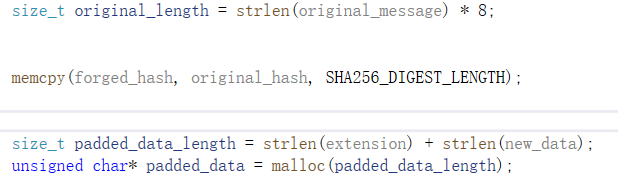
接下来，rho方法将填充后的消息分为若干个512比特（64字节）的块。如果消息长度超过了512比特，则将其分成多个块。  
  
最后，对每个块应用SM3算法的压缩函数进行处理，以生成哈希值。  
  
总之，rho方法是SM3算法中用于对消息进行扩展和分块处理的重要步骤，确保输入的消息能够适应SM3算法的运算要求。通过rho方法，可以将任意长度的消息转化为一系列512比特的块，然后对每个块进行进一步的处理，最终生成SM3哈希值。

3. sha256\_length\_extension 函数用于执行长度扩展攻击。它接收原始消息(original\_message)、原始哈希值(original\_hash)、扩展内容(extension)、新数据(new\_data)和一个指向已分配的内存区域的指针(forged\_hash)作为参数。  
 我们使用OpenSSL库提供的SHA-256哈希函数，创建一个SHA-256哈希上下文(SHA256\_CTX)对象，并初始化它。然后，使用SHA256\_Update函数将原始消息(original\_message)的内容添加到哈希计算中。最后，使用SHA256\_Final函数获取最终的哈希值，存储在original\_hash数组中。



  
 将原始哈希值(original\_hash)复制到预先分配的内存区域(forged\_hash)中。计算扩展数据的总长度，并为其动态分配内存。然后，使用memcpy函数将扩展内容(extension)和新数据(new\_data)复制到该内存区域中。



  
 计算新消息的总长度，包括原始消息长度、扩展内容长度和填充数据的长度。  
 创建一个新的SHA-256哈希上下文对象，并将它初始化。接下来，将forged\_hash中的字节顺序转换为特定的整数格式，并将其存储在哈希上下文的h数组中。还将新消息的总长度(new\_length)赋值给哈希上下文的Nl字段，并将填充数据(padded\_data)的指针赋值给哈希上下文的data字段。

最后，调用final函数获取最终的哈希值。

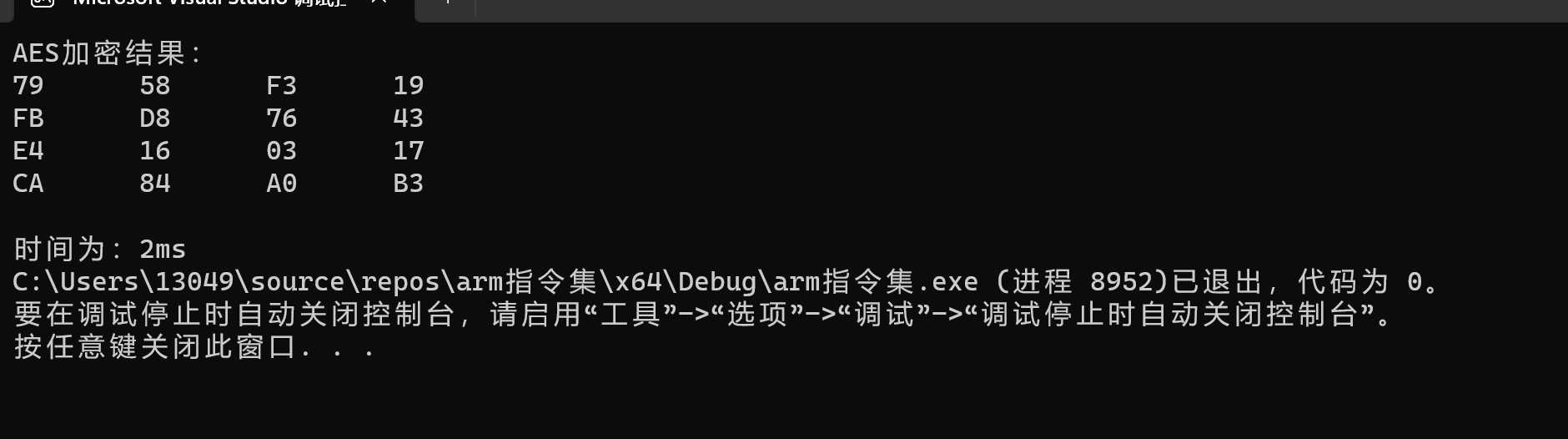
4.以下是优化SM3算法的思路（软件）：  
  
使用位运算代替乘法和除法：在SM3算法中，涉及到了大量的乘法和除法运算。通过使用位运算（如左移、右移、按位与、按位异或等）来代替乘法和除法，可以提高运算速度。  
  
利用并行计算：使用SIMD指令集或多线程技术来充分利用多核处理器的计算能力，加快计算速度。  
  
数据对齐和内存访问优化：合理地对数据进行对齐，以及优化内存访问模式（利用局部性原理），可以提高数据传输效率，减少内存访问的时间，从而加快算法执行速度。  
  
使用表格预计算：SM3算法中包含了很多固定的常量和查找表。可以在算法初始化阶段预先计算并存储这些常量和查找表，以减少运行时的计算量。  
 5. Project 5   
 我们可以准备任意一组数据，作为叶节点输入。对于每个叶节点，我们使用该标准推荐的哈希函数SHA-256计算其哈希值。计算得到的哈希值将被用于构建默克尔树。根据该标准，内部节点的哈希值是其子节点哈希值的连接结果再进行哈希计算得到的。编写一个函数来计算内部节点的哈希值。  
 使用上述函数，我们逐层构建默克尔树。首先，将所有叶节点添加到树中。然后，按照标准规定，将叶节点两两组合计算内部节点的哈希值，直到树的根节点生成。最后，验证根节点的哈希值是否是由所有叶节点数据计算得到的结果。  
 我们通过调用build\_merkle\_tree函数将数据传递给默克尔树构建程序，得到一个存储了所有节点哈希值的列表，其中最后一个节点即为默克尔树的根节点。

# 9.AES/SM4softwareimplementationAES

# 加密过程涉及到4种操作：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字节的明文、密文和轮密钥都以一个4x4的矩阵表示IMG_256



**实现结果**



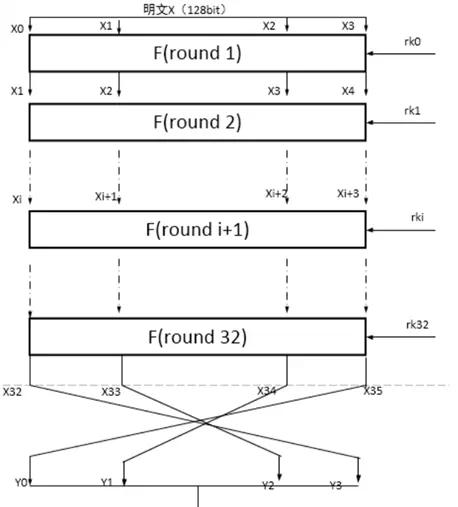
# SM4

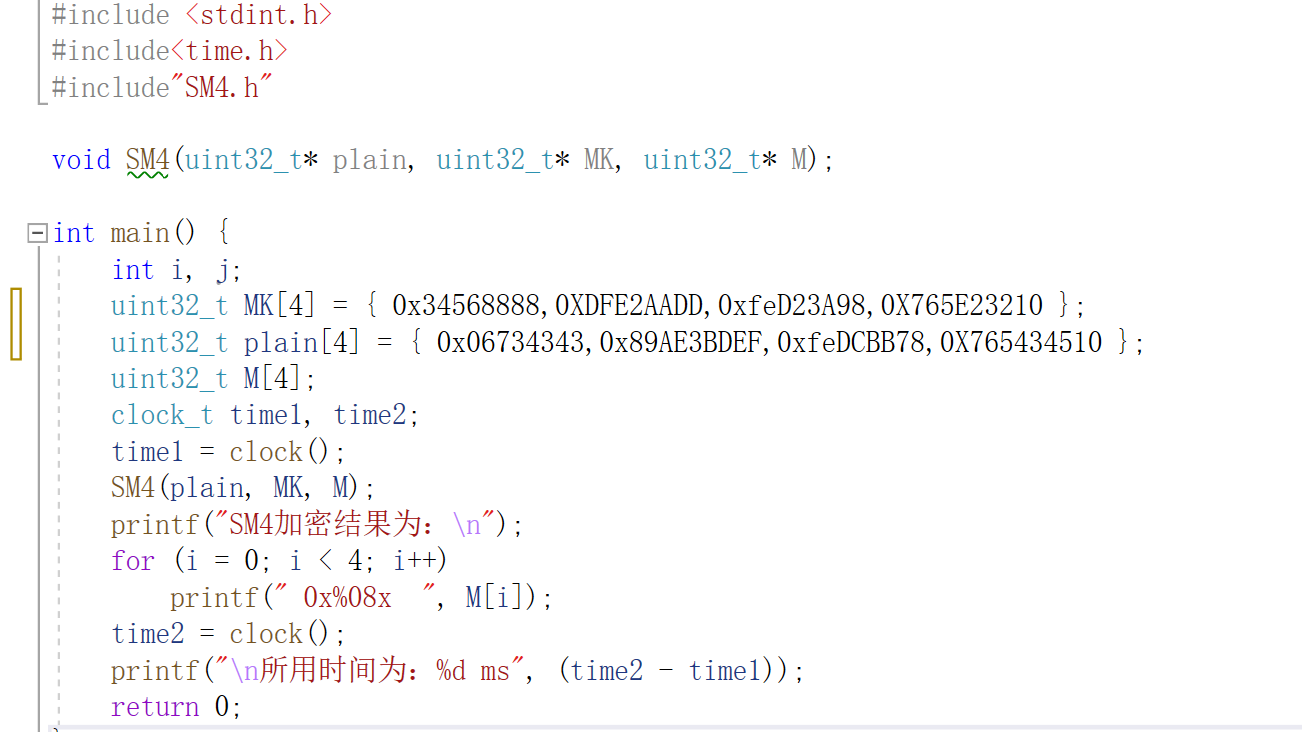
1.与DES和AES算法类似，SM4算法是一种分组密码算法。

2. 其分组长度为128bit，密钥长度也为128bit。

3.加密算法与密钥扩展算法均采用32轮非线性迭代结构，以字（32位）为单位进行加密运算，每一次迭代运算均为一轮变换函数F。

4. SM4算法加/解密算法的结构相同，只是使用轮密钥相反，其中解密轮密钥是加密轮密钥的逆序。





实现结果



# 10: report on the application of this deduce technique in Ethereum with ECDSA

**ECDSA在以太坊中的应用**

交易签名验证：在以太坊中，每个交易都需要进行数字签名来验证其合法性。发送方使用私钥对交易进行签名，接收方使用发送方的公钥和签名来验证交易的真实性。ECDSA算法被用于生成和验证这些数字签名，确保交易的安全性和完整性。

合约部署和调用：以太坊中的智能合约也需要进行数字签名来验证其合法性。合约的创建者使用私钥对合约进行签名，以太坊网络中的节点使用公钥和签名来验证合约的真实性。ECDSA算法被用于生成和验证这些数字签名，确保合约的安全性和完整性。

账户身份验证：以太坊中的账户也可以使用ECDSA算法进行身份验证。用户可以使用私钥对其账户进行签名，以太坊网络中的节点使用公钥和签名来验证账户的真实性。这种身份验证机制可以防止恶意用户冒充其他账户进行欺诈行为。

消息验证：以太坊中的消息也可以使用ECDSA算法进行验证。发送方使用私钥对消息进行签名，接收方使用发送方的公钥和签名来验证消息的真实性。这种消息验证机制可以确保消息的完整性和真实性。

总之，ECDSA在以太坊中广泛应用于交易签名验证、合约部署和调用、账户身份验证以及消息验证等方面，保证了以太坊网络的安全性和完整性。

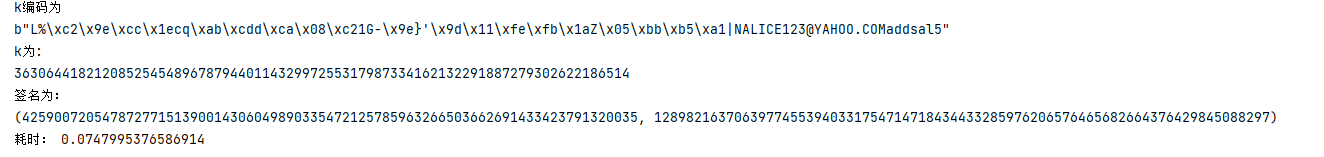


11.按照规定，我们希望k能做到随机，不会出现k值相同的情况。故对其进行补字和哈希。

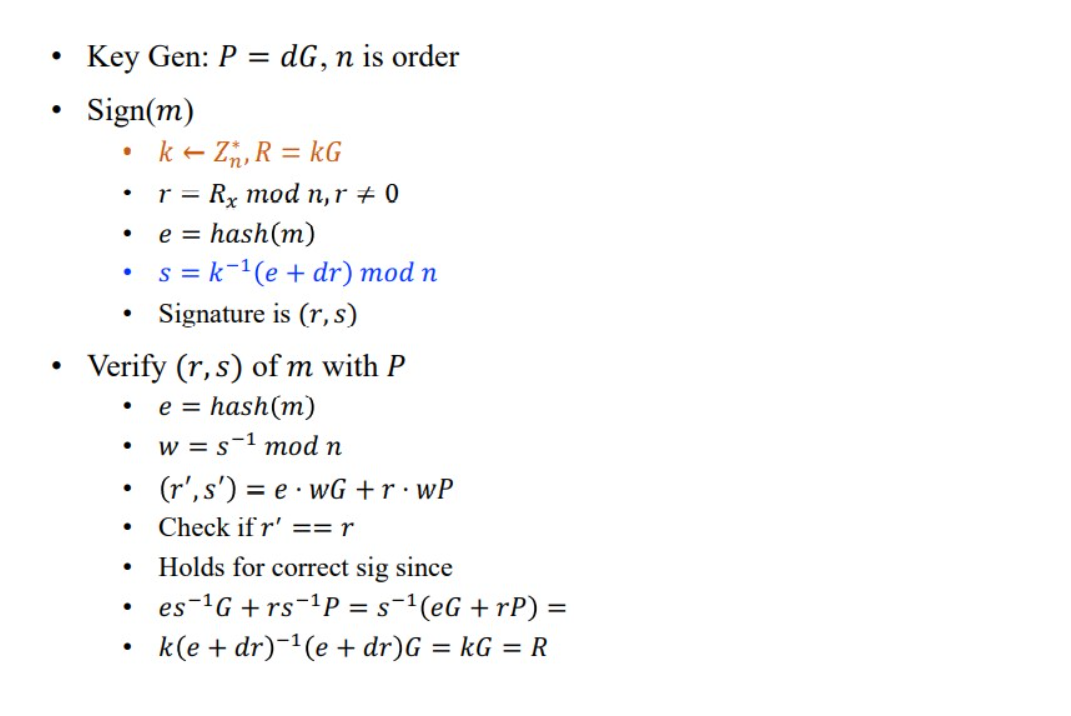
最后k=（d||ID||"addsalt"），通过混淆算法和加入两段 混淆字符串的办法，基于哈希函数的定义，不太可能出现两个一样的k值。哈希算法我们采用sm3

结合sm2算法完成该实验

最后可得结果：

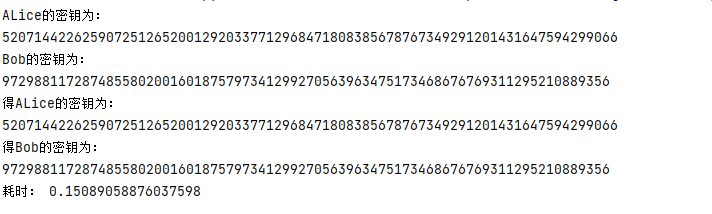


12.一：实验原理

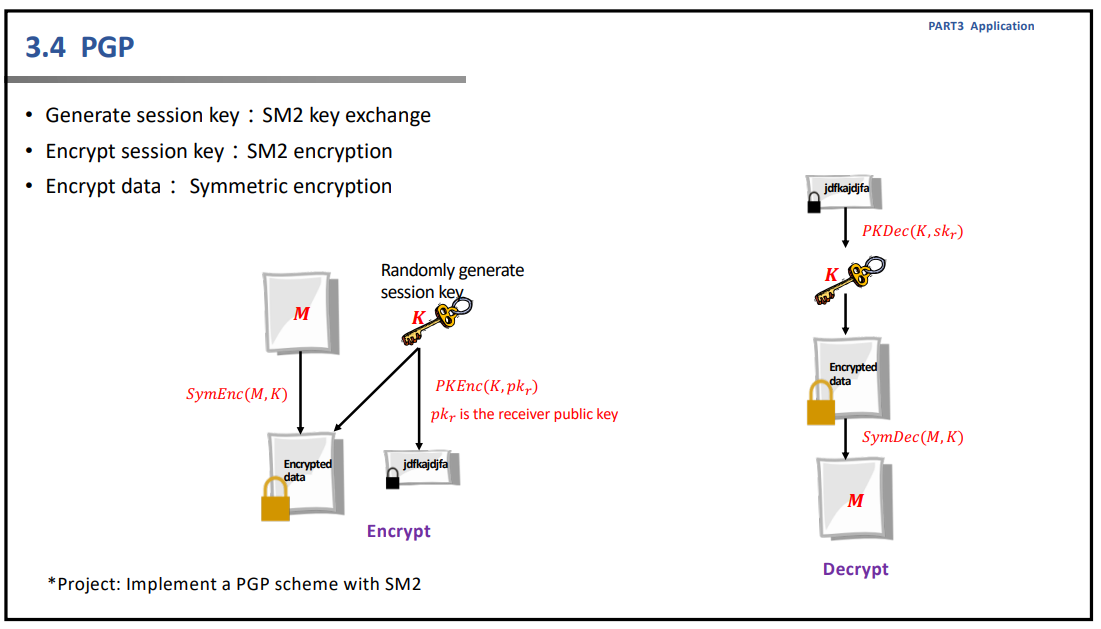


在选取了同样的k值时，根据这种算法可以还原出密钥。

二：实验结果：



14.



按照此结构，实现PGP方案

于是在代码中可以体现出:

在加密函数encrypt中：

我们随机产生密钥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

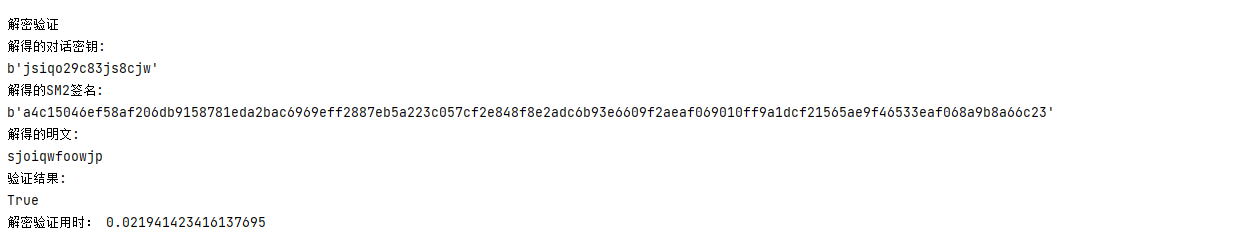
拼接发送

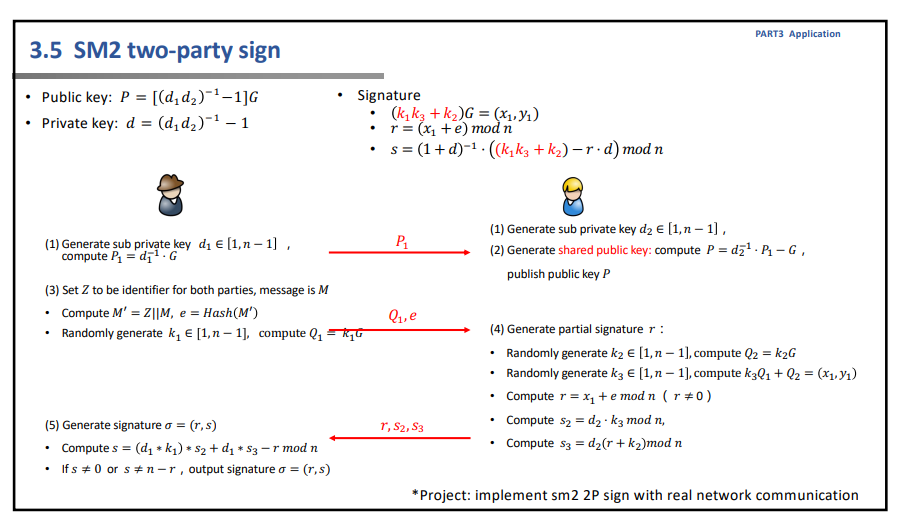
在解密函数decrypt中：

按照顺序分开解密，验证公钥

最后结果为：



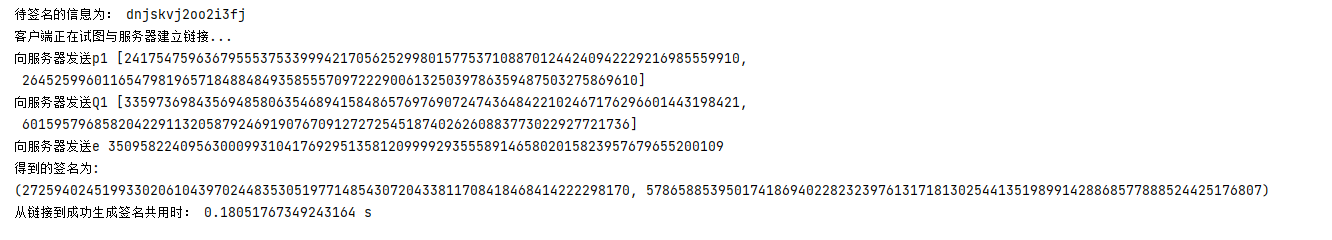


15.

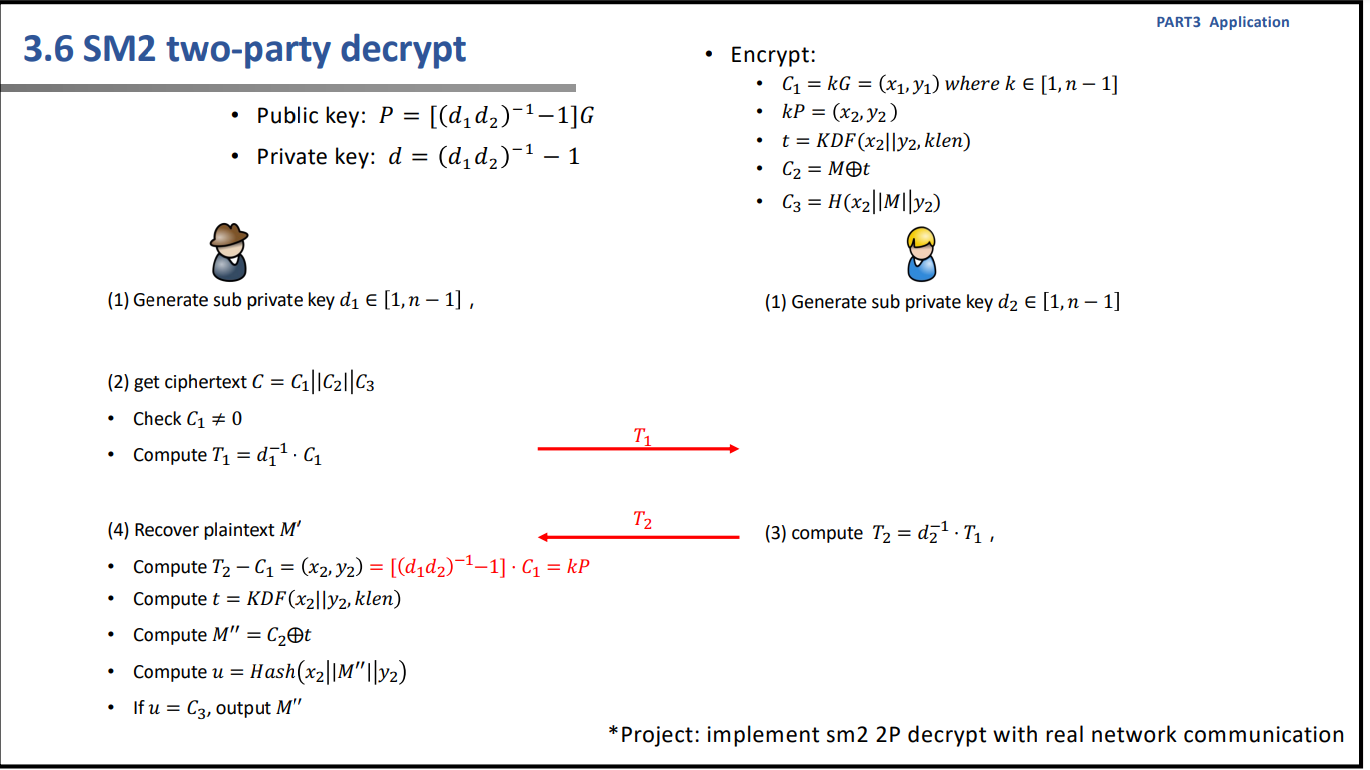
按照这种签名方案，在实现了椭圆曲线操作的基础上，实现了可交互的sm2签名方案

由于要求在真实网络上实现，于是调用socket库，编程实现网络通信。轮流调用两个py文件，可以在本机上实现真实网络的交互加密。

最后结果为：



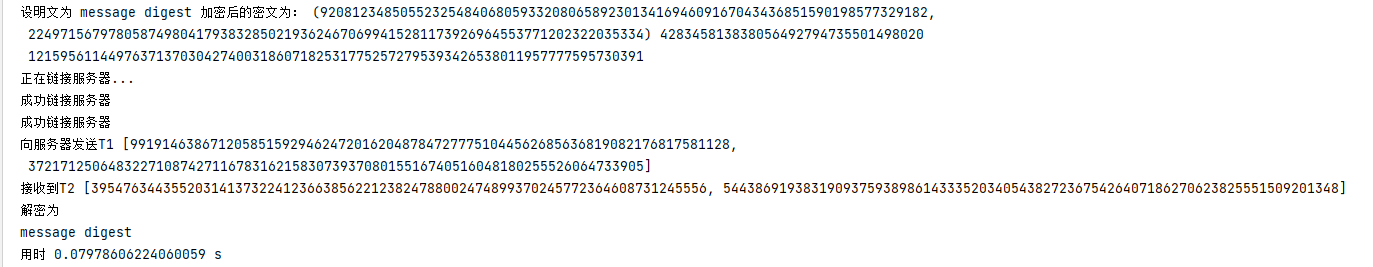
观察数据发现验证正确

16.

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

通过调用socket库，实现了在本机 真实网络上通信的解密方案。轮流调用两个py文件，即可运行成功

实验结果为：



17 18 19 22详见文件