|  |
| --- |
|  |
| **创新创业实践汇总报告** |
|  |
|  |
|  |
|  |

|  |
| --- |
|  |

**小组成员**

**高 畅（个人）**

**202100150084**

**目录**

**[一、project1](#_Toc7293) 3**

**[二、project2](#_Toc8564) 5**

**[三、project3](#_Toc22391) 8**

**[四、project4](#_Toc30308) 10**

**[五、project5](#_Toc12512) 13**

**[六、project8](#_Toc16005) 17**

**[七、project9](#_Toc16005) 18**

**[八、project10](#_Toc16005) 22**

**[九、project11 24](#_Toc16005)**

**[十、project14](#_Toc16005) 26**

**[十一、project15](#_Toc16005) 28**

**[十二、project16](#_Toc16005) 30**

**[十三、project17](#_Toc16005) 34**

**[十四、project18](#_Toc16005) 39**

**[十五、project19 4](#_Toc16005)2**

**十[六、project21 4](#_Toc16005)3**

**十[七、project22 4](#_Toc16005)4**

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

代码说明：

此项目是实现SM3生日攻击。

生日攻击是利用概率论中的生日问题,找到冲突的Hash值,伪造报文,使身份验证算法失效。生日攻击的基本思想是寻找两个不同的输入消息，它们经过SM3算法后产生相同的hash值。根据生日悖论，n位的哈希值经过2^(n/2)次尝试预计可以约1/2的概率产生一次碰撞。

本项目我先用python实现了FF、GG、p0、p1、消息填充、消息扩展、压缩、迭代等函数，进而实现了SM3算法，再进行生日攻击，实现SM3生日攻击的前n位比特的碰撞的步骤如下：

（1）定义v函数，接受参数n，生成一个随机8位（可改为自己想要的位数）字符串s，并将其转换为字节数组a。然后，对a进行SM3哈希，并截取前n位比特得到j。最后，将a和j作为一对返回。

（2）定义birthday\_attack函数，接受参数n，将n转换为前n位比特数，并调用v函数生成两个不同的pair（a1, j1）和（a2, j2）。然后，在for循环中比较a1、a2，j1、j2。如果找到两个不同的字符串（即a1≠a2），且它们的前n位哈希值相同（即j1=j2），则认为攻击成功。如果不满足，则继续生成新的随机字符串对，并进行下一轮循环。

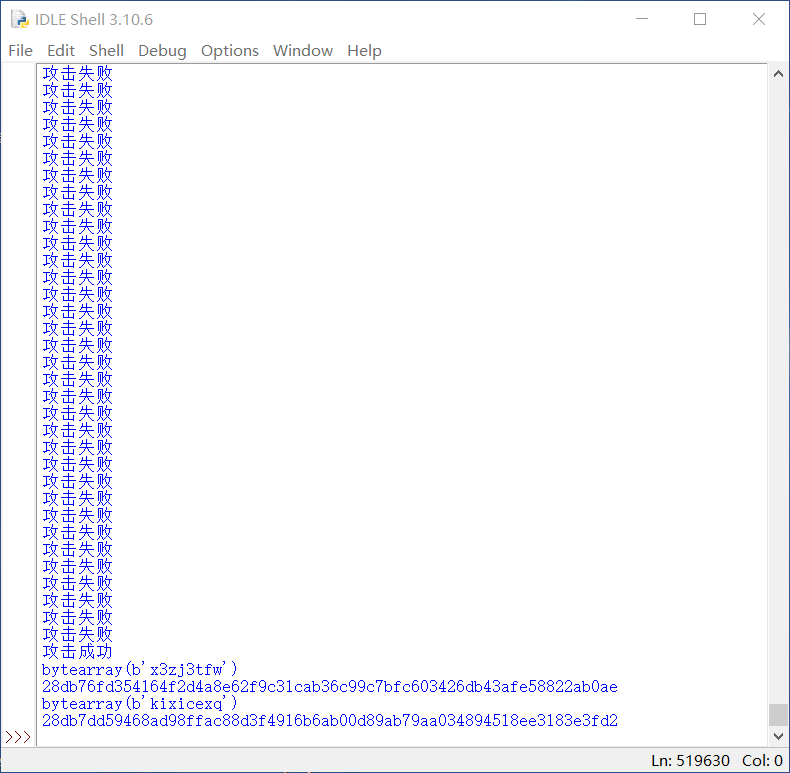
Ps：birthday\_attack(n)函数中的n=int(n / 4)是因为产生的随机消息一位单位为一个字节，除4后单位为1bit。

birthday\_attack(n)中可根据需要将函数变量n改为任意整数。

实现方式：python

效果：在自己电脑上CPU：11代i7

这里以birthday\_attack(20)为例，展示其结果：



分工：自己独立完成

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

代码说明：

此项目是实现SM3的Rho攻击。

Rho攻击是一种基于碰撞概率的攻击方法，适用于部分哈希函数。这种攻击方法利用了哈希函数的迭代结构和碰撞概率，通过找到两个具有相同哈希值的输入来构造攻击。

基本原理是通过选择不同的初始输入值，并跟踪哈希函数迭代过程中产生的中间值，以期望找到具有相同中间值的两个输入，进而产生哈希碰撞。

实现SM3的前n位Rho攻击：

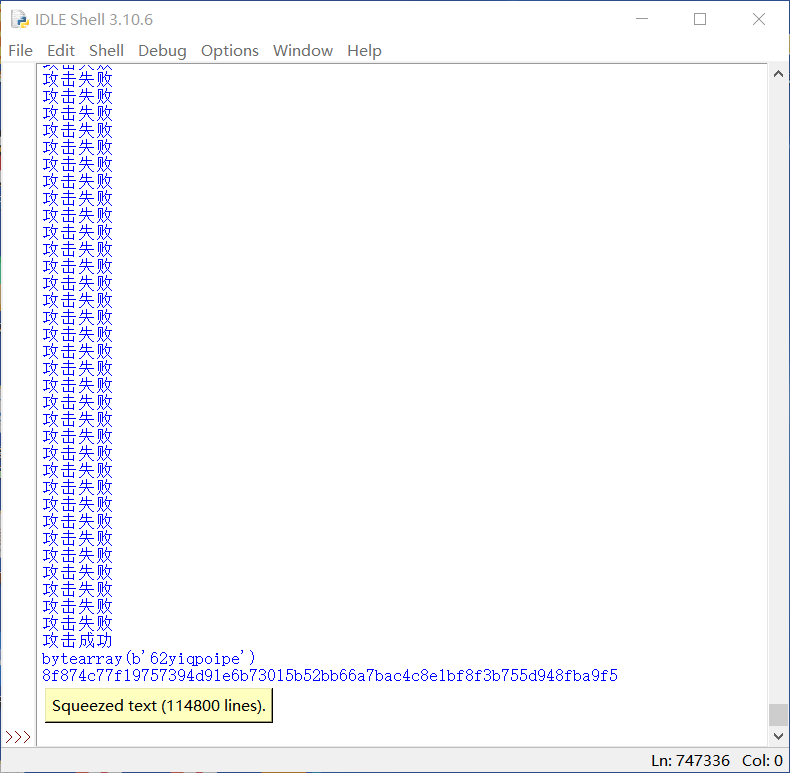
定义generate\_random\_string函数，接受一个参数length，生成一个长度为length的随机字符串。定义rho\_attack函数，接受一个参数n，将n转换为前n位比特数。然后，通过for循环迭代生成大量随机字符串b，并计算其SM3哈希值的前n位比特c。检查c是否在列表a中，如果在，则认为攻击成功，并打印出成功信息以及找到的两个不同字符串和哈希值。如果不在，将c添加到列表a中，并继续生成新的随机字符串b。

注：rho\_attack(n) 中可根据需要将函数变量n改为任意整数。

实现方式：python

效果：在自己电脑上CPU：11代i7

这里以rho\_attack(40)为例，展示其结果：



Squeezed text(114800 lines)中的内容：



分工：自己独立完成

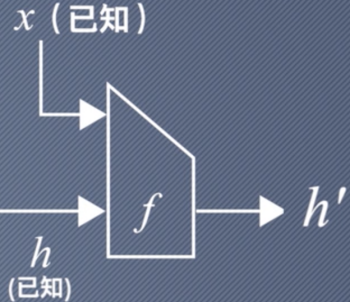
**\*Project3: implement length extension attack for SM3, etc.**

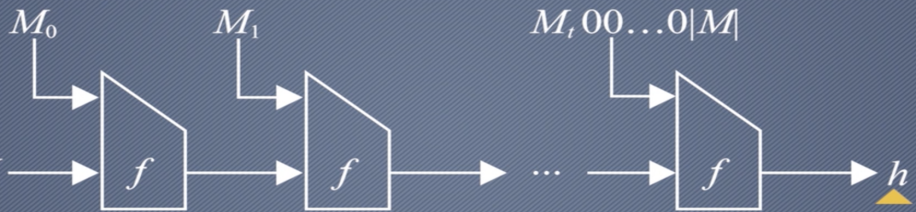
代码说明：

此项目是实现长度扩展攻击，在此实现了对SM3的长度扩展攻击。

SM3长度拓展攻击是一种利用哈希函数的长度扩展性质的攻击方式。SM3算法在哈希计算中采用了MD结构，这种结构使得：可以通过已知原始消息M 的长度及其hash值h，令z= 0^d|| |M| ||x，x为任意长度的附加消息，则根据h可计算h’,满足H(M||z)=h’。(M||z, h’)即是一个利用长度扩展攻击得到的伪造。

这是因为MD结构在处理消息块时，不对整个消息进行完整的处理，而是对每个消息块进行局部处理，因此可以对部分原始消息进行长度拓展。





步骤：

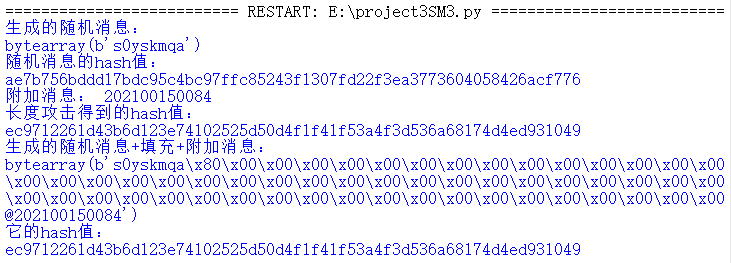
1. 定义len\_attack函数，接受参数m和length，m为原始消息（a）的hash值，length为原始消息的长度。在进行长度扩展攻击时，我们不知道原始消息具体是什么，只知道原始消息的长度，因此我们随意构造相同长度的消息（在这里我们每长度都用‘1’来构造）。
2. 对SM3函数进行改造：增加了两个参数：一个为IV，一个为flag。当flag为0时，按正常SM3计算消息m的hash值；当flag为1时，实际上是实现了从m中截取附加消息x，并利用h，x计算h’。
3. 我们将构造的相同长度的消息进行填充（fill）后，将附加消息'202100150084'拼接到其后面，然后将其传给稍微改造后的SM3函数的m，把原始消息的hash值m传给SM3函数的IV，SM3函数的flag取1。计算得到长度扩展后的结果c（即h’）。

（4）再构造出a|| 0^d|| |a| ||x，进行SM3计算出H(a|| 0^d|| |a| ||x)，与h’比较，若相等，则长度扩展攻击成功。

Ps：可任意更改生成的随机消息的长度，也可任意更改附加消息。

实现方式：python

效果：在自己电脑上CPU：11代i7



分工：自己独立完成

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

代码说明：

此项目是实现了SM3的优化。

首先，按照老师PPT里发的内容完成了SM3的基本实现（见project4），用来与优化后的SM3比较，作为对照。

此项目我主要在sm3\_compress(uint32\_t digest[8], const unsigned char block[64])函数里实现了优化。优化主要分为两种：

第一种优化方式，是利用了SIMD指令集优化。

[SIMD](https://so.csdn.net/so/search?q=SIMD&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_42817735/article/details/_blank)，即Single Instruction, Multiple Data，一条指令操作多个数据．是CPU基本指令集的扩展。SIMD 指令集能让一条指令同时对八路数据进行加减乘除、与非或操作，提升运行效率。

在本项目的优化中，以下面这段代码为例说明：

\_\_m256i a1 = \_mm256\_loadu\_epi32(&W[0]); //用SIMD指令集优化

\_\_m256i b1 = \_mm256\_loadu\_epi32(&W[4]);

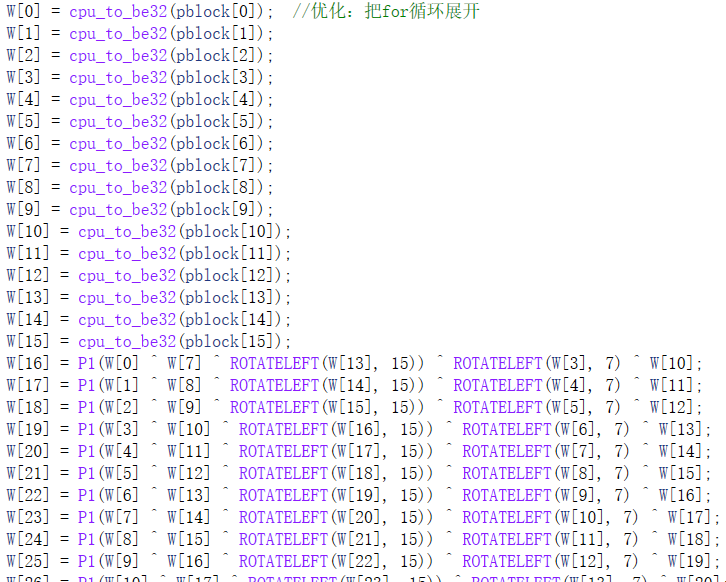
\_\_m256i c1 = \_mm256\_xor\_si256(a1, b1);

\_mm256\_storeu\_epi32(&W1[0], c1);

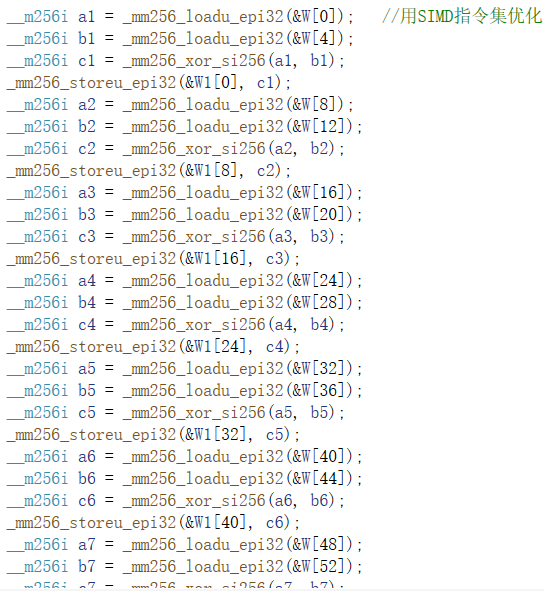
上述代码中使用了AVX2的256位寄存器\_\_m256i来处理数据。为了适应内存对齐的要求，我们使用了\_mm256\_loadu\_si256和\_mm256\_storeu\_si256函数来加载和存储未对齐的数据。\_mm256\_xor\_si256是AVX2指令集中用于执行两个数据按位异或操作的函数。

利用AVX2指令集的256位向量处理能力，我同时从数组W的两个位置加载数据，然后对数据进行按位异或，最后将结果存储回数组W1。这样的优化可以提高并行性能，并减少了对内存的多次读写操作，从而提高了代码的执行效率。

第二种，是比较简单的一种，是把for循环展开。我把一些循环次数较少的for循环直接展开，缩短了运行时间，部分如下图：



用第一种优化方式（即SIMD指令集）的地方也将for循环展开了，部分如下图：



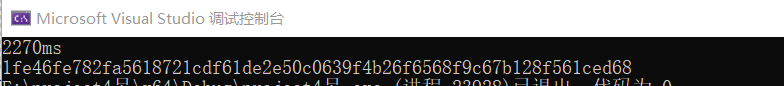
实现方式：

用c++实现

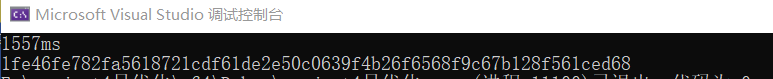
效果：

在自己电脑上CPU：11代i7

未优化代码运行1000000次所用时间及hash值：



优化后代码运行1000000次所用时间及hash值：



分工：

自己独立完成

**\*Project5: Impl Merkle Tree following RFC6962**

代码说明：

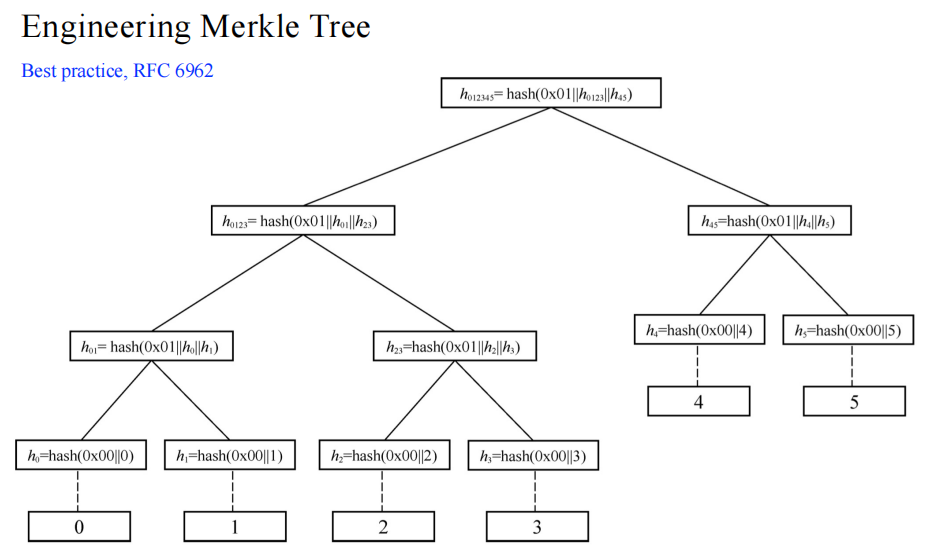
此项目是遵循RFC6962实现：

1. 构造一个具有100000个叶节点的Merkle树
2. 证明某个元素在这个Merkle树里
3. 证明某个元素不在这个Merkle树里

**Merkle树的建立：**

Merkle Tree是一种基于哈希值构建的树状数据结构。它通过对数据块进行递归地哈希运算，从而生成一个具有高效验证和安全性特性的树。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。

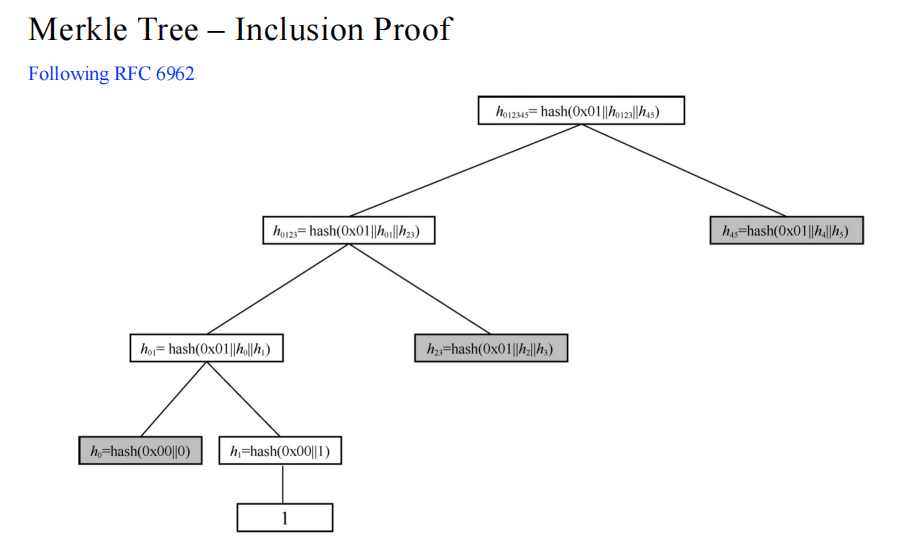
遵循RFC6962的Merkle树要按照以下规则构建（下图以6个叶节点为例建立Merkle树）：



在本项目中我创建了一个具有100000个叶节点（—）的Merkle树。

**存在性证明：**

对于随意选择的一个结点（函数inclusion里参数可改成0——99999中的任意一个整数，证明其hash值在Merkle tree里），我可以根据其的hash 值以及树中的所有相关节点（如下图阴影所示）计算出一个根值，与建立的Merkle tree的根值进行比较。如果它们是相同的，那它存在于Merkle树中。



**排除性证明：**

排除性证明只能用于叶节点hash前的数是紧挨着递增或递减的Merkle tree。

对于随意选择的一个结点（函数inclusion里参数可改成0——99999中的任意一个非整数，证明其hash值不在Merkle tree里），我可以根据验证其下取整的整数和其上取整的整数的hash值都在Merkle tree里来证明其hash值不在Merkle tree里。

因为一个数下取整和其上取整的数的hash在Merkle tree里是相邻的，而数又是递增或递减的，那么这个数下取整和其上取整的数的hash值都在Merkle tree里就可以说明这个数的hash值不在Merkle tree里。

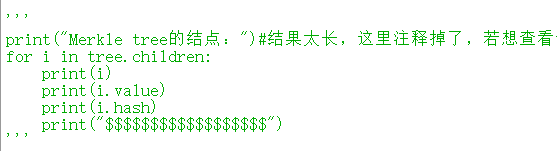
实现方式：python

效果：在自己电脑上CPU：11代i7

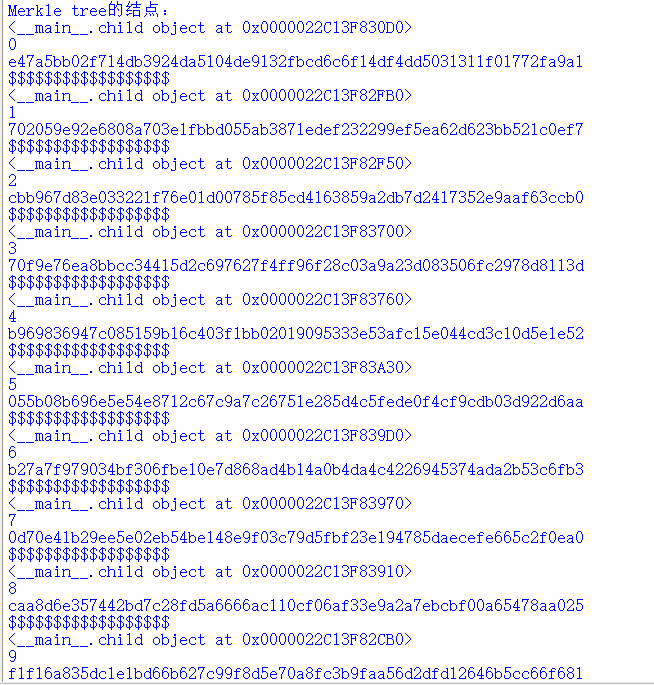
创建了一个具有100000个叶节点（—）的Merkle树，并输出其根的hash值。证明了6的hash值在Merkle tree里。证明了12.5的hash值不在Merkle tree里。



还可以输出创建的Merkle tree的结点，但是结果太长，代码里注释掉了那部分，若想查看这个有100000个叶节点的Merkle tree的每个结点，解注释这段即可：



这里展示Merkle tree的结点的一部分：



分工：自己独立完成

**\*Project8: AES impl with ARM instruction**

代码说明：

此项目是要求在ARMv8架构上使用ARMv8—AES内部函数，进行AES的加密以及解密。利用ARMv8的AES扩展指令进行AES的加密以及解密，可以优化AES算法，使加密和解密更高效。在ARM 的 SIMD 指令集中，AESE即为AES的单轮加密，涵盖了 AddRoundKey, SubBytes 和 ShiftRows 。

在https://developer.arm.com/architectures/instruction-sets/intrinsics/#q=AES（指令集）中，我主要用了下面两个函数：

（1）AES单轮加密：

uint8x16\_t vaeseq\_u8(uint8x16\_t data, uint8x16\_t key);

（2）AES混淆列：

uint8x16\_t vaesmcq\_u8(uint8x16\_t data);

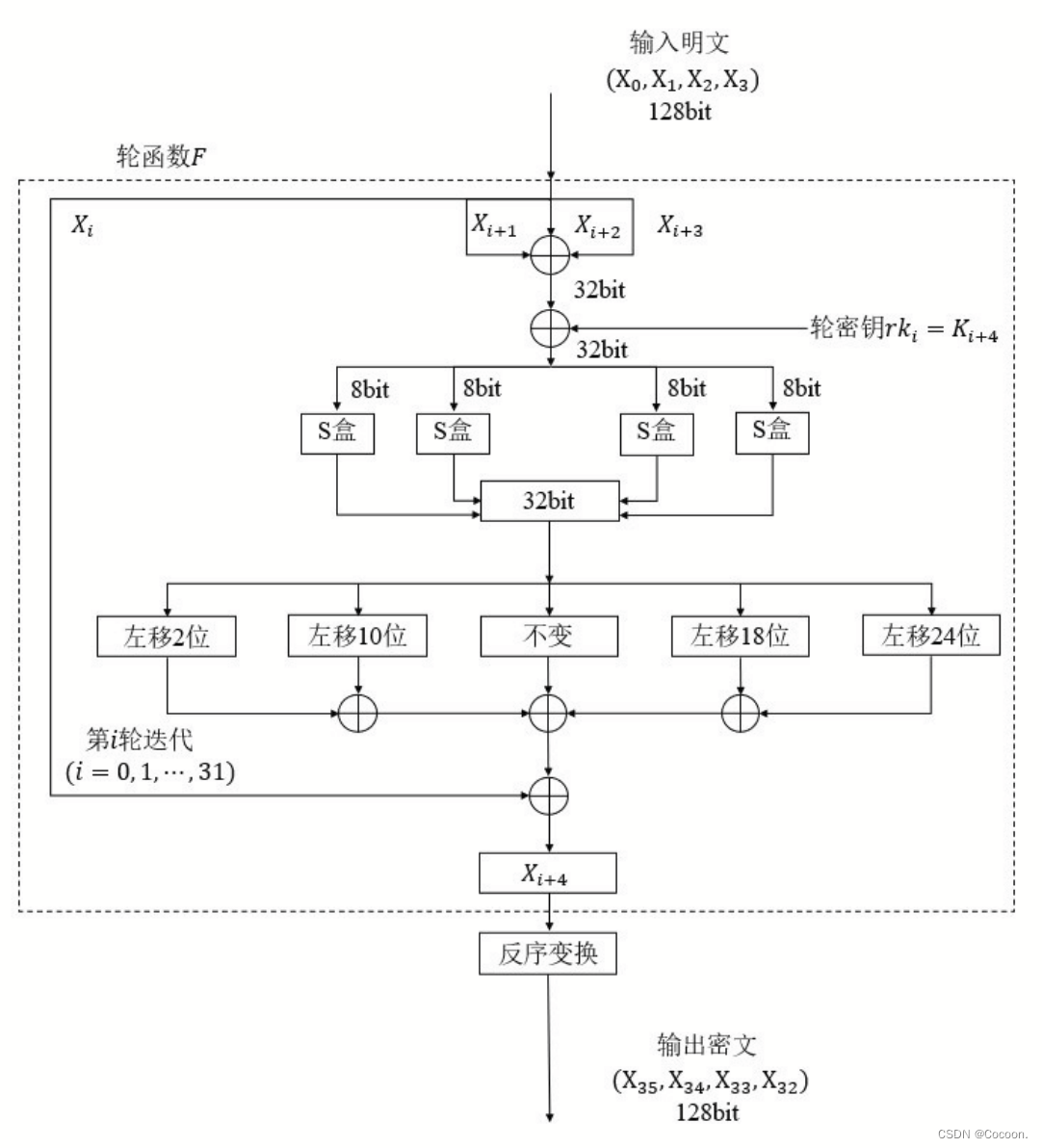
实现方式：c++

分工：自己独立完成

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

SM4的代码说明：

## 流程图：



线性变化：主要是移位运算和异或运算，输入输出都为32位。例如B为32位，则运算为B⊕(B<<<2) ⊕(B<<<10) ⊕(B<<<18) ⊕(B<<<24)

密钥扩展算法：输入128位的密钥，进行32轮迭代，每一轮产生一个32位的轮密钥，共产生32个轮密钥。先进行(k0,k1,k2,k3)=(mk0⊕fk0,mk1⊕fk1,mk2⊕fk2, mk3⊕fk3),然后进行32轮迭代，每一轮为rki=ki+4=ki⊕t’(ki+1⊕ki+2⊕ki+3⊕cki)，其中rki为i轮的密钥，cki与fki都为常数，t’运算为先进行s盒代换，然后进行线性变化，只不过这里的线性变化为B⊕(B<<<13) ⊕(B<<<23)。

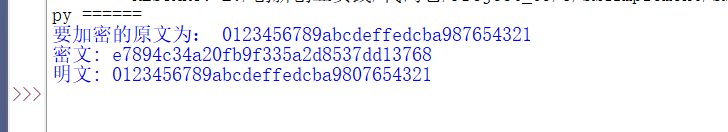
加密算法：输入为128位，即4个32位的字，输出也为128位。共有32轮迭代，每一轮使用一个32位的轮密钥。每一轮的运算为x0⊕t(x1⊕x2⊕x3⊕rk)。之后将得到的x35,x34,x33,x32再进行一个反序处理，作为密文。

其中rk为该轮轮密钥，x0,x1,x2,x3为4个32位的字，t运算包含s盒代换和线性变换，即每一轮的运算为x0⊕[s(B)] ⊕[s(B)<<<2] ⊕[s(B)<<<10] ⊕[s(B)<<<18] ⊕ [s(B)<<<24] ,B为x1⊕x2⊕x3⊕rk

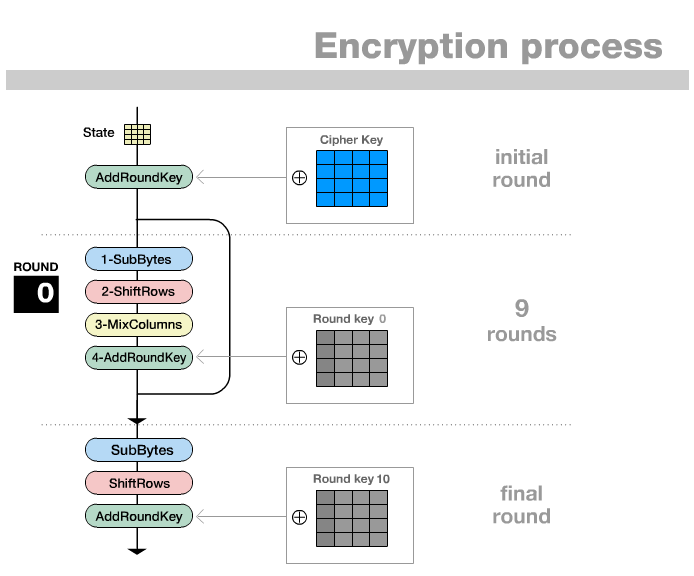
解密算法：与加密算法相同,只是轮密钥的使用顺序相反。

实现方式：python

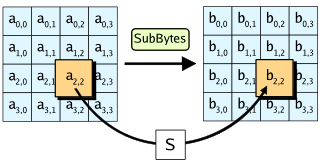
## 运行结果：



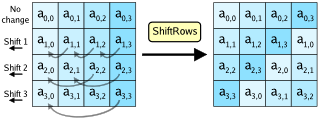
**AES：**

[](AES.exe)

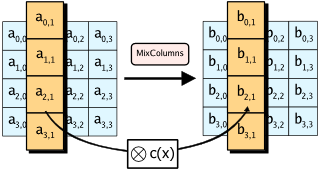
S盒：

[](http://www.answers.com/main/Record2?a=NR%26url=http://commons.wikimedia.org/wiki/Image:AES-SubBytes.svg)

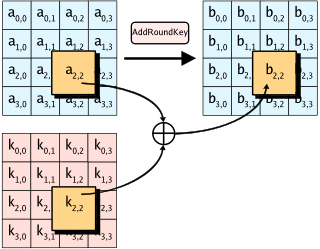
行移位：

[](http://www.answers.com/main/Record2?a=NR%26url=http://commons.wikimedia.org/wiki/Image:AES-ShiftRows.svg)

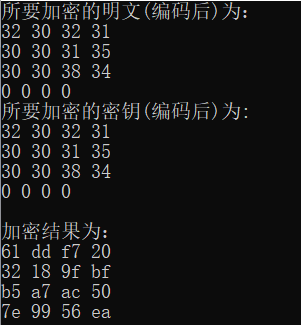
列混合：

[](http://www.answers.com/main/Record2?a=NR%26url=http://commons.wikimedia.org/wiki/Image:AES-MixColumns.svg)

轮密钥异或：

[](http://www.answers.com/main/Record2?a=NR%26url=http://commons.wikimedia.org/wiki/Image:AES-AddRoundKey.svg)

结果：



分工：自己独立完成

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

**Ethereum with ECDSA**

说明：

一、ECDSA 概述

ECDSA 是 Elliptic Curve Digital Signature Algorithm（椭圆曲线数字签名算的缩写。它是一种基于椭圆曲线密码学的数字签名算法。ECDSA 结合了椭圆曲线上的离散对数问题和哈希算法的安全性特性，提供了一种高效、安全的数字签名方案。

ECDSA 被广泛应用于各种加密和认证场景，包括数字证书、电子支付系统、区块链技术等。它的安全性基于椭圆曲线离散对数问题的困难性，即在给定椭圆曲线上的基点和一个点的情况下，计算出这个点的离散对数是困难的。这使得ECDSA 具有相对较短的密钥长度和较高的计算效率。

ECDSA 的基本原理是使用私钥对消息进行签名，生成数字签名，然后使用对应的公钥对签名进行验证，确保签名的完整性和真实性。ECDSA 通过椭圆曲线上的点运算实现签名和验证操作，同时结合了哈希函数以增强安全性。

1. ECC 的实现

签名过程：

1. 选择一条椭圆曲线 Ep(a, b)和基点 G。

2. 选择私钥 k（k < n），其中 n 是基点 G 的阶数，通过计算公钥 K = kG。

3. 生成随机数 r（r < n），计算点 R = rG。

4. 将原数据和点 R 的坐标值 x、y 作为参数，计算哈希值 Hash = Hash(原数据,x, y)（通常使用哈希函数）。

5. 计算 s ≡ r - Hash \* k (mod n)。

6. 如果 r 或 s 中有一个为 0，则重新从步骤 3 开始执行。

验证过程：

1. 接收方在收到消息 m 和签名值(r, s)后进行以下运算。

2. 计算 sG + H(m)P = (x1, y1)，其中 H(m)是对消息 m 进行哈希计算的结果。

3. 验证等式：r1 ≡ x1 (mod n)，其中 r1 为计算得到的临时值。

4. 如果等式成立，则接受签名；否则，签名无效。

三、ECDSA在以太坊中的应用：

ECDSA作为以太坊中重要的加密算法之一，广泛用于验证交易、地址生成以及智能合约的部署。通过深入了解ECDSA在以太坊中的应用，我们可以更好地理解以太坊的安全机制和保障用户资产的安全。

（1）交易验证和身份认证：以太坊中的交易是通过ECDSA签名进行验证的。每笔交易都需要发送者使用其私钥对交易内容进行签名，然后接收者可以使用发送者的公钥验证签名的有效性。这确保了交易的身份认证和完整性，防止了篡改和伪造交易。

（2）地址生成与管理：以太坊地址是由公钥生成的，并通过哈希函数转化为可识别的形式。这意味着用户可以使用其私钥生成一个唯一的地址，用于接收以太币或其他代币。ECDSA的安全性保证了地址生成过程的安全性，同时也确保了地址和私钥之间的一一对应关系。

（3）智能合约的安全性：智能合约是以太坊上的自动化合约，也可以使用ECDSA进行验证和授权。当智能合约需要执行涉及安全权限的操作时，可以使用ECDSA签名来确保只有经过授权的用户才能执行这些操作，从而增强智能合约的安全性。

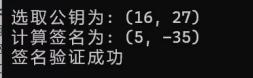
（4）链上身份认证：ECDSA还可以用于在以太坊上实现分布式的身份认证系统。用户可以使用其私钥对一些身份信息进行签名，从而证明自己的身份。这在去中心化身份验证和数字身份领域具有潜在的应用。

（5）多重签名钱包：以太坊支持多重签名钱包，这意味着需要多个私钥的持有者共同签署一笔交易才能使其有效。这可以通过ECDSA实现，确保了资金的更高安全性，尤其在需要多方参与决策的情况下。

（6）隐私保护：虽然ECDSA本身并不是专门用于隐私保护的技术，但它可以与其他隐私技术结合使用，如环签名、零知识证明等，从而在以太坊交易中增加一定程度的隐私保护。

实现方式：C

结果：在自己电脑上 CPU：11 代 i7



分工：自己独立完成

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

说明：

此项目是实现遵循RFC6979的sm2。

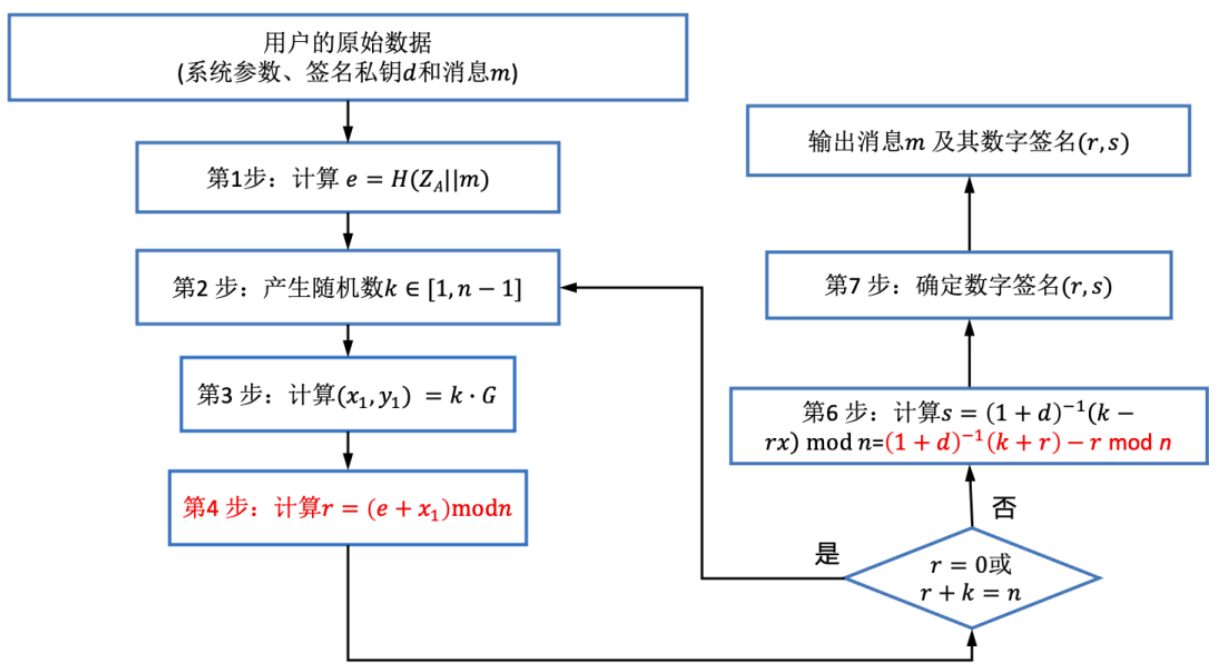
在椭圆曲线里面k值(用于签名)是要严格保密的。如果使用相同的值k在不同的消息（交易）上生成两个签名，那么任何人都可以计算出签名私钥。在签名算法中重用相同的k值的会导致私钥的暴露。k值要保证两点：保密和唯一。

RFC6979是一种用“确定性”方式来产生k值的方法，保证了“保密”且“唯一”。在本项目中我用以下公式来产生k值：

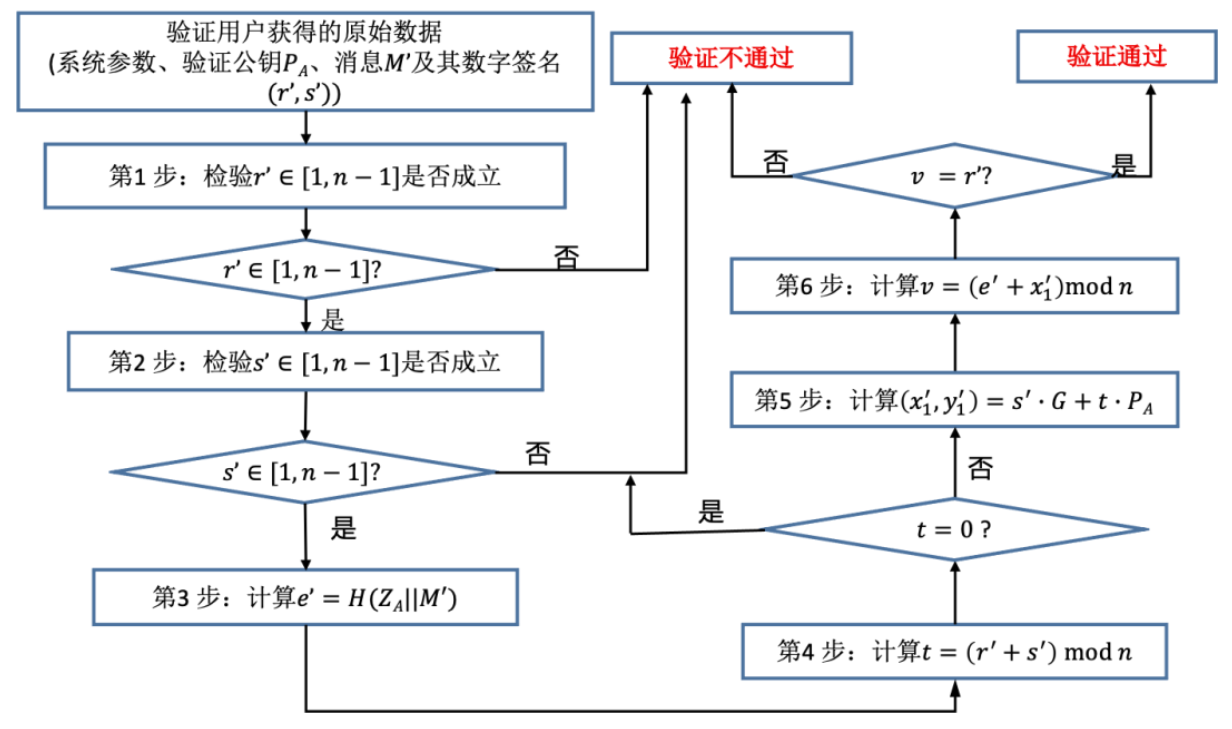
k = SHA256(d + HASH(m))

参数里有私钥d，就保证了“保密”，再加上消息m，保证了“唯一”，HASH用SM3。只要SHA256是安全的，此算法就是安全的。

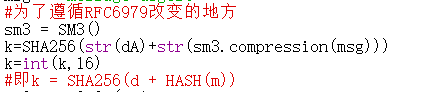
SM2数字签名算法签名过程示意图：



SM2数字签名算法验证过程示意图：

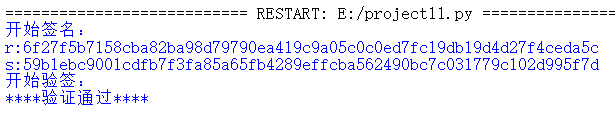


# 在本项目中我在已封装的sm2签名与验签算法python实现代码的基础上，改变了k值获得的方法，使其符合RFC6979。sm2签名与验签算法python实现代码的原代码可从https://blog.csdn.net/weixin\_43261410/article/details/126991494查看，对k做出的改变如下：



实现方式：python

结果：在自己电脑上CPU：11代i7

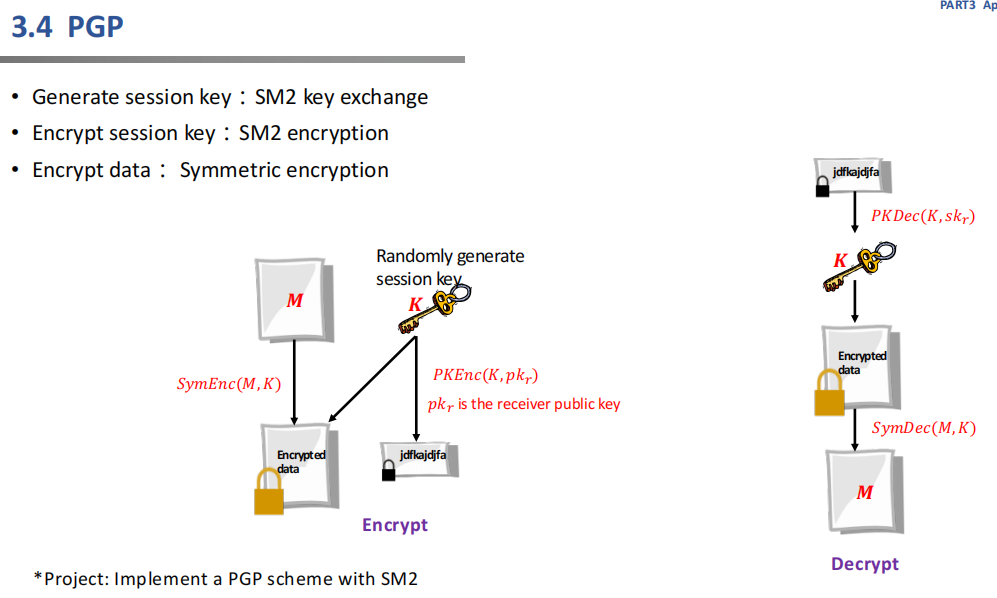


分工：自己独立完成

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

说明：

此项目是利用SM2实现PGP。

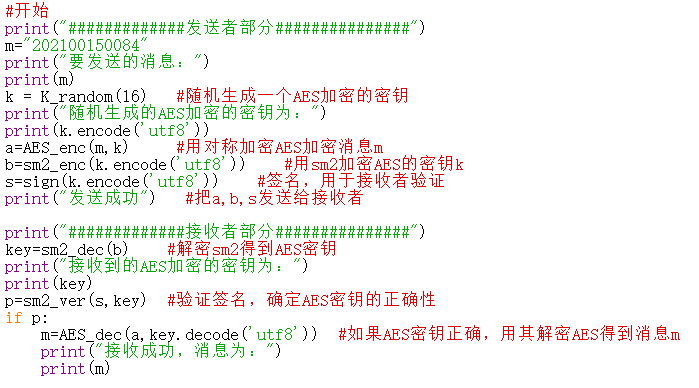


首先发送者和接收者分别生成SM2的公钥和私钥，并进行密钥交换。

发送者基本步骤如下：

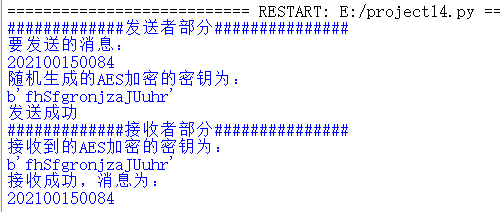
1. 随机生成临时会话密钥K（对称加密的密钥），本项目对称加密使用的为AES。
2. 用K对要发送的消息进行AES加密得到a。
3. 用SM2的公钥加密K得到b。
4. 对K进行签名，便于接收者验证得到s。
5. 把a、b、s发给接收者。  
    接收者基本步骤如下：
6. 用SM2私钥解密b得到K。
7. 验证签名。
8. 用K进行AES解密得到发送者发的消息m。

为了本次项目实现方便，把发送者、接收者写在了一个文件里，重要代码如下：



实现方式：python

效果：在自己电脑上CPU：11代i7



分工：自己独立完成

# Project15: implement sm2 2P sign with real network communication

## 算法简介：

为了实现SM2曲线上的2P签名并进行真实的网络通信，我们需要对SM2算法进行一些增强。步骤之一包括：

调整所使用的椭圆曲线参数，将SM2的参数设置为你选择的SM2曲线参数。

利用Python的socket模块来实现网络通信。需要编写适当的代码来创建UDP客户端或服务器，从而能够发送和接收数据。请注意，发送和接收数据时，需要对数据进行适当的编码和解码，例如：

1. client.sendto(x.encode('utf-8'), address)

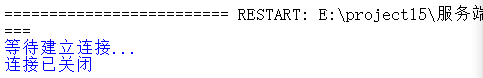
2. client.sendto(y.encode('utf-8'), address)

同时修改代码，以便可以根据SM2的签名结果和已知的参数来推导SM2公钥。

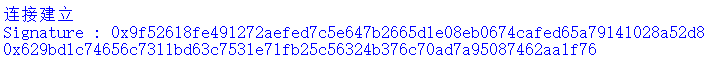
实现方式：python

## 结果：

服务端：



客户端：

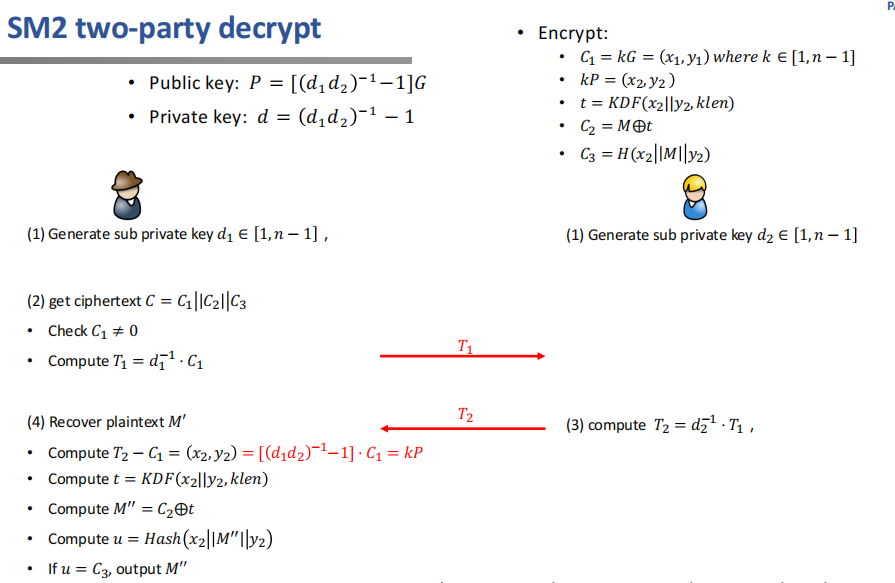


分工：自己独立完成

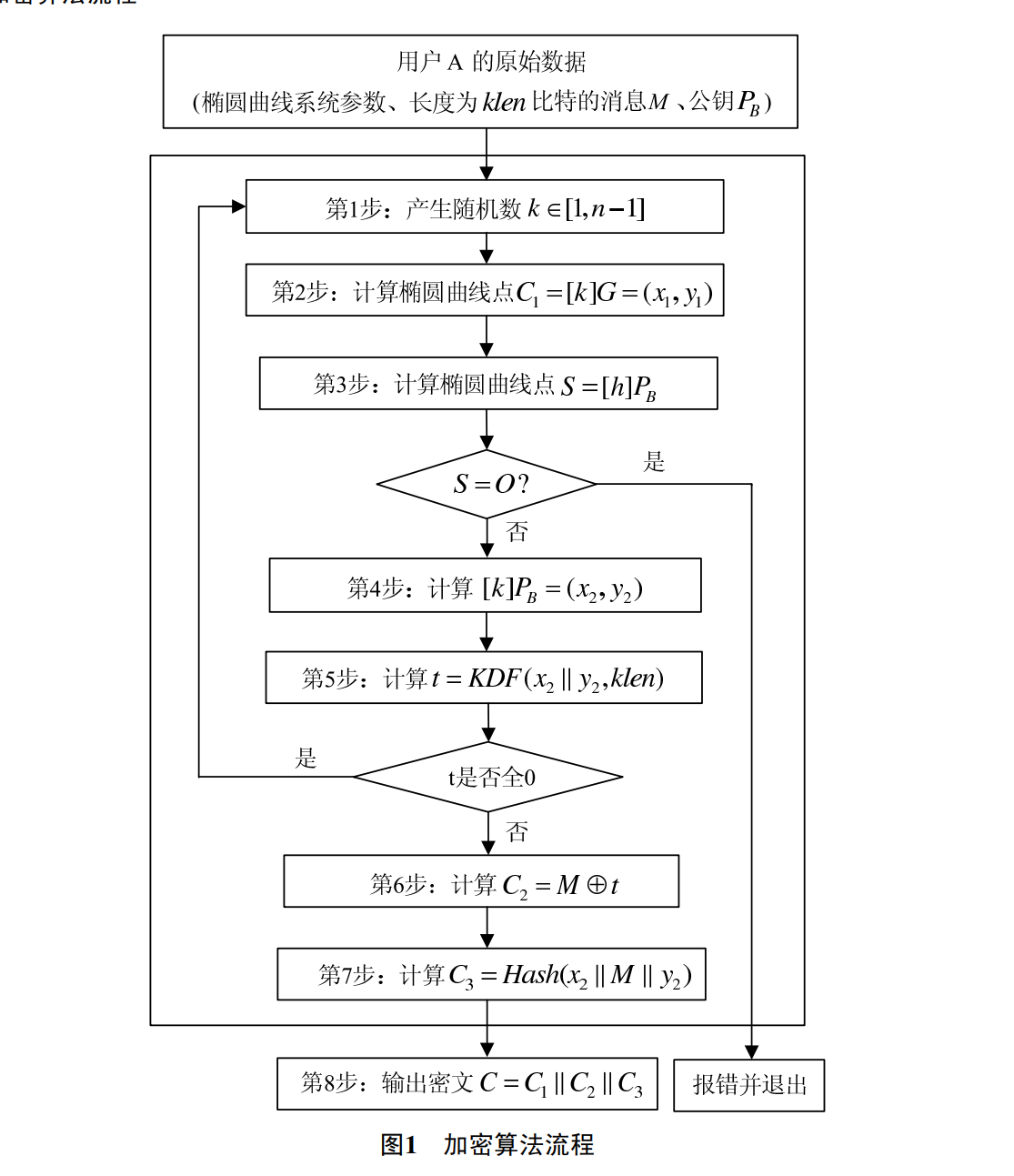
# **\*Project16: implement sm2 2P decrypt with real network communication**

说明：

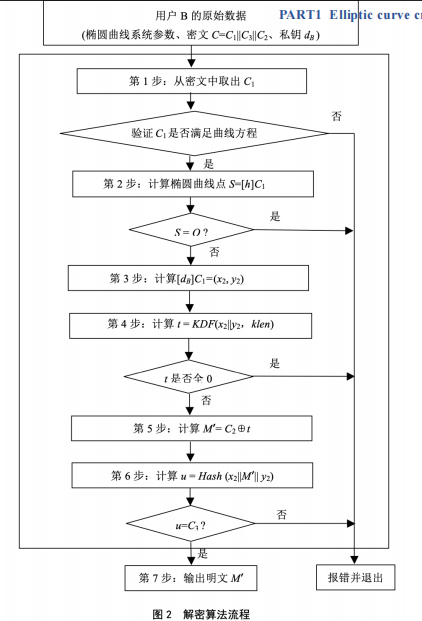
本项目要求用实际网络通信实现sm2解密，示意图如下：



用户A的加密如下：



用户B的解密如下：



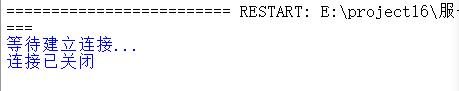
实际网络通信可以用以下等代码实现：

1. client.sendto(x.encode('utf-8'), address)

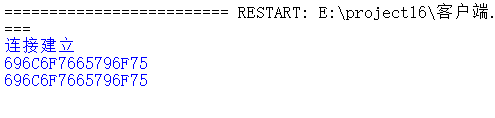
2. client.sendto(y.encode('utf-8'), address)

## 结果：

服务端：



客户端：



实现方式：python

分工：自己独立完成

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

## 一、密码的用途和储存方法

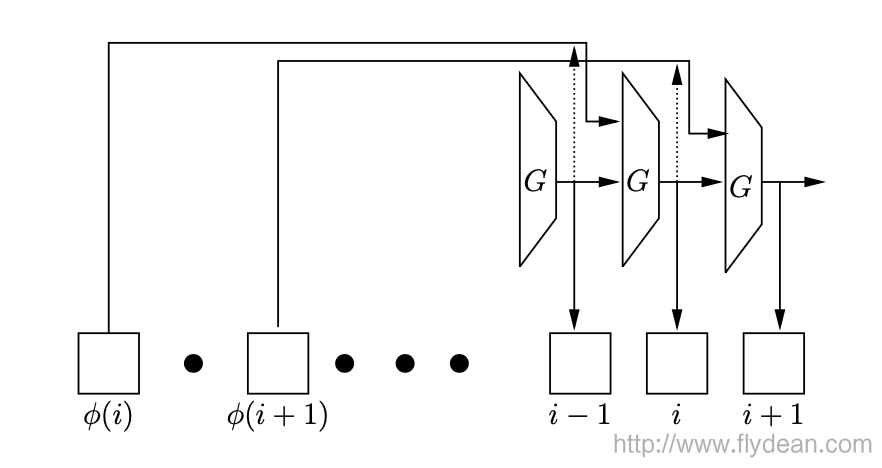
Web服务主要通过使用密码进行用户认证，这些密码往往以哈希形式存在于数据库内。然而数据库如果被盗取，经历字典攻击后的数据库可以被轻易破解，原因在于这些密码的熵值较底，有许多用户会在不同的系统中使用相同的密码。

解决方案是在密码哈希过程中加入盐（Salt）。尽管加盐可以解决许多问题，但无法挡住暴力攻击，特别是当哈希计算可以借助涉及GPU、FPGA、ASIC等定制硬件进行低成本计算的情况下。密码和盐同时被盗取时，破解的成本将会更低。

解决以上问题的关键在于，当哈希方法需要使用大块内存进行计算时，GPU和ASIC等硬件就无法发挥作用。因此，开发了需要大量内存计算的哈希函数（Memory-hard Hash Function）。

**火狐浏览器使用PBKDF2作为其哈希函数，而谷歌浏览器使用Argon2。**

## 二、Argon2d的使用



谷歌浏览器中所使用的Argon2d进行数据相关的内存访问，使其在加密数字货币以及工作量证明的应用上表现优良，同时抵抗侧信道定时攻击。Argon2i使用数据无关的内存访问，是密码哈希的首选方法。Argon2id则结合了Argon2i和Argon2d的优点，旨在提供侧信道攻击保护，同时节省暴力开销。

## 三、Argon2的安全性分析

对Argon2的分析表明该协议展示出两个主要的安全属性：

“被动”攻击者即使获取了服务器存储的数据库内容，只能做两件事：1. 学习到密码的哈希值kA。2. 对密码进行依赖硬性哈希计算的暴力攻击。

“主动”攻击者，即那些窃取TLS连接信息或者干扰正在运行的密钥服务器的攻击者，可以做的事包括：1. 学习到密码的哈希值kA。2. 控制账户，也就是说可以生产校验。3. 对密码和哈希值kB执行简单暴力攻击，即对每个猜测的密码都需要进行1000轮PBKDF。

尽管Argon2相比于基于SRP的协议稍显不足，但实际上它的安全性强于大部分业界实践，并且更适合客户端实现。一段长期服务器数据可以阻止简单的字典攻击，即使攻击者获取了所有服务器数据库内容，也需要对每个猜测进行完整的scrypt延伸计算。

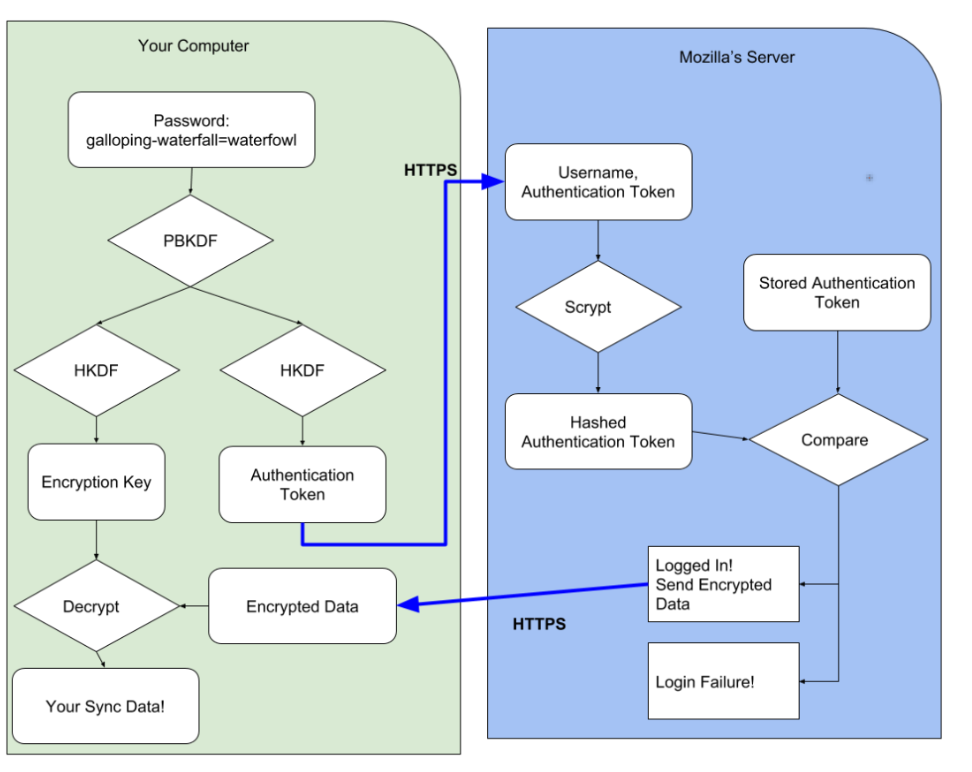
被动攻击者可以通过两个值预测密码猜测。其中一种是“verifyHash”，源自完整的基于scrypt的扩展输出。另一种则是对一些B类加密数据进行测试密码，这同样受到scrypt过程的保护：对于每个密码，攻击者都需要执行完整的计算得出kB，然后尝试解密一部分数据，并检查其HMAC是否一致。

HMAC检查通过表示候选密码正确。攻击者可以利用预加密响应作为预测机，但服务器明确不会保留加密它的响应。所以，keyFetchToken是随机生成的，独立于用户的密码，因此加密的数据并不能帮助测试密码猜测。

## 四、PBKDF2的使用

PBKDF2实际上就是将伪散列函数PRF（pseudorandom function）应用到输入的密码、salt中，生成一个散列值，然后将这个散列值作为一个加密key，应用到后续的加密过程中，以此类推，将这个过程重复很多次，从而增加了密码破解的难度，这个过程也被称为是密码加强。

我们看一个标准的PBKDF2工作的流程图：



从图中可以看到，初始的密码跟salt经过PRF的操作生成了一个key，然后这个key作为下一次加密的输入和密码再次经过PRF操作，生成了后续的key，这样重复很多次，生成的key再做异或操作，生成了最终的T，然后把这些最终生成的T合并，生成最终的密码。

根据2000年的建议，一般来说这个遍历次数要达到1000次以上，才算是安全的。当然这个次数也会随着CPU计算能力的加强发生变化。这个次数可以根据安全性的要求自行调整。

有了遍历之后，为什么还需要加上salt呢？加上salt是为了防止对密码进行彩虹表攻击。也就是说攻击者不能预选计算好特定密码的hash值，因为不能提前预测，所以安全性得以提高。标准salt的长度推荐是64bits，美国国家标准与技术研究所推荐的salt长度是128 bits。

## 五、参考文献

> Password Hashing: Scrypt, Bcrypt and ARGON2

> <https://github.com/mozilla/fxa-auth-server/wiki/onepw-protocol#vs-old-sync>

> [PBKDF2加密\_pbkdf2是对称加密\_柏修的博客-CSDN博客](https://blog.csdn.net/cfl20121314/article/details/46994393)

## 六、附录：

1. public static final int HASH\_MILLIS = 1231;

2. public static final String ALGORITHM = "asfdasdfdfsafs";

3. public static final int ITERATION\_COUNT = 123123;

4. public static final int KEY\_SIZE = 123;

5. public static final int SALT\_LENGTH = 123;

6.

7. public static String encryptPassword(String salt,String password) throws Exception{

8. byte[] saltByte = Base64.decodeBase64(salt.getBytes());

9. byte[] hash = PasswordsUtils.hashPassword(password.toCharArray(), saltByte);

10. String pwd\_hash\_str = new String(Base64.encodeBase64(hash));

11. return pwd\_hash\_str;

12. }

13.

14. public static byte[] hashPassword(char[] password, byte[] salt)

15. throws GeneralSecurityException {

16. return hashPassword(password, salt, ITERATION\_COUNT, KEY\_SIZE);

17. }

18.

19. public static byte[] hashPassword(char[] password, byte[] salt,

20. int iterationCount, int keySize) throws GeneralSecurityException {

21. try {

22. PBEKeySpec spec = new PBEKeySpec(password, salt, iterationCount, keySize);

23. SecretKeyFactory factory = SecretKeyFactory.getInstance(ALGORITHM);

24. return factory.generateSecret(spec).getEncoded();

25. } catch (IllegalArgumentException e) {

26. throw new GeneralSecurityException("key size " + keySize, e);

27. }

28. }

29.

30. public static boolean matches(char[] password, byte[] passwordHash, byte[] salt)

31. throws GeneralSecurityException {

32. return matches(password, passwordHash, salt, ITERATION\_COUNT, KEY\_SIZE);

33. }

34.

35. public static boolean matches(char[] password, byte[] passwordHash, byte[] salt,

36. int iterationCount, int keySize) throws GeneralSecurityException {

37. return Arrays.equals(passwordHash, hashPassword(password, salt,

38. iterationCount, keySize));

39. }

40.

41.

42. public static byte[] nextSalt() {

43. byte[] salt = new byte[SALT\_LENGTH];

44. SecureRandom sr = new SecureRandom();

45. sr.nextBytes(salt);

46. return salt;

47. }

48.

**\*Project18: send a tx on Bitcoin testnet, and parse the tx data down to every bit, better write script yourself**

说明：

比特币测试币是一种用于模拟比特币网络交易的虚拟货币，用于开发和测试新的应用程序和功能。它们没有真实价值，主要在测试网络中使用，这里我们也使用这种测试币进行测试。比特币测试币分为Testnet和Regtest两种类型。Testnet是由比特币社区维护的公共测试网络，可通过水龙头获取。

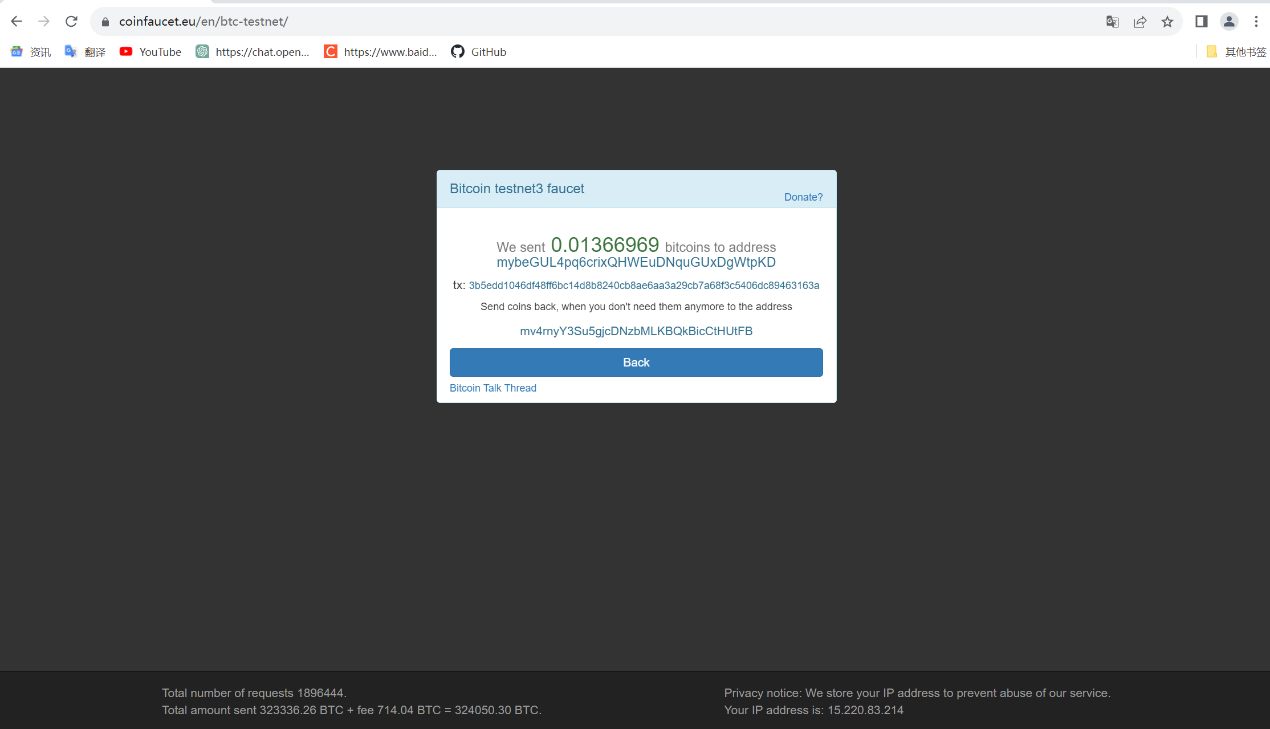
## 实现方式

在比特币测试地址获取网站<https://www.bitaddress.org/?testnet=true>，获得测试地址

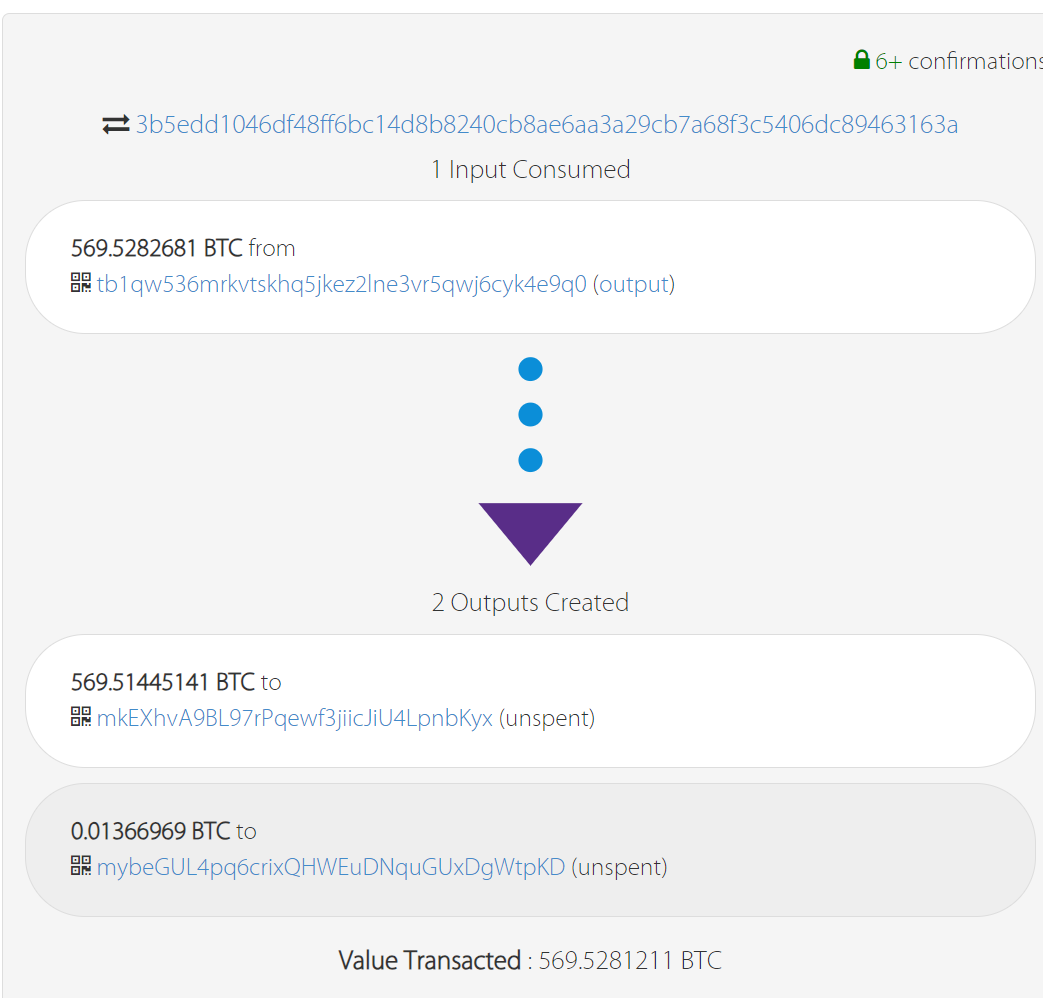


[mybeGUL4pq6crixQHWEuDNquGUxDgWtpKD](https://live.blockcypher.com/btc-testnet/address/mybeGUL4pq6crixQHWEuDNquGUxDgWtpKD/) 是我的地址

登录Faucets网站，输入我的地址获取一定数量的免费测试用币



最后查询本次交易信息



写脚本来爬取网页，解析tx信息，可以得到更多交易细节。

## 运行结果

脚本运行结果如下：（更多内容见result.md文件）



在这里可以看到交易的地址、交易的哈希值、交易的大小、交易的生效时间，交易的输入输出等

实现方式：python

分工：自己独立完成

**\*Project19: forge a signature to pretend that you are Satoshi**

代码说明：

实现方法：

实现对于中本聪的伪造即要实现对于ECDSA签名的伪造

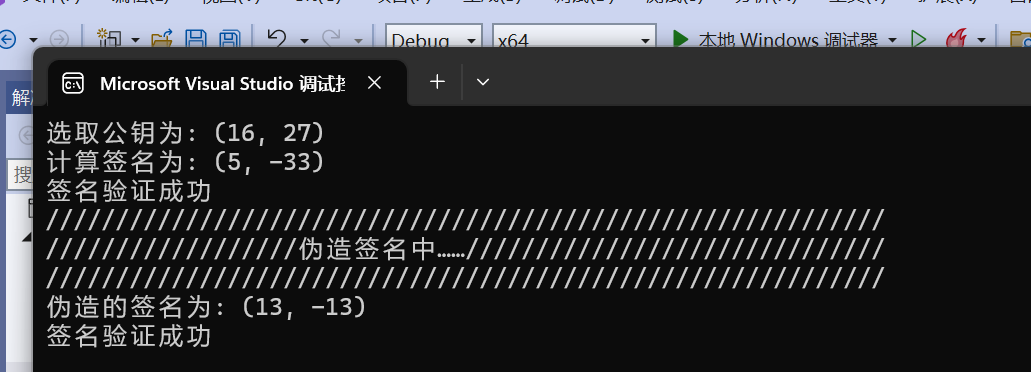
可以在已知公钥P的前提下，通过构造e来重组合法签名。

第一步：选取随机数u,v ，计算R= uG+vP

第二步：计算r=R.x、s = r\*b^-1 、e = r\*v^-1. 其中e=H(m’)

(r', s')即对消息m'的合法签名

运行结果：

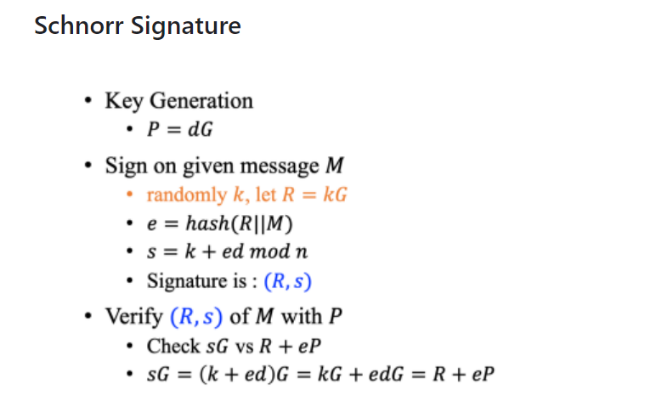


实现方式：C

分工：自己独立完成

**\*Project21: Schnorr Bacth**

说明：



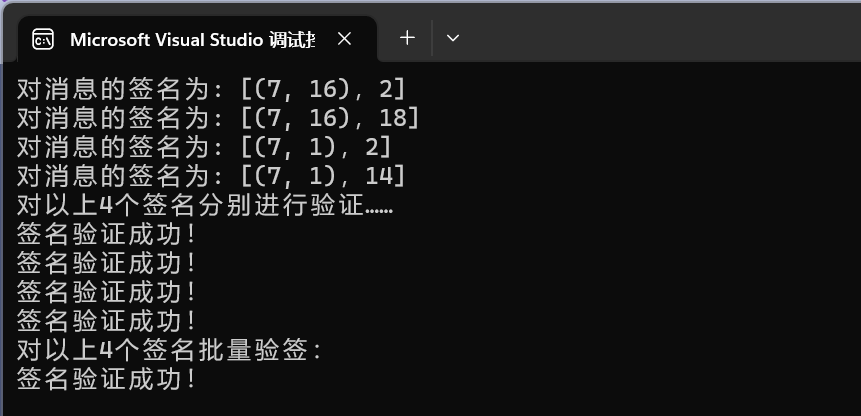
根据以上Schnorr签名方案实现基础的Schnorr签名以及认证。实现Schnorr签名的批量验签。

批量验证的原理：

给定有 n 个公钥、消息和签名元组 (Pi , mi , (Ri , si))，验证者生成 n 个随机数 a1, … , an ，来计算 n 个挑战哈希值 ei = H(Pi , Ri , mi) ，然后检查：

(a1s1+a2s2+…+ansn)∗G?=a1∗R1+a2∗R2+…+an∗Rn+(a1e1)∗P1+(a2e2)∗P2+…+(anen)∗Pn，如果相等则通过验证。

运行结果：



实现方法：

本项目使用C语言编程实现

分工：自己独立完成

## \*Project22: research report on MPT

## 介绍

Merkle Patricia Tree (MPT)，也被称为 Merkle Patricia Trie，是一种结合了前缀树（Trie）和默克尔树（Merkle Tree）的数据结构，它在以太坊中被用来组织和管理账户数据以及生成交易集的重要数据结构哈希。以下是对 MPT 的功能进行解释：

1.存储任意长度的 key-value 键值数据：MPT 允许存储任意长度的数据，其中键值对是通过 key 进行关联的。这些键值对可以是账户的状态数据、交易信息等。

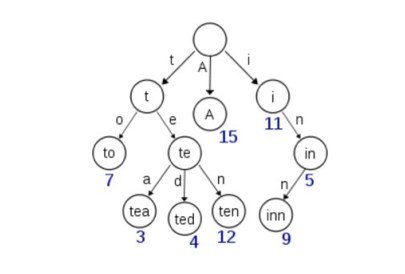
2．提供快速计算维护数据集哈希的机制：MPT 的一个关键特点是可以高效地计算数据集的哈希值。这使得在以太坊这样的去中心化区块链系统中，可以快速验证整个数据集的完整性。

3.提供快速状态的滚动机制：MPT 允许对数据集进行状态的更改和滚动更新。在区块链中，当有新的交易发生时，只需要更新部分数据而不是整个数据集，这样可以显著提高处理效率。

4.提供一种叫默克尔证明的方法，轻量级扩展节点，简单的支付验证：MPT 基于默克尔证明（Merkle Proof）的概念，允许在不访问所有数据的情况下验证特定数据是否在树中。这使得轻客户端可以快速验证交易和账户状态，而无需完整地同步整个区块链。

MPT结合了前缀树和Merkel Tree两种树结构的特点和优点，下面先来了解这两种树。

## 前缀树

前缀树是一种树形数据结构，特别适合存储字符串集合，其中共享相同前缀的字符串会共享相同的前缀树节点，从而节省存储空间。在 MPT 中，前缀树的特点用于存储键值数据，以便快速查找和定位特定的账户或交易。

前缀树在某些方面具有优势，但也有一些明显的缺点。

优点：

高效查询公共前缀键数据：对于具有公共前缀的键值数据，前缀树能够高效地进行查询，因为它可以共享相同前缀的节点，减少了搜索的范围。

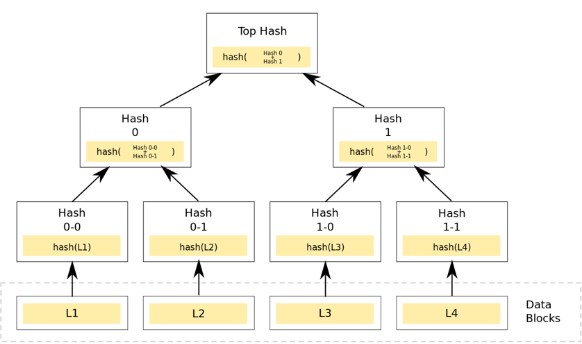
避免哈希冲突：前缀树不会像哈希表一样出现哈希冲突问题，因为数据的存储是基于键的前缀而不是哈希值。

缺点：

直接查找效率较低：在最坏的情况下，当要查找的前缀为空字符串或与树中大多数键值数据没有公共前缀时，前缀树的效率会下降，需要遍历整棵树。相比之下，哈希表的查找效率是常数时间O(1)。

可能造成空间浪费：当有一个节点的键值很长，而且与树中的其他节点没有公共前缀时，前缀树需要创建许多非叶子节点来构建路径，可能会造成存储空间的浪费。

## 默克尔树

默克尔树是一种用于验证数据完整性的二叉树结构，其中每个非叶节点的值是其子节点值的哈希。默克尔树在区块链中被广泛使用，特别是在比特币和以太坊等区块链中，用于验证区块数据的完整性和有效性。在 MPT 中，默克尔树的概念用于生成整个数据集的根哈希，从而便于快速验证数据的完整性。

在比特币网络中，默克尔树就是数据项的值，而数据项的值就是数据项的哈希值。自下而上构建。下面的例子中，先对L1-L4的四个单元数据进行哈希处理，然后将哈希值存储到对应的叶子节点中。这些节点分别是 Hash0-0、Hash0-1、Hash1-0、Hash1-1 将两个相邻节点的 hash 合并成一个字符串，然后计算该字符串的 hash，这两个节点就是父节点的 hash 值

如果两棵树的根哈希一致，那么两棵树的结构、节点的内容一定是相同的。如上图所示，一棵树有四个叶子节点，计算代表整棵树的hash需要经过7次计算。

同样，默克尔树也存在鲜明的特点：

快速重新哈希：默克尔树的结构使得在树中节点内容发生改变时，只需要对修改后的节点重新计算哈希值，就能快速得到新的根哈希，用来表示整棵树的状态。这样的特性在区块链中非常有用，因为区块链数据是不可变的，当有新的交易添加到一个区块时，只需对受影响的节点重新计算哈希值，而无需重新计算整个区块的哈希。

轻节点扩展：默克尔树使得在公链环境下扩展轻节点成为可能。轻节点只需要存储区块头数据，因此占用存储空间较小。通过轻节点，可以在不可信的公链环境下验证一笔交易是否包含在区块链账本中，而不需要下载完整的区块数据。这为在资源受限的终端设备上运行区块链客户端提供了解决方案。

存储空间开销大：默克尔树的一个主要缺点是它需要额外的存储空间来存储节点的哈希值。每个节点的哈希值都会占用一定的空间，因此在构建大型的默克尔树时，存储空间开销会相对较大。不过在实际应用中，这个缺点往往可以被接受，因为默克尔树在区块链和其他一些领域中的优势往往能够弥补这个缺点。

## 节点分类

前缀树虽然可以起到维护键值数据的目的，但是它有一个非常明显的局限性。无论是查询操作，还是增减数据，不仅效率低下，而且存储空间占用严重。因此，在以太坊中，为MPT树添加了几种不同类型的树节点，以尽量压缩整体树的高度，降低操作的复杂度。

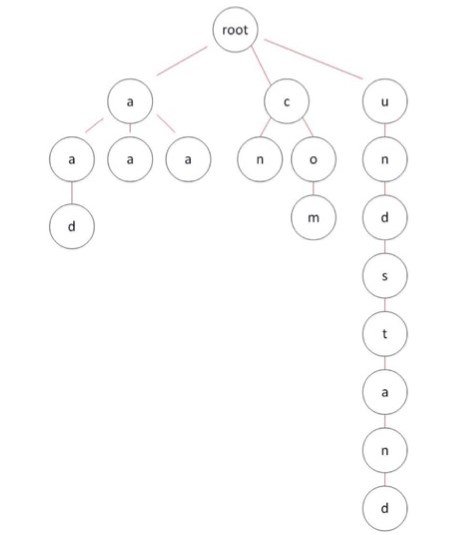
在MPT树中，树节点可以分为以下四类：

(1)空节点：其空心节点用于表示空字符串。

(2)分支节点：分支节点用于表示 MPT 树中具有多个子节点的所有非叶节点。分支节点的子列表中的最后一个元素用于存储其自身的内容。

(3)叶子节点&扩展节点

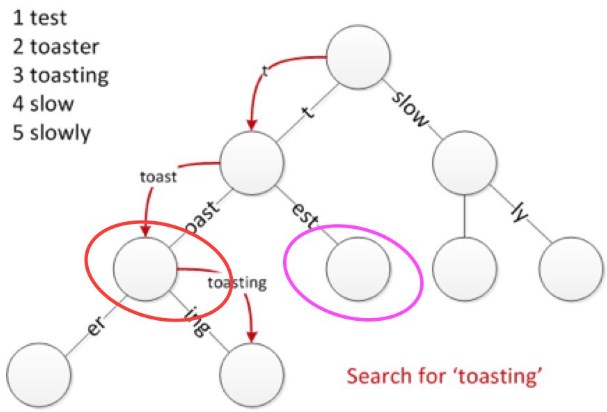
叶子节点与扩展节点的定义类似。前面提到，前缀树会存在存储空间的严重浪费，如下图：



图中右侧有一长串节点，其中大部分只是非叶子节点，用于构建一条路径，只存储该节点路径上的叶子。

在这种情况下，MPT树对此进行了优化：当MPT尝试插入节点时，插入过程发现当前不存在与节点Key具有相同前缀的路径。此时，MPT 将剩余的密钥存储在叶/扩展节点的 Key 字段中，并充当“快捷方式”。

例如我们将红线的节点称为node1，蓝线的节点称为node2。node1和node2共享路径前缀t，但是node1在插入时，树中并没有与oast路径共同的前缀，因此node1以oast为键，实现编码路径压缩。



这种方法有以下优点：

(1)提高节点查找效率，避免过多的磁盘访问；

(2)减少存储空间浪费，避免存储无用节点；

## 键值编码

在以太坊中，MPT树键值共有三种不同的编码方式，以满足不同场景的不同需求。

三种编码方式是：

(1) 原始编码（原生字符）：Raw Encoding 是原始的键值，不要做任何改变。这种按键编码方式，MPT对外接口提供默认的编码方式。

例如，键为“cat”，值为“dog”数据项，Raw 代码为 ['c', 'a', 't']，替换为 ASCII 表示为 [63, 61, 74]

（2）十六进制编码（scalablecoding hex）：在分支节点中，为了减少子节点的分支数量，需要对键码进行转换，将原来键的高低四位拆分成两个字节进行存储。这种转换的关键编码，即Hex编码。

（3） Hex-encoding the Prefix（十六进制前缀编码）：节点加载到内存中时，还需要通过额外的机制来区分节点的类型。因此etherbox提出了一种HP编码来区分数据库中存储的叶子/扩展节点的密钥。这两类节点持久化到数据库之前，都会将节点的key进行编码转换，即从Hex编码转换为HP编码。

## MPT安全性

MPT树可用于存储任意长度的键值数据项的内容。如果数据项key的长度不限制，当树中维护的数据量很大时，仍然会导致整棵树的深度变得越来越深，会造成以下影响：

（1）查询一个节点可能需要很多读IO次数，效率低下；

（2）Dos系统容易受到攻击，攻击者通过“构造”一棵很长路径的树来存储合约中的具体数据，然后调用SLOAD指令不断读取树中节点的内容，导致系统效率下降极度减少；

（3）所有的密钥实际上都是明文的存储形式；

为了解决这个问题，在MPT以太坊的平方中再封装一个数据项（sha3（key），value）。使得传入MPT接口的密钥是可以避免的固定长度（32字节），以便将密钥传递到MPT接口。但是，缺点是每棵树需要操作一次，以增加哈希计算量；同时，需要在数据库中存储额外的SHA3(密钥)与密钥的对应关系。