

**创新创业实践课汇总报告**

**网络空间安全学院（研究院）**

**网安一班张雨萌**

**2023年08月02日**

**目录**

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

**(yes)\*Project2: implement the Rho method of reduced SM3**

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

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

**(yes)\*Project5: Impl Merkle Tree following RFC6962**

**\*Project6: impl this protocol with actual network communication**

**\*Project7: Try to Implement this scheme**

**(yes)\*Project8: AES impl with ARM instruction**

**(yes)\*Project9: AES / SM4 software implementation**

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

**(yes)\*Project11: impl sm2 with RFC6979**

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

**(yes)\*Project13: Implement the above ECMH scheme**

**(yes)\*Project14: Implement a PGP scheme with SM2**

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

**(yes)\*Project16: implement sm2 2P decrypt with real network communication**

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

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

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

**(yes)\*Project20: ECMH PoC**

**(yes)\*Project21: Schnorr Bacth**

**(yes)\*Project22: research report on MPT**

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

# 一、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。

# 二、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。

#三、软件环境：Visual Studio 2022

硬件环境：处理器11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz 2.61 GHz（过后不再赘述）

#四、部分代码：

//查表寻找弱碰撞

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

if (outlist[j].substr(0, 4) == result.substr(0, 4)) {

cout << endl;

cout << "collision string input 1 :" + str << endl << endl;

cout << "collision hash value 1:" << endl;

cout << result.substr(0, 8) << " ";

cout << result.substr(8, 8) << " ";

cout << result.substr(16, 8) << " ";

cout << result.substr(24, 8) << " ";

cout << result.substr(32, 8) << " ";

cout << result.substr(40, 8) << " ";

cout << result.substr(48, 8) << " ";

cout << result.substr(56, 8) << " ";

cout << endl;

cout << "collision string input 2 :" + inlist[j] << endl << endl;

cout << "collision hash value 2:" << endl;

cout << outlist[j].substr(0, 8) << " ";

cout << outlist[j].substr(8, 8) << " ";

cout << outlist[j].substr(16, 8) << " ";

cout << outlist[j].substr(24, 8) << " ";

cout << outlist[j].substr(32, 8) << " ";

cout << outlist[j].substr(40, 8) << " ";

cout << outlist[j].substr(48, 8) << " ";

cout << outlist[j].substr(56, 8) << " ";

cout << endl << "finding num in all: " << i;

return;

}

}

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

#一、步骤

1. 随机生成1个消息

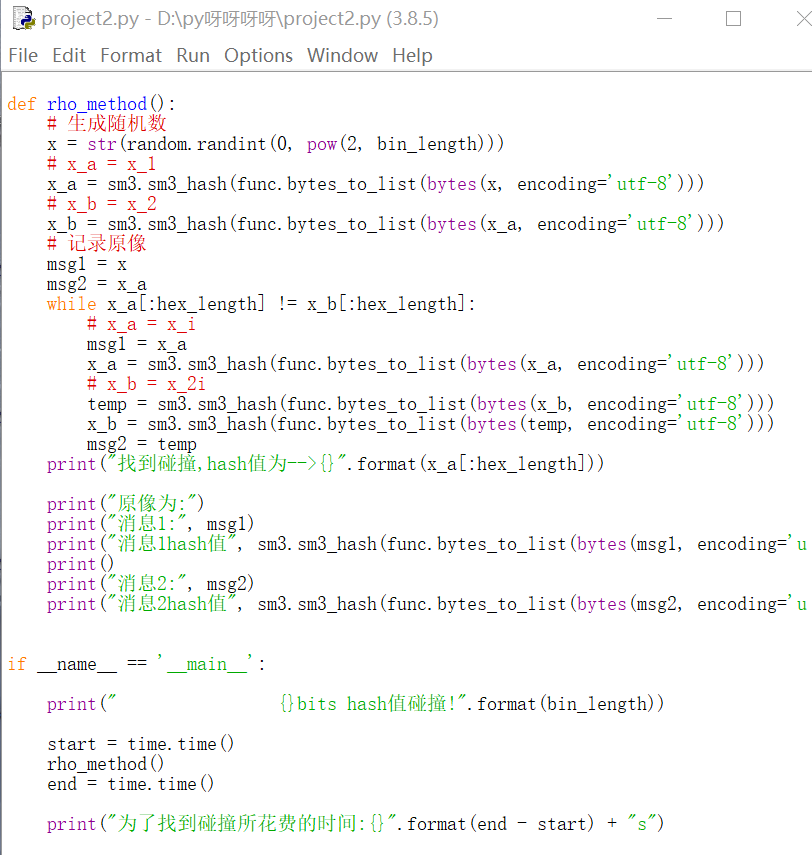
2. 由前面消息的hash作为新的消息，并求hash值

3. 判断是否构成环

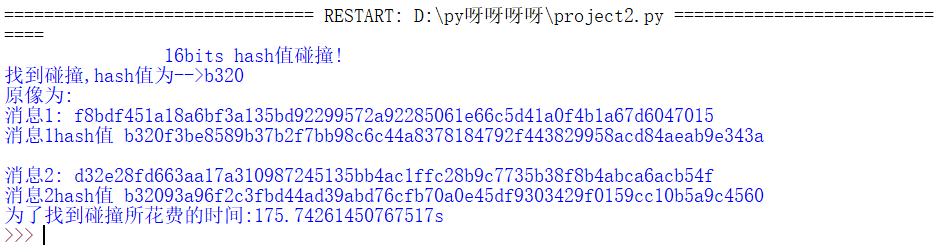
#二、攻击原理

Rho Method是一种随机化算法，每一个数都由前一个数决定，可以生成的数是有限的，所以会进入循环。 因为Rho Method是一种随机化算法，所以每次寻找碰撞的时间也不确定。

#三、代码摘要



#四、运行结果



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

#一、步骤:

1. 首先计算原消息(secret)的hash值

2. 在secret+padding之后附加一段消息,用原消息的hash值作为IV计算附加消息之后的hash值,得到消息扩展后的hash1

3. 用sm3加密伪造后的整体消息，得到hash2

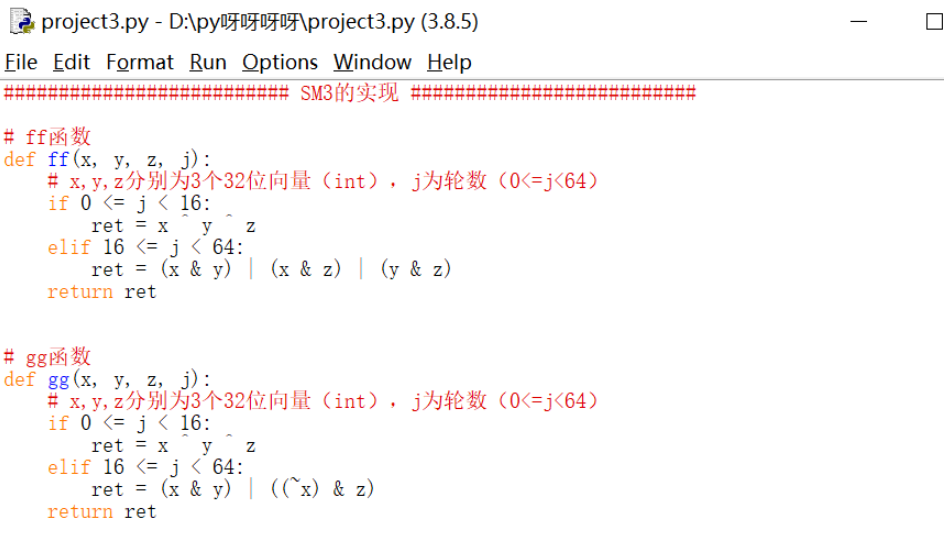
4. 验证hash1 与 hash2 是否相等

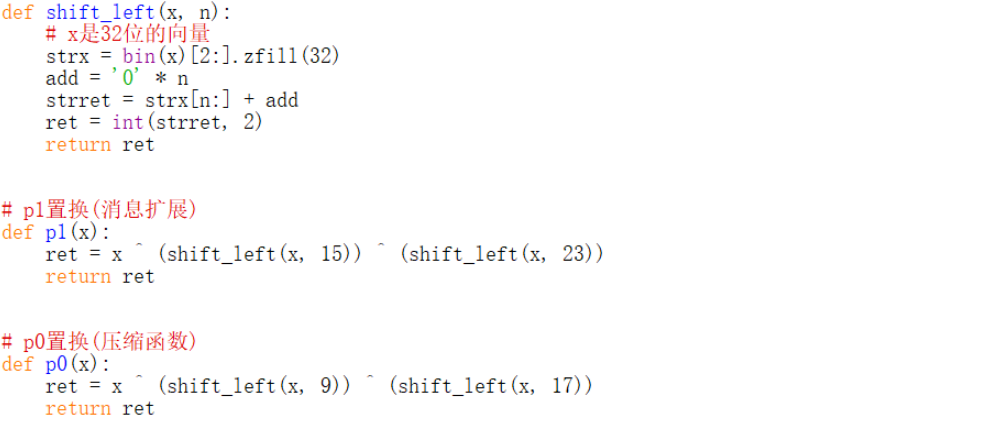
#二、攻击原理

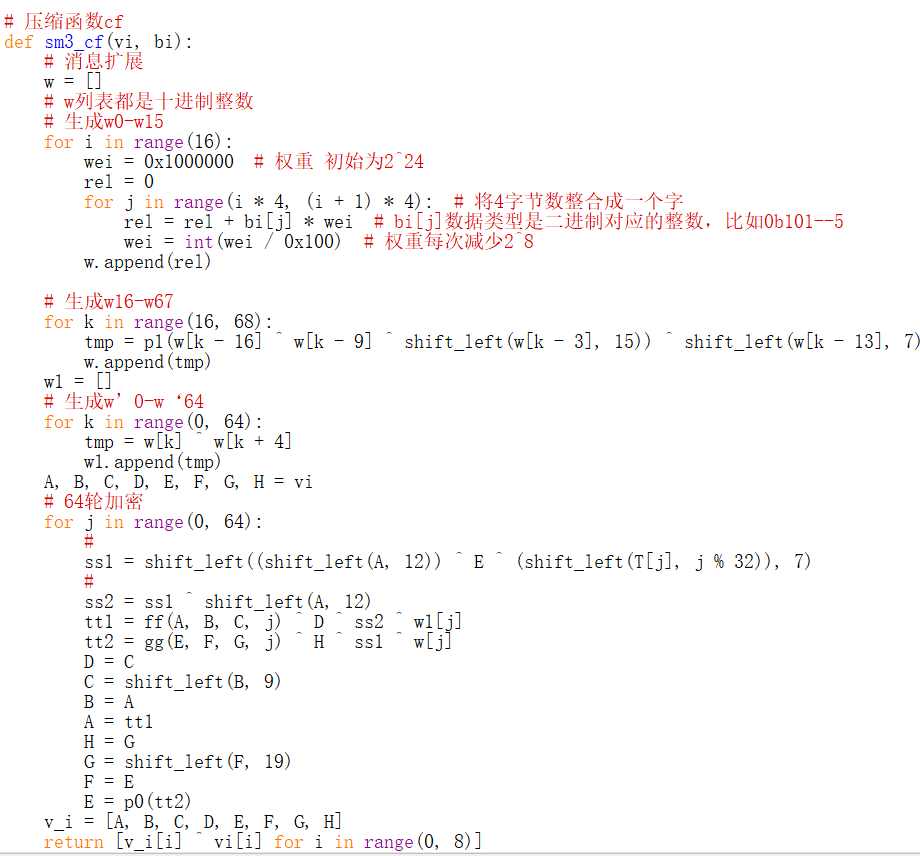
SM3的消息长度是64字节或者它的倍数，如果消息的长度不足则需要padding。在padding时，首先填充一个1，随后填充0，直到消息长度为56(或者再加整数倍的64)字节，最后8字节用来填充消息的长度。

在SM3函数计算时，首先对消息进行分组，每组64字节，每一次加密一组，并更新8个初始向量(初始值已经确定)，下一次用新向量去加密下一组，以此类推。我们可以利用这一特性去实现攻击。当我们得到第一次加密后的向量值时，再人为构造一组消息用于下一次加密，就可以在不知道secret的情况下得到合法的hash值。

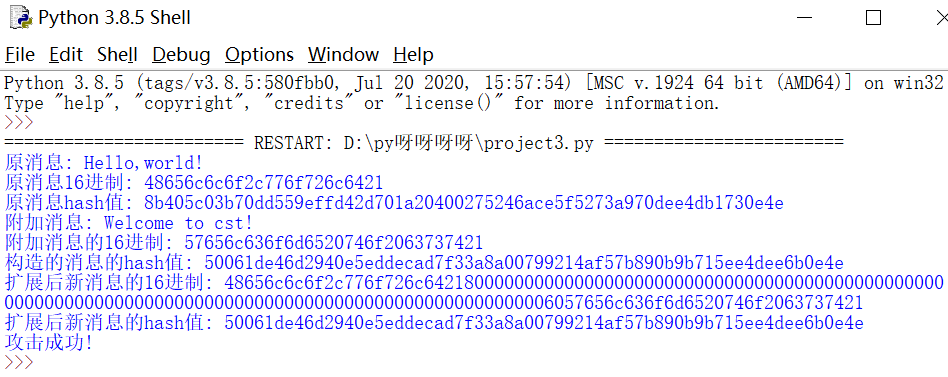
#三、代码摘要







#四、运行结果



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

#一、实验思路

1. 消息处理优化

(1) 初始化时，消息扩展只计算W0-W3四个32-bit字

(2) 在优化后的轮函数中首先计算W[i+4]，然后再计算W'[i]=W[i]^W[i+4]

经过这样的调整,去掉了字W'0,…,W'63,减少了字W0,…,W67和W'0,…,W'63的加载和存储次数,提高了消息扩展的速度

2. 每一轮压缩函数优化：

(1) 为了减少循环移位导致的不必要的赋值运算,可以将字的循环右移变更每轮输入字顺序的变动,且这个顺序变动会在4轮后还原

(2) 优化压缩函数的中间变量的生成流程,去除不必要的赋值,减少中间变量个数

(3) 利用上述调整以及消息扩展部分的调整可以将原来计算TT1、TT2、D和H的过程进行进一步简化

#二、部分代码：

/\*sm3主函数：包括消息填充、扩展、压缩步骤，返回最后的hash值\*/

std::vector<uint32\_t> SM3::sm3\_hash\_optimize()

{

caculT();//预计算压缩函数中要用到的T[i]<<i的值

std::vector<uint32\_t> hash\_result(32, 0);//（0，0，0，……，0）32个0，用来存放hash结果

unsigned char buffer[4] = { 0x01,0x03,0x4a,0x95 };//要处理的明文

unsigned char hash\_output[32];//存放结果的中间变量

SM3::SM3\_optimize(buffer, 4, hash\_output);//sm3的主函数，包括消息填充、扩展、压缩函数

hash\_result.assign(&hash\_output[0], &hash\_output[32]);//要返回一个vector变量所以把值赋到hash\_result

return hash\_result;

}

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

#一、代码说明

以sm3的散列模块为基础，用C++代码实现了默克尔树，可以手动输入数据，默克尔树叶节点会生成相应的哈希值，非叶节点会生成相应的联结哈希值。

可以运行test.cpp对该模块的构建默克尔树功能进行简单测试。

#二、运行指导

将源码clone到本地运行main函数即可运行。

#三、软件环境：Visual Studio 20

#五、部分代码

//默克尔树类

template<class T>

class MerkleTree {

private:

MerkleTreeNode<T>\* root;

public:

//构造函数

MerkleTree() {

root = NULL;

cout << "Now start to construct the MerkleTree" << endl;

CreateNode(root);

}

//析构函数

~MerkleTree() {

root = NULL;

}

//前序构建默克尔树

void CreateNode(MerkleTreeNode<T>\*& treeNode) {

cout << "Please enter value or '#' to stop:";

T value;

cin >> value;

treeNode = new MerkleTreeNode<T>(value);

if (value == "#") {

delete treeNode;

treeNode = NULL;

}

else {

CreateNode(treeNode->leftchild);

CreateNode(treeNode->rightchild);

//如果不是叶子结点，则对左右两个字结点做联合哈希

if (treeNode->leftchild != NULL && treeNode->rightchild != NULL) {

treeNode->data = iteration(padding(treeNode->leftchild->data + treeNode->rightchild->data));

}

}

}

//取根节点

MerkleTreeNode<T>\* getRoot() {

return root;

}

//递归前序遍历

void PreOrder(MerkleTreeNode<T>\* treeNode) {

if (treeNode != NULL) {

treeNode->VisitNode();

PreOrder(treeNode->getLeft());

PreOrder(treeNode->getRight());

}

}

//递归中序遍历

void InOrder(MerkleTreeNode<T>\* treeNode) {

if (treeNode != NULL) {

InOrder(treeNode->getLeft());

treeNode->VisitNode();

InOrder(treeNode->getRight());

}

}

//递归后序遍历

void PostOrder(MerkleTreeNode<T>\* treeNode) {

if (treeNode != NULL) {

PostOrder(treeNode->getLeft());

PostOrder(treeNode->getRight());

treeNode->VisitNode();

}

}

};

**\*Project6: impl this protocol with actual network communication**

用实际的网络通信实现该协议

**\*Project7: Try to Implement this scheme**

试图实施这个计划

**\*Project8: AES impl with ARM instruction**  
.section .data  
.align 4  
key:  
    .word 0x2B7E1516, 0x28AED2A6, 0xABF71588, 0x09CF4F3C  
text:  
    .word 0x3243F6A8, 0x885A308D, 0x313198A2, 0xE0370734  
.section .text  
.global \_start  
\_start:  
    @ Load key into registers  
    LDR r0, =key  
    LDR r1, [r0]       @ r1 = key[0]  
    LDR r2, [r0, #4]   @ r2 = key[1]  
    LDR r3, [r0, #8]   @ r3 = key[2]  
    LDR r4, [r0, #12]  @ r4 = key[3]  
    @ Load text into registers  
    LDR r0, =text  
    LDR r5, [r0]       @ r5 = text[0]  
    LDR r6, [r0, #4]   @ r6 = text[1]  
    LDR r7, [r0, #8]   @ r7 = text[2]  
    LDR r8, [r0, #12]  @ r8 = text[3]  
    @ Round 0 -- Add Round Key  
    ADD r5, r5, r1    @ r5 ^= r1  
    ADD r6, r6, r2    @ r6 ^= r2  
    ADD r7, r7, r3    @ r7 ^= r3  
    ADD r8, r8, r4    @ r8 ^= r4  
    mov r9, #10       @ r9 = 10 (number of rounds)  
    mov r10, #1       @ round counter  
round\_loop:  
    @ SubBytes  
    bl sub\_bytes  
    @ ShiftRows  
    bl shift\_rows  
    @ MixColumns  
    bl mix\_columns  
    @ AddRoundKey  
    bl add\_round\_key  
    @ Check if this is the last round  
    cmp r10, r9  
    beq end\_encryption  
    @ Update round counter  
    add r10, r10, #1  
    b round\_loop  
end\_encryption:  
    @ Store the result back  
    LDR r0, =text  
    STR r5, [r0]  
    STR r6, [r0, #4]  
    STR r7, [r0, #8]  
    STR r8, [r0, #12]  
    @ Terminate the program  
    MOV r7, #1  
    SWI 0x11  
sub\_bytes:  
    @ Implement SubBytes logic here  
    @ (omitted for brevity)  
    bx lr  
shift\_rows:  
    @ Implement ShiftRows logic here  
    @ (omitted for brevity)  
    bx lr  
mix\_columns:  
    @ Implement MixColumns logic here  
    @ (omitted for brevity)  
    bx lr  
add\_round\_key:  
    @ Implement AddRoundKey logic here  
    @ (omitted for brevity)  
    bx lr  
.end

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

Feistel 网络和 SP 网络是分组密码设计中的两种不同的整体结构类型，也是使用最 多的结构类型。AES 使用的是 SP 网络。

Feistel 网络（又称 Feistel 结构）把某一函数（通常称为 F 函数，又称轮函数）转化为一个置换。由于是 Horst Feistel 在设计分组密码时发明的，所以用了这位发明者的名字，并且因为 DES 的使用而流行。 当前世界上著名密码中还有 Twofish、 MARS、 FEAL、 GOST、LOKI、E2 、Blowfish、Camellia 和 RC6 等是采用 Feistel 结构的。 （Feistel 结 构还分平衡 Feistel 和非平衡 Feistel，这里只讨论平衡 Feistel。 ） 对于一个分组长度为 2n bits 的 r 轮 Feistel 型分组密码，其加密过程如下：

Step1：给定明文 X ，记 X = L0 R0 ，其中 L0 是 X 的左边 n bits， R0 是 X 的右边 n bits。

Step2：进行 r 轮循环运算。根据下列规则计算 Li Ri ， 1 ≤ i ≤ r ：

Li = Ri ?1 Ri = Li ?1 ⊕ F ( Ri ?1 , K i )，这里， F : F2n × F2N → F264 是轮函数， K 1 ,L K r 是由种子密钥 K 生成的子密钥， N 为 子密钥的长度。

Step3：输出密文 C = Rr Lr

在加密的最后一轮，为了使算法同时用于加密盒解密，略去“左右交换” 。

SP 网络（又称 SP 结构，Substitution & Permutation Net，SPN）是 Feistel 网络的一种推广，Rijndael、SAFER 和 SHARK 等著名密码算法都采用此结构。 在这种密码结构中，每轮的输入首先经过一个由子密钥控制的可逆函数 S，然后再 被作用于一个置换（或一个可逆的线性变换）P。S 被称为混淆层，主要起混淆的作用；P 被称为扩散层，主要起扩散的作用。

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

#一、实验内容

主要完成了以下4个task

1. Leaking k leads to leaking of d (k的泄露会导致泄露d)

2. Reusing k leads to leaking of d (对不同的消息使用相同的k进行签名会泄露d)

3. Two users, using k leads to leaking of d, that is they can deduce each other's d (两个不同的user使用相同的k,可以相互推测对方的私钥d)

4. Malleability, e.g. (r,s) and (r,-s) are both valid signatures (验证(r,s) and (r,-s)均为合法签名)

其中1和2还分别根据推测得到的d值进行了消息的伪造并成功通过验签

#二、文件内容

1. ECDSA.py:包含了ECDSA密钥生成、签名、验签算法

2. pitfalls.py:完成4个task

#三、攻击原理

1. Leaking k leads to leaking of d (k的泄露会导致泄露d)

由签名算法中s = k^(-1) \* (e + dr) mod n 推得 d = (s \* k - e) / r

2. Reusing k leads to leaking of d (对不同的消息使用相同的k进行签名会泄露d)

s1 = k^(-1) \* (e1 + dr) mod n

s2 = k^(-1) \* (e2 + dr) mod n

推得 s1/s2 = (e1 + dr) / (e2 + dr) 即 d = (（e1 - e2）\* s2 / (s1 - s2) - e2) \* r^(-1) mod n

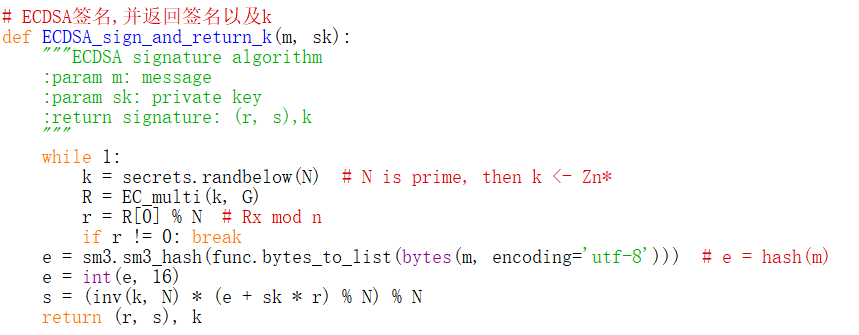
3. Two users, using k leads to leaking of d, that is they can deduce each other's d (两个不同的user使用相同的k,可以相互推测对方的私钥d)

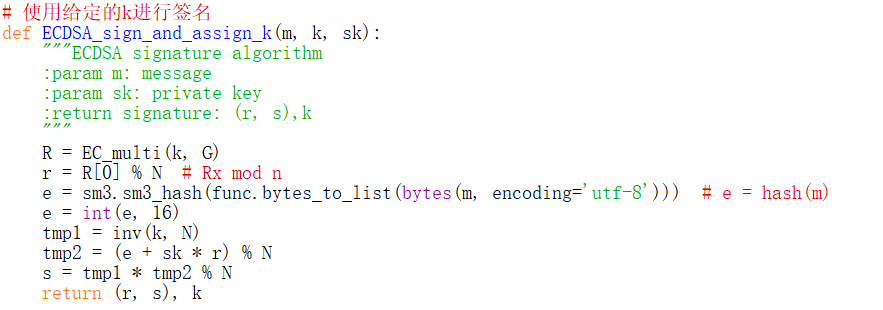
两个不同的user使用相同的k:此类情况下相当于leaking k,所以可以相互推测对方的私钥d

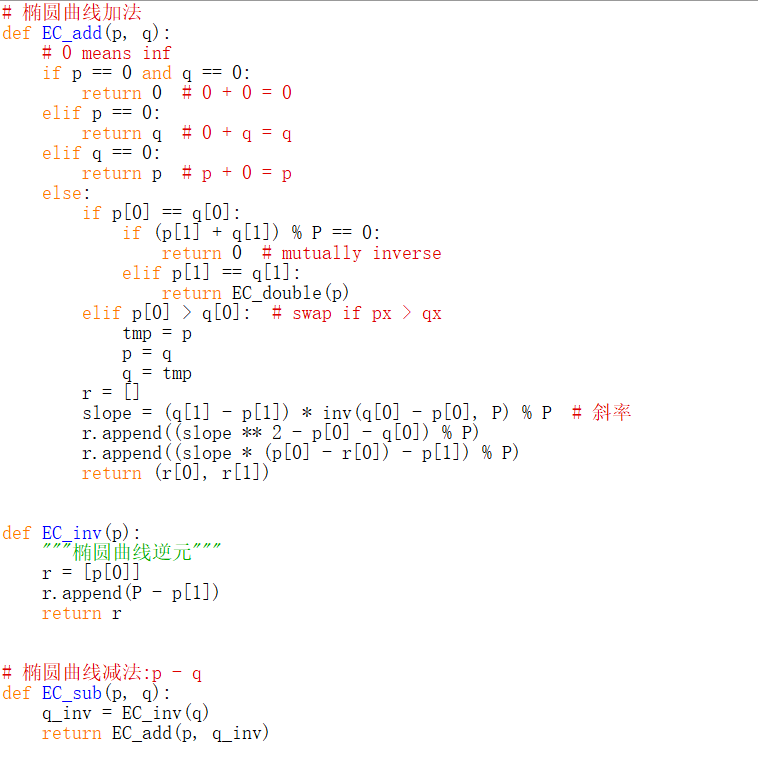
4. Malleability, e.g. (r,s) and (r,-s) are both valid signatures (验证(r,s) and (r,-s)均为合法签名)

首先生成签名(r, s),然后使用验签算法验证(r, -s)是否是合法签名

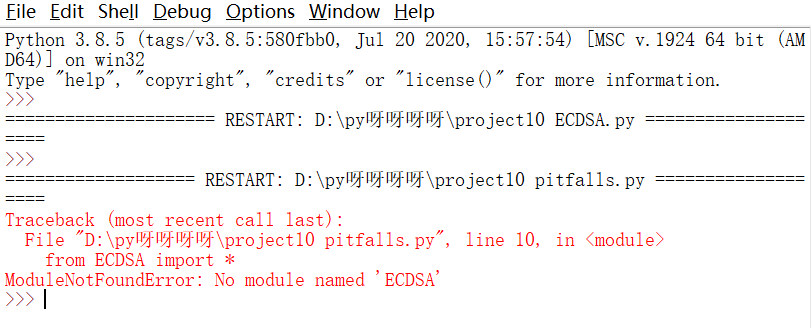
#四、代码摘要







#五、运行结果？



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

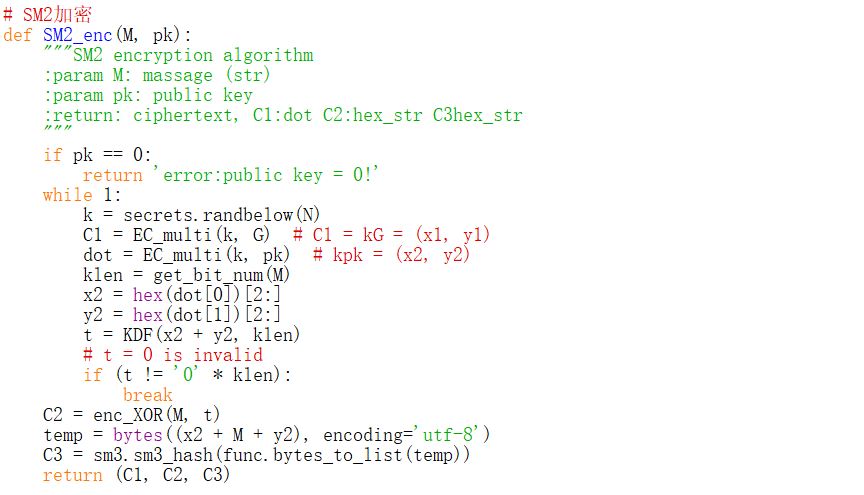
#一、实验内容

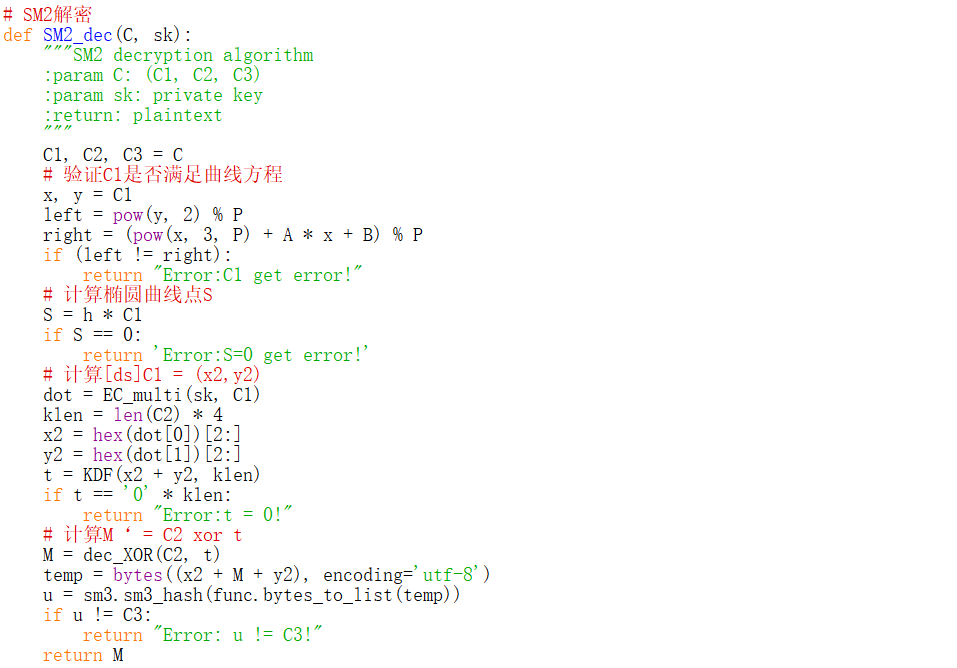
本code实现sm2的加解密算法以及签名验签算法

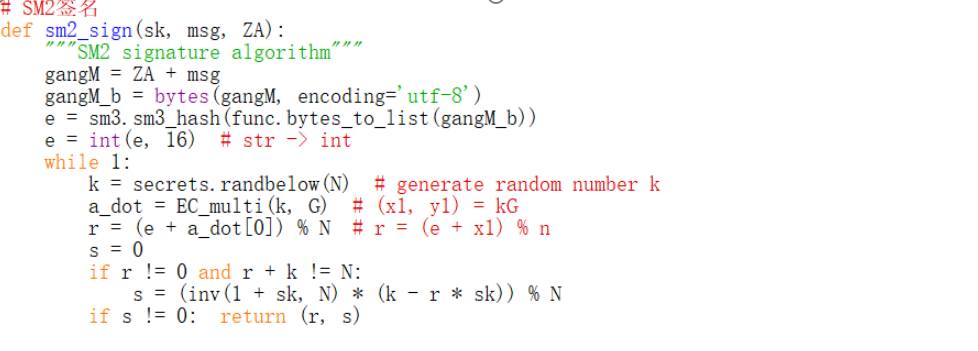
#二、实现原理

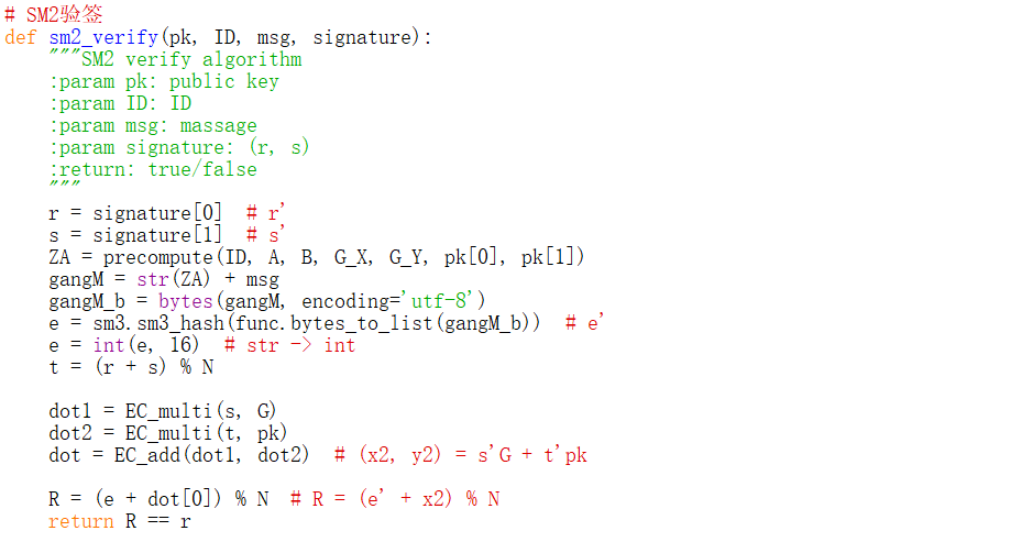
按照ppt所给出的SM2 Parameters、SM2 Signature Algorithm、SM2 Verify Algorithm、SM2 Encryption、SM2 Decryption进行实现

#三、代码摘要



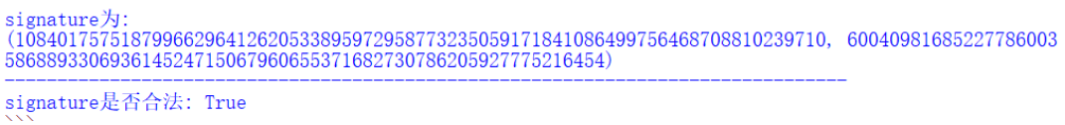






#四、运行结果



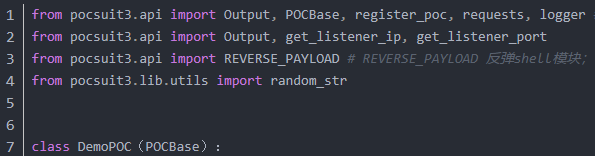


**\*Project12: verify the above pitfalls with proof-of-concept code**

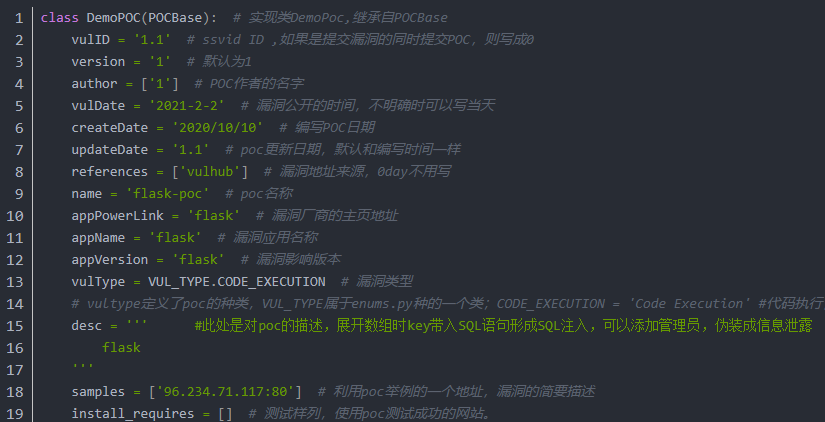
#一、poc脚本的编写步骤

1.首先新建一个.py文件，文件名应当符合命名格式

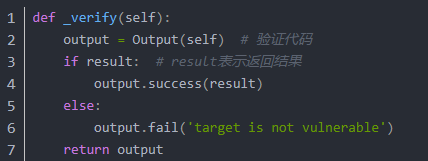
2.编写poc实现类DemoPOC，继承自POCBase类



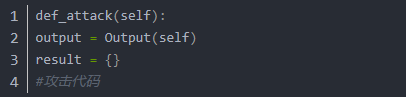
3.填写poc信息字段，需要认真填写所有字段的基本信息，规范信息以便于查找



4.编写验证模式，在\_verify方法中写入POC雁阵脚本



5.编写攻击模式。用\_attack（）函数中，写入EXP脚本，在攻击模式下可以对目标进行getshell，查询管理员账户密码等操作，定义它的方法与检测模式类似；



**\*Project13: Implement the above ECMH scheme**

#一、原始思路:将hash值作为椭圆曲线上的x点,根据椭圆曲线方程y^2 = x^3 + Ax + B来计算y值,但是会存在无解的情况,所以舍弃该做法

#二、本code将消息hash到椭圆曲线上的方法

(1) 首先计算消息的sm3 hash值

(2) 将sm3 hash值作为k值

(3) (x, y) = k \* G

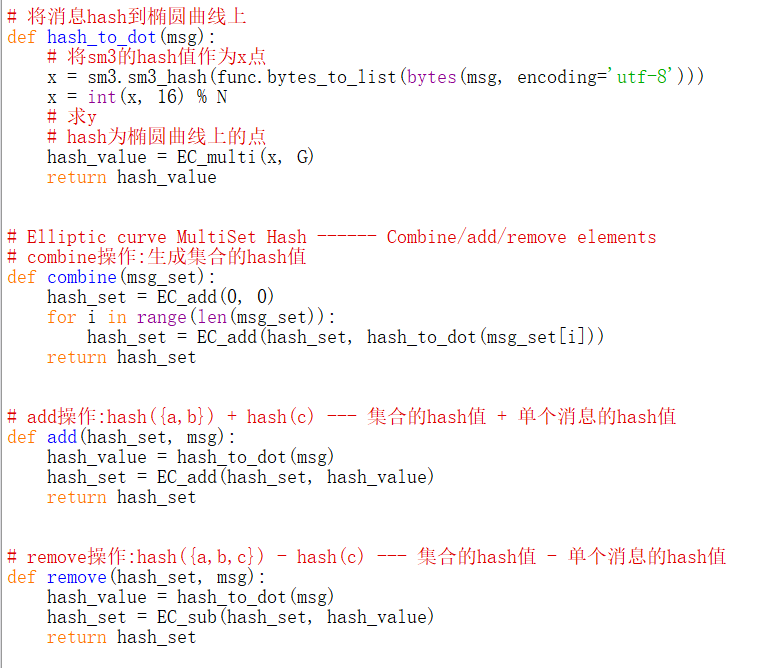
#三、本code完成的测试

(1) hash({a,b}) == hash({b,a})

(2) hash(a) + hash(b) == hash({a,b})

1. hash({a,b,c}) - hash(c) == hash({a,b})

#四、 代码摘要



#五、 运行结果



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

#一、PGP概念

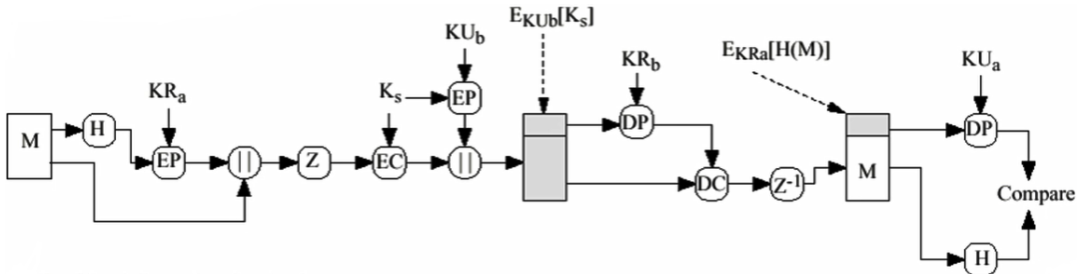
PGP（Pretty Good Privacy）加密，由一系列散列、数据压缩、对称密钥加密，以及公钥加密的算法组合而成，每个步骤支持几种算法。PGP支持消息认证和完整性检测：

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。

# #二、PGP加密流程



#三、举例说明

假设在一个数据M传输的过程中，发送者为A 、接受者为B ，公私钥对(pk, rk)，会话密钥s k

散列函数（哈希函数）用H 代替，公钥密码算法以R S A 为例，对称加密算法以A E S 为例:

1️⃣、数据经过散列处理H形成摘要：

H ( M )

2️⃣、用公钥密码算法R S A RSARSA对摘要数字签名：

s i g R S A − r k A ( H ( M ) )

3️⃣、链接数据与数字签名得到证书，（并进行压缩，压缩可省略）：

s i g R S A − r k A ( H ( M ) ) ∣ ∣ M

4️⃣、对称加密算法A E S AESAES加密前一步得到的证书：

A E S s k ( s i g R S A − r k A ( H ( M ) ) ∣ ∣ M )

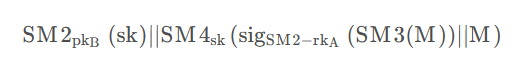
5️⃣、公钥密码算法R S A RSARSA构造数字信封，封装密钥：

R S A p k B ( s k ) RSA\_{pk\_B}(sk)

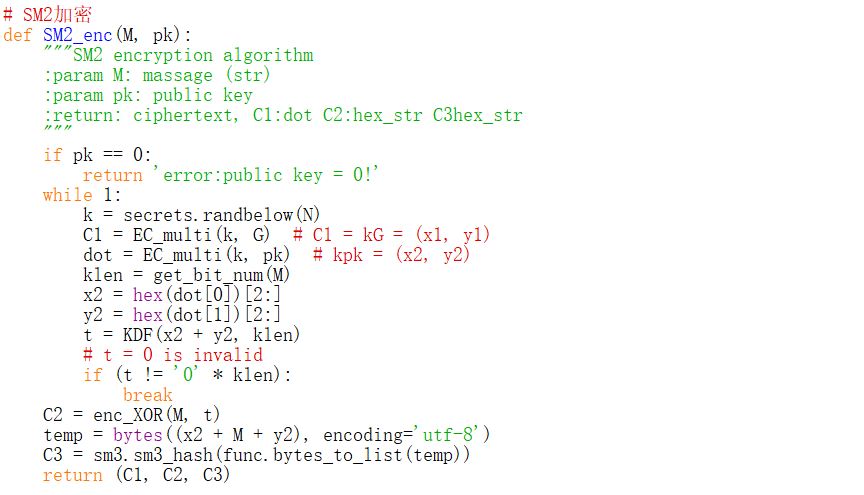
6️⃣、链接，完成PGP加密：

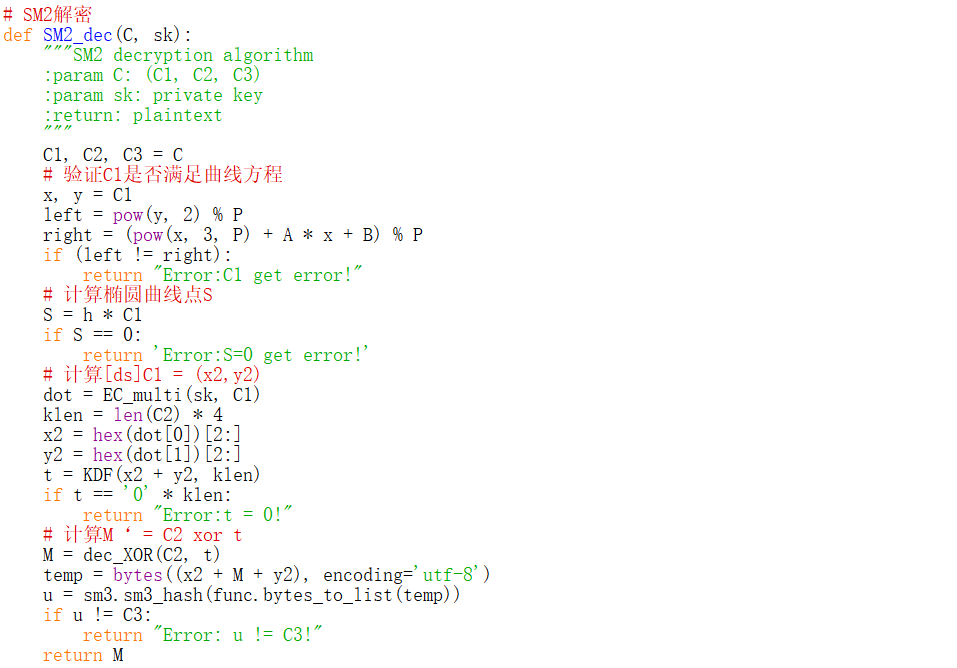
R S A p k B ( s k ) ∣ ∣ A E S s k ( s i g R S A − r k A ( H ( M ) ) ∣ ∣ M )

除了上述利用H 、 R S A 、 A E S H、RSA、AESH、RSA、AES的PGP加密原理应用，还可以用S M 2 、 S M 3 、 S M 4 SM2、SM3、SM4SM2、SM3、SM4构建一个同样具有机密性和认证服务的文件传输方案:



#四、代码摘要





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

#一、文件内容

1. SM2.py:包含SM2加解密算法以及签名验签算法

2. server.py:server端

3. client.py:client端

#运行指导:在pycharm中先运行server.py,然后运行client.py

#二、实验内容

implement sm2 2P sign with real network communication

公钥:P = [(d1d2)^-1 - 1] \* G

私钥:d = (d1d2)^-1 - 1

对于消息,Server与Client需要协作才能进行签名,因为二者均不能独自计算得到私钥

image

(一)Server端:

1. 接收Client端P1 = d1^-1 \* G

2. 生成私钥d2,计算公钥P = [(d1d2)^-1 - 1] \* G并公开公钥P

3. 与client进行交互共同签名:接收client发送的Q1,e并返回r,s2,s3给client

(二)Client端:

1. 生成私钥d1以及P1 = d1^-1 \* G,将P1发送给server

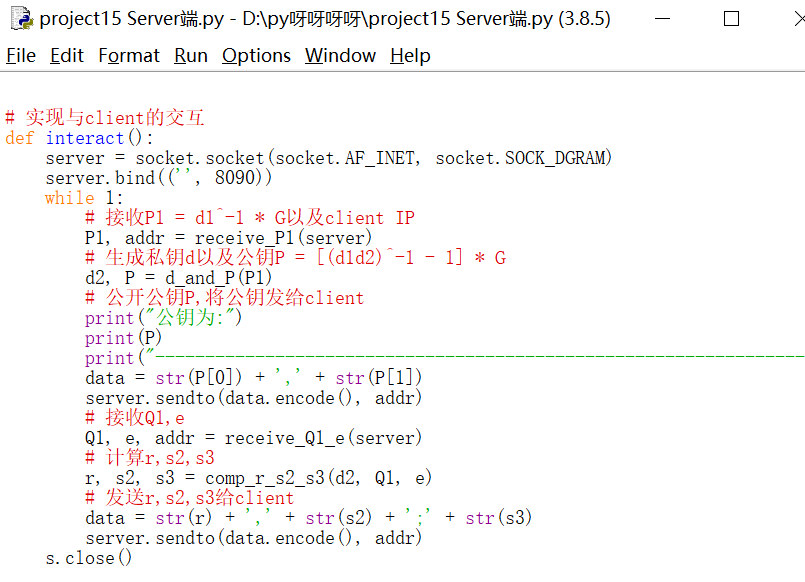
2. 与server进行交互共同签名：

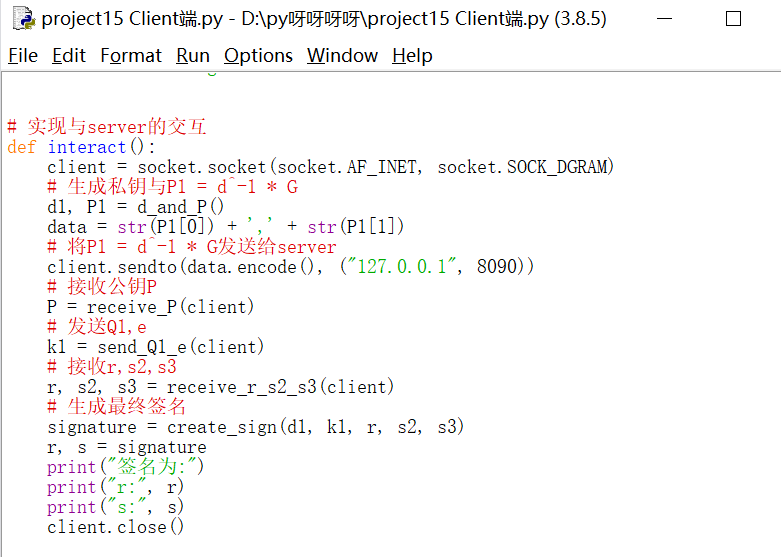
(1) 规定两方ID以及消息msg

(2) 计算Q1,e并发送给server

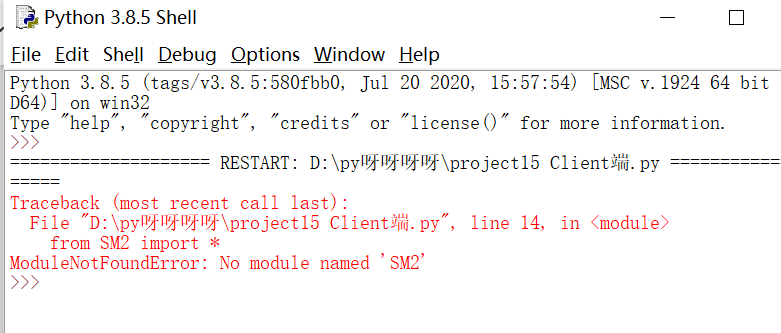
(3) 接收r,s2,s3并计算最终签名(r, s)

＃三、代码摘要





＃四、运行结果



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

#一、文件内容

1. SM2.py:包含SM2加解密算法以及签名验签算法

2. server.py:server端

3. client.py:client端

4. 运行指导:在pycharm中先运行server.py,然后运行client.py

#二、实验内容

implement sm2 2P decrypt with real network communication

公钥:P = [(d1d2)^-1 - 1] \* G

私钥:d = (d1d2)^-1 - 1

对于密文,Server与Client需要协作才能进行解密,因为二者均不能独自计算得到私钥 image

#三、交互流程

(一)Server端:

1. 接收Client端P1 = d1^-1 \* G

2. 生成私钥d2并计算公钥P = [(d1d2)^-1 - 1] \* G

3. 这里Server端自己生成密文并将密文返回给client

4. 与client进行交互共同解密:接收client发送的T1并返回T2给client

(二)Client端:

1. 生成私钥d1以及P1 = d1^-1 \* G

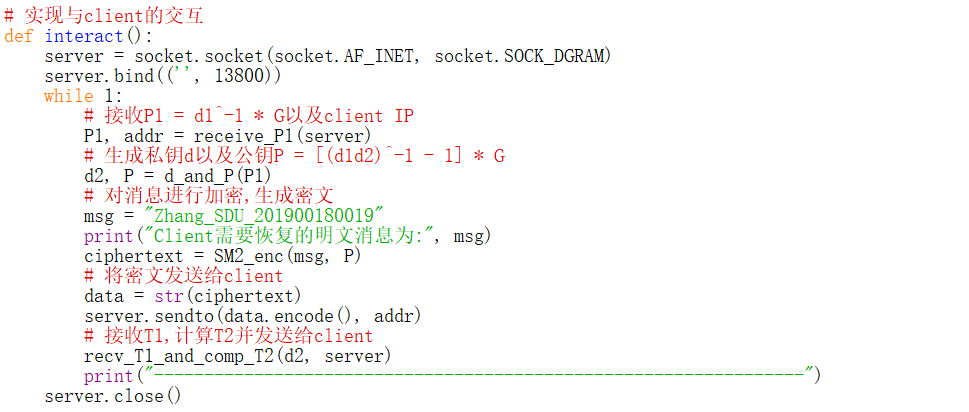
2. 接收密文

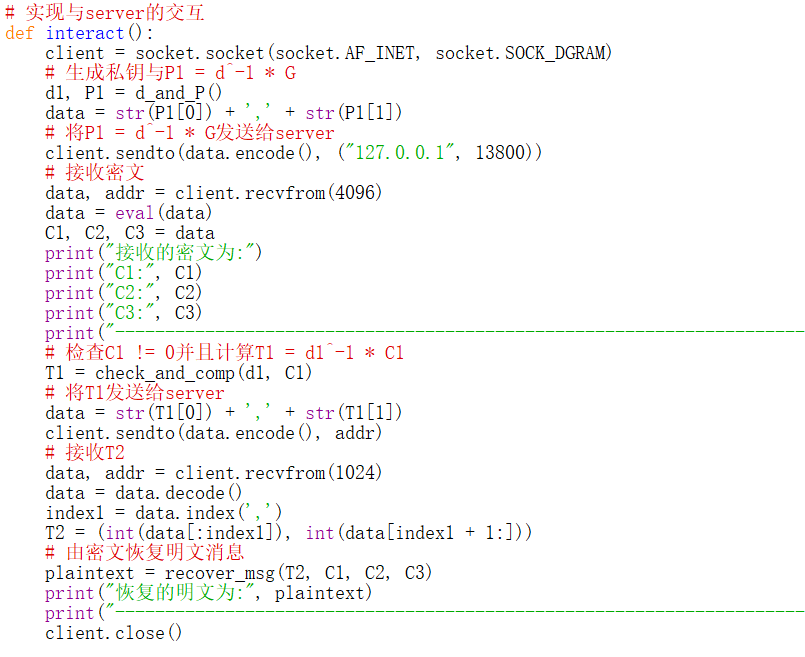
3. 与server进行交互共同解密：

(1) 检查C1 != 0并且计算T1 = d1^-1 \* C1

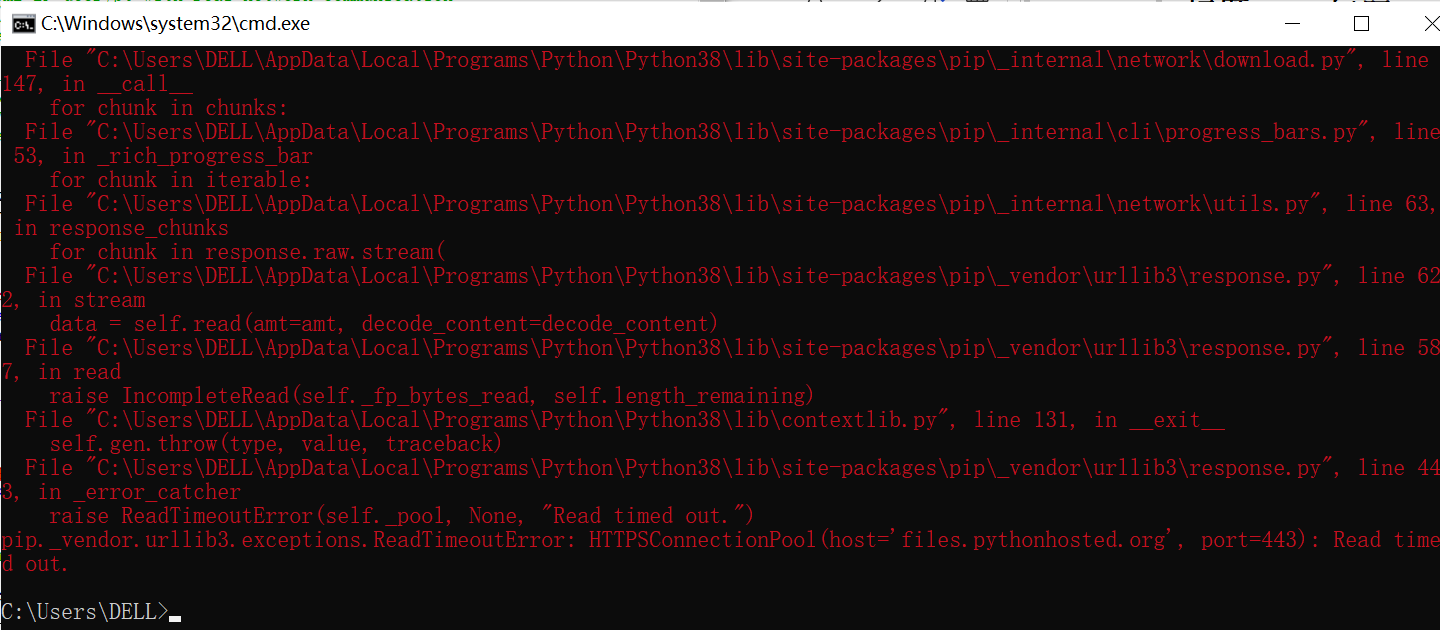
(2) 接收T2并恢复明文

＃四、代码摘要





＃五、运行结果



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

#一、Firefox记住密码插件

浏览器一般都具有自动记录密码功能,但是这往往也会给我们带来一些的开发上的困扰,比如火狐浏览器,首先抛出问题:

前端加密一般是:对密码框内的密码进行加密,然后又把加密后的值塞回密码框中,并通过form表单提交到后端进行解密,验证登录。

如果是火狐浏览器,它会监控post动作,询问是否记录账号密码,以便下次登录自动填充,但这时火狐浏览器保存的却是加密后的密码,导致下次打开登录页面自动填充了加密后的密码，再次点击登录密码会二次加密，从而登录失败！

经过验证，Chrome会很机智的记住加密前的密码，没有这个问题，怎么解决火狐出现的这种情况，首先先看看火狐的自动记住密码和自动填充的过程：

1. 火狐浏览器自动记住密码是在监控到post登录请求后，从上往下，找到最后一个type=‘password’的input框，然后询问你是否记住该密码；

2.火狐浏览器自动填充是从上而下，寻找第一个为type=‘password’的input框，并把值填充进去，不管这个input是否是隐藏（display:none）的。

结合以上分析过程，我们就能解决自动填充的问题：

1. 把原来的密码框A设置成隐藏域类型，即type=‘hidden’，隐藏起来,前端看不见，其余id，name属性不变；
2. 在原来的密码框A下面增加一个密码框B：\u003Cinput type='password' id='passwordTemp' >,PS:此处不要设置name属性,form表单就不会提交;
3. 在前端登录按钮的click()方法中执行如下逻辑:
4. .获取B的值value,js加密后得到encryptedValue,并把encryptedValue赋值给A,然后submmit(此时,表单把A的值提交到后端);
5. .执行submmit后,将encryptedValue再赋值给B,这样前端密码框就会有密码变长的加密效果,而且加密后的密码是在post过后赋值的,火狐浏览器记住的是post时B的未加密密码;

这样,我们就能保证,

1.post提交的密码是加密的,不会有被抓包解析出来的危险;

2.火狐也会自动记录正确未加密的密码;

3.密码自动填充,也会准确找到显示B密码框,不会填充到A中,导致每次还要手动选择密码;

#二、firefox记住密码插件部分源码

<?xml version="1.0" encoding="UTF-8"?>

<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:em="http://www.mozilla.org/2004/em-rdf#">

<Description about="urn:mozilla:install-manifest">

<em:id>firesheep@codebutler.com</em:id>

<em:type>2</em:type>

<em:name>Firesheep</em:name>

<em:version>0.1</em:version>

<em:creator>Eric Butler</em:creator>

<em:contributor></em:contributor>

<em:aboutURL>chrome://firesheep/content/about.xul</em:aboutURL>

<em:optionsURL>chrome://firesheep/content/preferences/prefsWindow.xul</em:optionsURL>

<em:homepageURL>http://codebutler.github.com/firesheep</em:homepageURL>

<em:updateURL>http://codebutler.github.com/firesheep/update.rdf</em:updateURL>

<em:updateKey>MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCx0jVpi2A7WnvIDJlqYFlSk+9Ny6ccmRSFShlcpmjqfEGYDJFNLnXERSHhNGIVYArBhHkunwWyZJ4Es4kUZ6BGE0IbEIPhJPgD18DCdgRwFL+ts9sn+2DWxd0KrU62P9phCNcsvrlL+G+O/znSoBYDeiMS0OrGN5HcftwdReHaMQIDAQAB</em:updateKey>

<em:unpack>true</em:unpack>

<em:targetPlatform>Darwin\_x86-gcc3</em:targetPlatform>

<em:targetPlatform>WINNT\_x86-msvc</em:targetPlatform>

<em:targetApplication>

<Description>

<em:id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</em:id> <!-- Firefox -->

<em:minVersion>3.6.10</em:minVersion>

<em:maxVersion>3.6.\*</em:maxVersion>

</Description>

</em:targetApplication>

</Description>

</RDF>

#三、谷歌记住密码插件

ShowPassword是一款能够显示出当前密码框中的密码的谷歌浏览器插件，在用户输入密码的时候，如果忘记了当前已经输入的密码，或者不确定输入的密码是否正确的情况下，就可以使用ShowPassword插件来临时查看一下当前的密码。

还有一种情况是，用户如果使用的谷歌浏览器的记住密码功能，在网站上输入密码的地方，谷歌浏览器会自动帮助用户填上帐号和密码，但是久而久之，用户就会因为长时间没有操作过，而忘记当前网站中的密码，这时候，如果使用ShowPassword插件，就可以临时查看一下当前密码框中的密码，来用户一个提醒的作用。

ShowPassword的使用方法:

1. 在谷歌浏览器中安装ShowPassword插件，并在Chrome的扩展器中启动显示密码的功能，ShowPassword插件的下载地址可以在本文的下方找到，离线ShowPassword插件的安装方法可参考：怎么在谷歌浏览器中安装.crx扩展名的离线Chrome插件？
2. 安装完成以后，用户就可以使用Chrome打开需要输入帐号和密码的网站，在密码框中输入一定的字符以后，把鼠标放在密码框500毫秒以后，该密码框就会把密码以明文的方式显示出来

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

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

#一、实验内容

本实验实现了当ECDSA验签算法在不验证m的情况下,能够通过公钥来伪造合法签名

#二、攻击原理

1. 在 Fn\* 上随机选择u,v,计算R = (x,y) = uG + vP (P为公钥)

2. r = x mod n

3. e = ruv^-1 mod n

4. s = rv^-1 mod n

5. signature = (r,s)

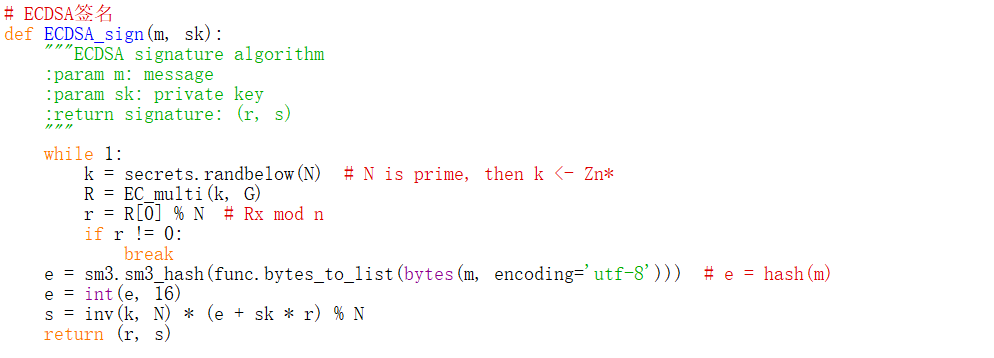
#三、攻击前提

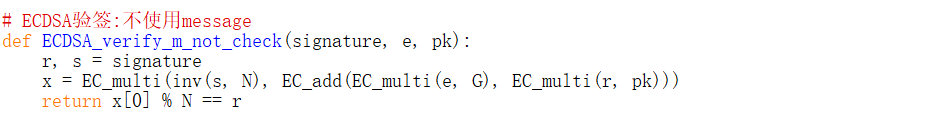
验签算法:不验证消息m

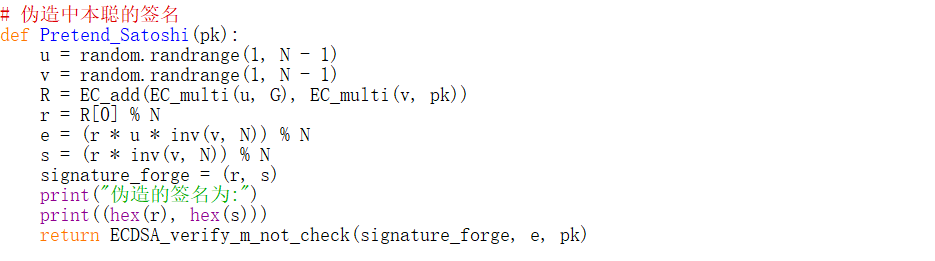
e \* s^-1 \* G + r \* s^-1 \* P = kG = R

前提数据:Satoshi的公钥,这里我们通过随机生成来代替Satoshi的真实公钥

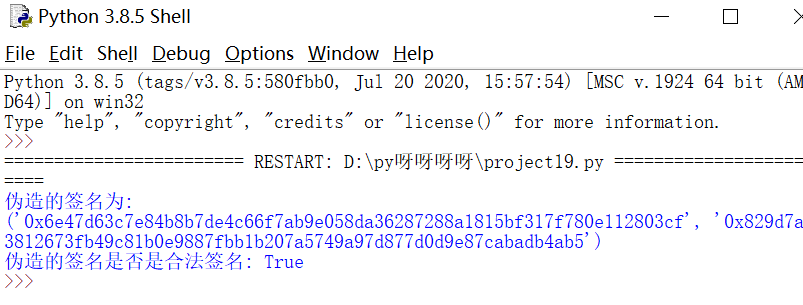
四、代码摘要







#五、运行结果



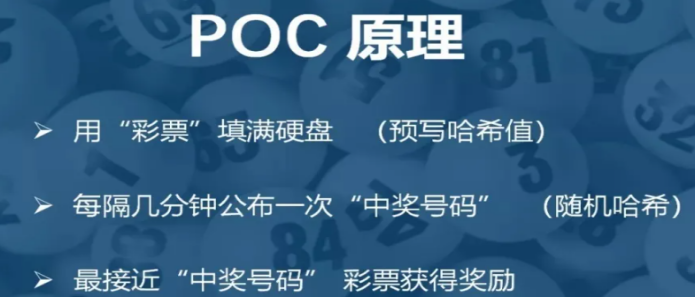
**\*Project20: ECMH PoC**

#一、概念

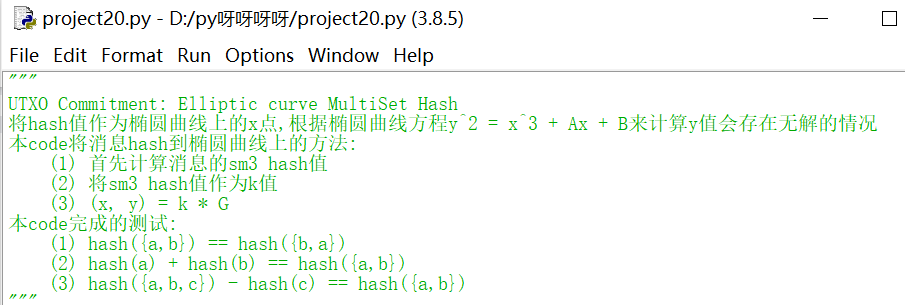
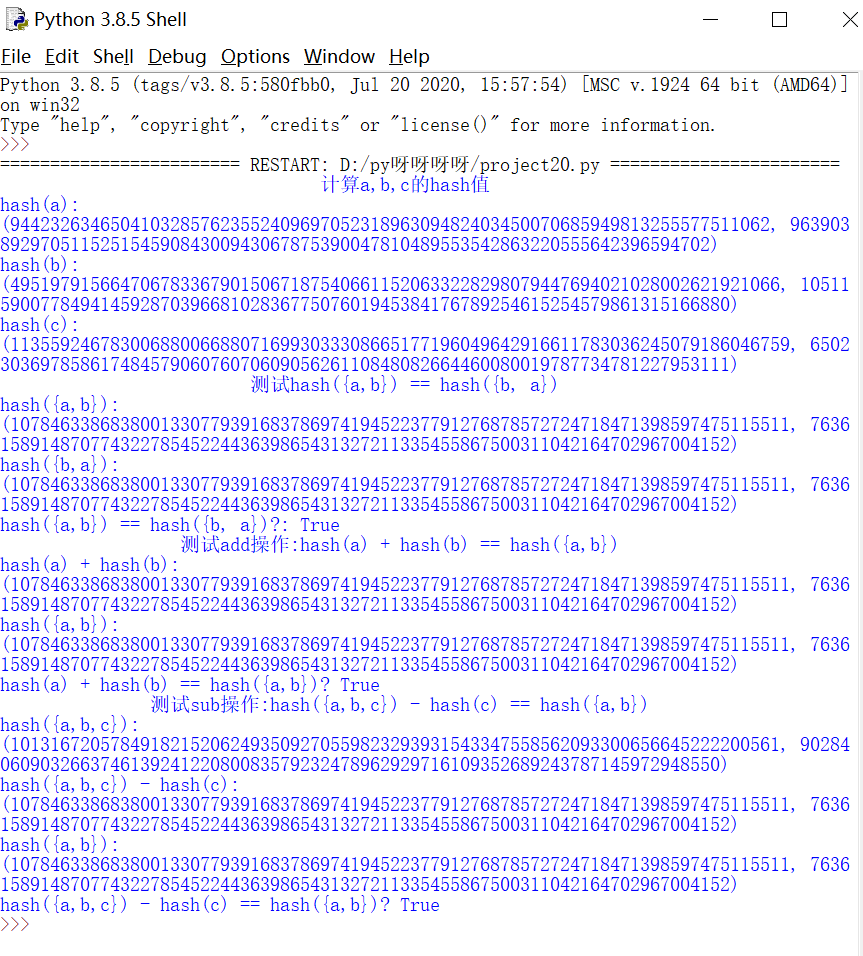
PoC即Proof of Capacity，也就是容量证明。PoC也被称作空间证明，是PoW[共识机制](https://www.zhihu.com/search?q=%E5%85%B1%E8%AF%86%E6%9C%BA%E5%88%B6&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":872494760}" \t "_blank)的升级版。主要就是通过数据预生成后检索的算法，这使得整个系统比PoW更加快速，PoC仅需在数分钟内生成新区块，而PoW需要10分钟。相比PoW是通过计算来获取区块的[打包权](https://www.zhihu.com/search?q=%E6%89%93%E5%8C%85%E6%9D%83&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":872494760}" \t "_blank)，PoC则是将“计算”变成了“寻找”。

整个[记账权](https://www.zhihu.com/search?q=%E8%AE%B0%E8%B4%A6%E6%9D%83&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":872494760}" \t "_blank)的争取过程如下：

在挖矿还没开始前，网络就会把破解谜题的[计算方法](https://www.zhihu.com/search?q=%E8%AE%A1%E7%AE%97%E6%96%B9%E6%B3%95&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":872494760}" \t "_blank)（solutions）储存在硬盘空间里。这些计算方法有的比较快，有的比较慢，如果你的硬盘里恰好有一个计算方法，是目前最近产生的这个区块里的谜题所对应的“最快解”，那么你就赢得了这个区块的记账权——挖矿的奖励就是你的了。如果你在硬盘空间里拥有越多的计算方法（也被称为plots），你用[最快的速度](https://www.zhihu.com/search?q=%E6%9C%80%E5%BF%AB%E7%9A%84%E9%80%9F%E5%BA%A6&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":872494760}" \t "_blank)去破解当前这个区块的谜题，实现的概率也就越大。

  
PoC共识机制下的挖矿被称为硬盘挖矿，从最早的Burst到大火的IPFS，再到7月的BHD，还有现在的BSN，每一个PoC共识下的项目都会被市场追捧和追随。BSN被誉为PoC算法中的新比特。

PoC共识机制虽然不具备完全取代PoW挖矿的可能性，但是，它的确是代表了矿业新的探索方向。

#二、一些实践  
  


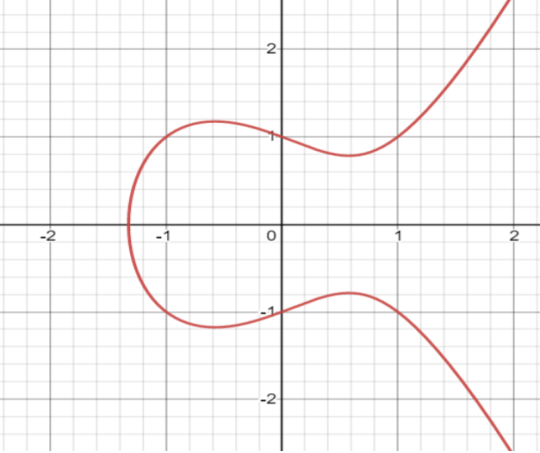
**\*Project21: Schnorr Bacth**

一、基本知识

Schnorr签名算法最初是由德国密码学家ClausSchnorr于2008年提出的，在密码学中，它是一种[数字签名](https://so.csdn.net/so/search?q=%E6%95%B0%E5%AD%97%E7%AD%BE%E5%90%8D&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_41546054/article/details/_blank)方案，以其简单高效著称，其安全性基于某些离散对数问题的难处理性。

密码学中，常用以下形式的椭圆曲线： E: y^2= x^3+ax+b(mod\ p) 同时要求 4a^3+27b^2 ≠0。其中p为一个大素数，a、b、x和y均在有限域 GF(p)中，即从{0,1,···,p-1\}中取值。该曲线常用E\_{p}(a,b)表示。若该曲线上只有有限个离散点，设为N个，则椭圆曲线的阶为N。N越大，椭圆曲线安全性越高。椭圆曲线的阶可通过schoof算法计算求得。

椭圆曲线 E: y^2= x^3-x +1图形如下：



加法规则

椭圆曲线E\_{p}(a,b)在如下定义的加法规则构成Abel群（交换群）。

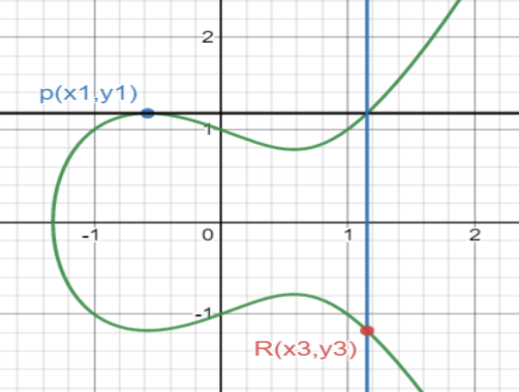
Ο + Ο = Ο

∀P=(x,y)∈E\_{p}(a,b)，有 P +Ο =Ο + P = P

∀P=(x,y)∈E\_{p}(a,b)，有 P + (-P) =Ο，P的逆为 -P = (x,-y)

∀P=(x\_{1},y\_{1}),Q=(x\_{2},y\_{2})∈E\_{p}(a,b)，则 P + Q = R = (x\_{3},y\_{3})∈E\_{p}(a,b)

其中x\_3=λ^2-x\_1-x\_2 ,y\_3=λ(x\_1-x\_3 )-y\_1

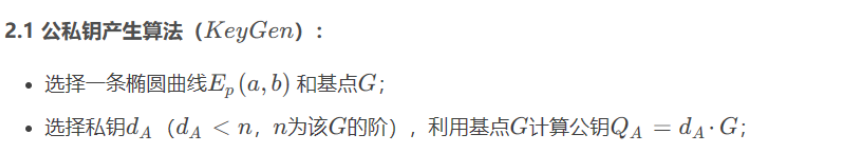


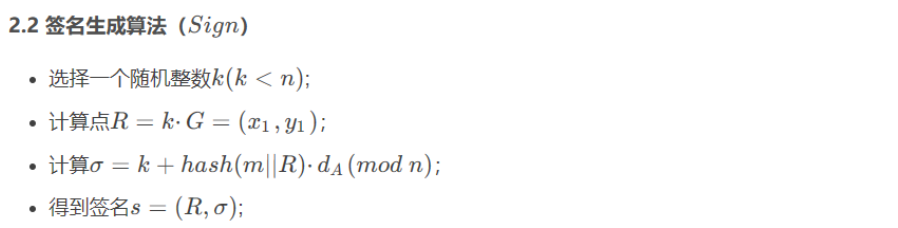
乘法规则

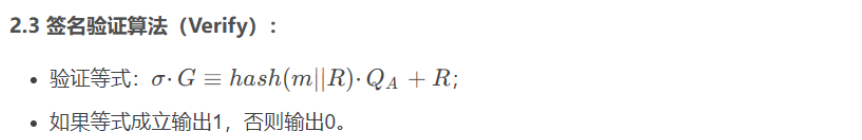
∀k∈Z,∀P∈E\_p (a,b)，有 kP=P+···+P(k个P相加)；

∀s,t∈Z, ∀P∈E\_p (a,b)，有 (s+t)P=sP+tP,s(tP)=(st)P

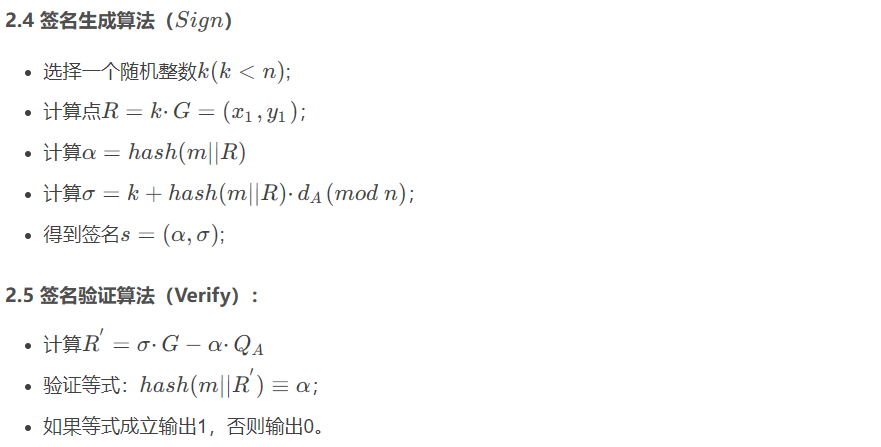
1. 算法细节







Schnorr签名除了上面一种形式外，还有另外一种形式：



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

#一、概述

Merkle Patricia Tree（又称为Merkle Patricia [Trie](https://so.csdn.net/so/search?q=Trie&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_40713201/article/details/_blank)）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：

存储任意长度的key-value键值对数据；

提供了一种快速计算所维护数据集哈希标识的机制；

提供了快速状态回滚的机制；

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；

1前缀树

前缀树（又称[字典树](https://so.csdn.net/so/search?q=%E5%AD%97%E5%85%B8%E6%A0%91&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_40713201/article/details/_blank)），用于保存关联数组，其键（key）的内容通常为字符串。前缀树节点在树中的位置是由其键的内容所决定的，即前缀树的key值被编码在根节点到该节点的路径中。

优势：

相比于哈希表，使用前缀树来进行查询拥有共同前缀key的数据时十分高效，例如在字典中查找前缀为pre的单词，对于哈希表来说，需要遍历整个表，时间效率为O(n)；然而对于前缀树来说，只需要在树中找到前缀为pre的节点，且遍历以这个节点为根节点的子树即可。

但是对于最差的情况（前缀为空串），时间效率为O(n)，仍然需要遍历整棵树，此时效率与哈希表相同。

相比于哈希表，在前缀树不会存在哈希冲突的问题。

劣势：

直接查找效率低下：

前缀树的查找效率是O(m)，m为所查找节点的key长度，而哈希表的查找效率为O(1)。且一次查找会有m次IO开销，相比于直接查找，无论是速率、还是对磁盘的压力都比较大。

可能会造成空间浪费：

当存在一个节点，其key值内容很长（如一串很长的字符串），当树中没有与他相同前缀的分支时，为了存储该节点，需要创建许多非叶子节点来构建根节点到该节点间的路径，造成了存储空间的浪费。

2默克尔树

Merkle树是由计算机科学家 Ralph Merkle 在很多年前提出的，并以他本人的名字来命名,由于在比特币网络中用到了这种数据结构来进行数据正确性的验证，在这里简要地介绍一下merkle树的特点及原理。

在比特币网络中，merkle树被用来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹。此外，由于merkle树的存在，使得在比特币这种公链的场景下，扩展一种“轻节点”实现简单支付验证变成可能，关于轻节点的内容，将会下文详述。

特点：

默克尔树是一种树，大多数是二叉树，也可以多叉树，无论是几叉树，它都具有树结构的所有特点；

默克尔树叶子节点的value是数据项的内容，或者是数据项的哈希值；

非叶子节点的value根据其孩子节点的信息，然后按照Hash算法计算而得出的；

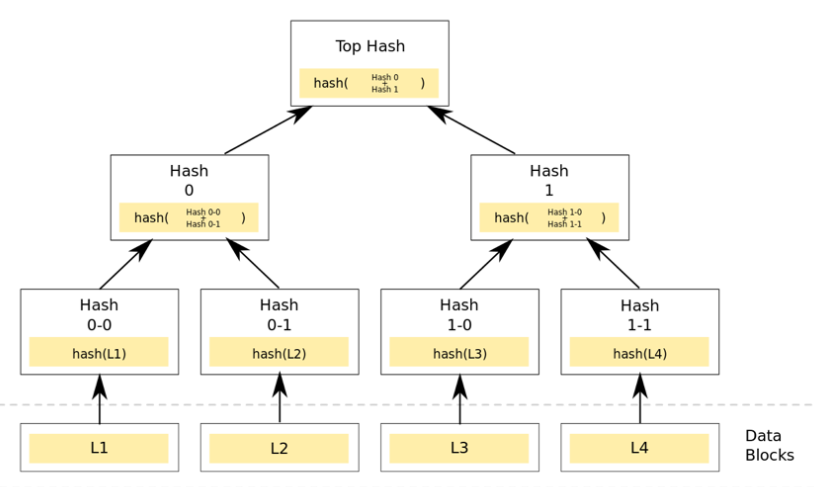
原理：

在比特币网络中，merkle树是自底向上构建的。在下图的例子中，首先将L1-L4四个单元数据哈希化，然后将哈希值存储至相应的叶子节点。这些节点是Hash0-0, Hash0-1, Hash1-0, Hash1-1

将相邻两个节点的哈希值合并成一个字符串，然后计算这个字符串的哈希，得到的就是这两个节点的父节点的哈希值。

如果该层的树节点个数是单数，那么对于最后剩下的树节点，这种情况就直接对它进行哈希运算，其父节点的哈希就是其哈希值的哈希值（对于单数个叶子节点，有着不同的处理方法，也可以采用复制最后一个叶子节点凑齐偶数个叶子节点的方式）。循环重复上述计算过程，最后计算得到最后一个节点的哈希值，将该节点的哈希值作为整棵树的哈希。

若两棵树的根哈希一致，则这两棵树的结构、节点的内容必然相同。



优势

1快速重哈希：

默克尔树的特点之一就是当树节点内容发生变化时，能够在前一次哈希计算的基础上，仅仅将被修改的树节点进行哈希重计算，便能得到一个新的根哈希用来代表整棵树的状态。

2轻节点扩展：

采用默克尔树，可以在公链环境下扩展一种“轻节点”。轻节点的特点是对于每个区块，仅仅需要存储约80个字节大小的区块头数据，而不存储交易列表，回执列表等数据。然而通过轻节点，可以实现在非信任的公链环境中验证某一笔交易是否被收录在区块链账本的功能。这使得像比特币，以太坊这样的区块链能够运行在个人PC，智能手机等拥有小存储容量的终端上。

劣势：

1存储空间开销大：

#二、结构设计

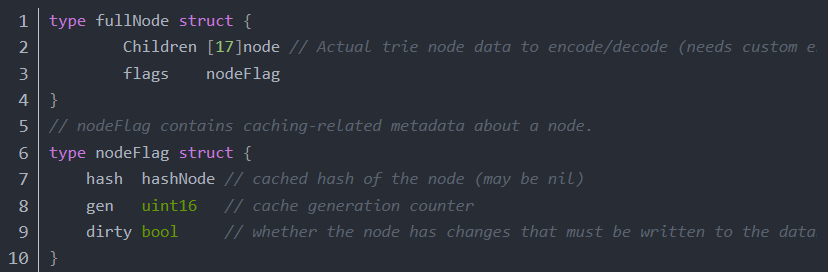
1 节点分类

前缀树无论是查询操作，还是对数据的增删改，不仅效率低下，且存储空间浪费严重。故，在以太坊中，为MPT树新增了几种不同类型的树节点，以尽量压缩整体的树高、降低操作的复杂度。

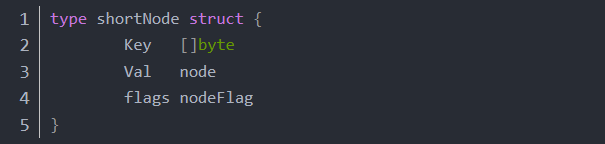
MPT树中，树节点可以分为以下四类：空节点、分支节点、叶子节点、扩展节点

空节点：用来表示空串

分支节点：用来表示MPT树中所有拥有超过1个孩子节点以上的非叶子节点， 其定义如下所示：



叶子节点&&扩展节点：



其中关键的字段为：

Key：用来存储属于该节点范围的key；

Val：用来存储该节点的内容；

其中Key是MPT树实现树高压缩的关键！

2安全的MPT

以上介绍的MPT树，可以用来存储内容为任何长度的key-value数据项。倘若数据项的key长度没有限制时，当树中维护的数据量较大时，仍然会造成整棵树的深度变得越来越深，会造成以下影响：

查询一个节点可能会需要许多次IO读取，效率低下；

系统易遭受Dos攻击，攻击者可以通过在合约中存储特定的数据，“构造”一棵拥有一条很长路径的树，然后不断地调用SLOAD指令读取该树节点的内容，造成系统执行效率极度下降；

所有的key其实是一种明文的形式进行存储；

为了解决以上问题，在以太坊中对MPT再进行了一次封装，对数据项的key进行了一次哈希计算，因此最终作为参数传入到MPT接口的数据项其实是(sha3(key), value)

优势：

1传入MPT接口的key是固定长度的（32字节），可以避免出现树中出现长度很长的路径；

劣势：

1每次树操作需要增加一次哈希计算；  
2需要在数据库中存储额外的sha3(key)与key之间的对应关系；

#三、基本操作

1 Get

一次Get操作的过程为：

将需要查找Key的Raw编码转换成Hex编码，得到的内容称之为搜索路径；

从根节点开始搜寻与搜索路径；

内容一致的路径；

若当前节点为叶子节点，存储的内容是数据项的内容，且搜索路径的内容与叶子节点的key一致，则表示找到该节点；反之则表示该节点在树中不存在。

若当前节点为扩展节点，且存储的内容是哈希索引，则利用哈希索引从数据库中加载该节点，再将搜索路径作为参数，对新解析出来的节点递归地调用查找函数。

若当前节点为扩展节点，存储的内容是另外一个节点的引用，且当前节点的key是搜索路径的前缀，则将搜索路径减去当前节点的key，将剩余的搜索路径作为参数，对其子节点递归地调用查找函数；若当前节点的key不是搜索路径的前缀，表示该节点在树中不存在。

若当前节点为分支节点，若搜索路径为空，则返回分支节点的存储内容；反之利用搜索路径的第一个字节选择分支节点的孩子节点，将剩余的搜索路径作为参数递归地调用查找函数。

2 Insert

插入操作也是基于查找过程完成的，一个插入过程为：

根据3.1中描述的查找步骤，首先找到与新插入节点拥有最长相同路径前缀的节点，记为Node；

若该Node为分支节点：  
（1）剩余的搜索路径不为空，则将新节点作为一个叶子节点插入到对应的孩子列表中；  
（2）剩余的搜索路径为空（完全匹配），则将新节点的内容存储在分支节点的第17个孩子节点项中（Value）；

若该节点为叶子／扩展节点：  
（1）剩余的搜索路径与当前节点的key一致，则把当前节点Val更新即可；  
（2）剩余的搜索路径与当前节点的key不完全一致，则将叶子／扩展节点的孩子节点替换成分支节点，将新节点与当前节点key的共同前缀作为当前节点的key，将新节点与当前节点的孩子节点作为两个孩子插入到分支节点的孩子列表中，同时当前节点转换成了一个扩展节点（若新节点与当前节点没有共同前缀，则直接用生成的分支节点替换当前节点）；

若插入成功，则将被修改节点的dirty标志置为true，hash标志置空（之前的结果已经不可能用），且将节点的诞生标记更新为现在；

3 Delete

删除操作与插入操作类似，都需要借助查找过程完成，一次删除过程为：

根据3.1中描述的查找步骤，找到与需要插入的节点拥有最长相同路径前缀的节点，记为Node；

若Node为叶子／扩展节点：  
（1）若剩余的搜索路径与node的Key完全一致，则将整个node删除；  
（2）若剩余的搜索路径与node的key不匹配，则表示需要删除的节点不存于树中，删除失败；  
（3）若node的key是剩余搜索路径的前缀，则对该节点的Val做递归的删除调用；

若Node为分支节点：  
（1） 删除孩子列表中相应下标标志的节点；  
（2） 删除结束，若Node的孩子个数只剩下一个，那么将分支节点替换成一个叶子／扩展节点；

若删除成功，则将被修改节点的dirty标志置为true，hash标志置空（之前的结果已经不可能用），且将节点的诞生标记更新为现在；

4 Update

更新操作就是3.2Insert与3.3Delete的结合。当用户调用Update函数时，若value不为空，则隐式地转为调用Insert；若value为空，则隐式地转为调用Delete，故在此不再赘述。

5 Commit

Commit函数提供将内存中的MPT数据持久化到数据库的功能。在第一章中我们提到的MPT具有快速计算所维护数据集哈希标识以快速状态回滚的能力，也都是在该函数中实现的。

在commit完成后，所有变脏的树节点会重新进行哈希计算，并且将新内容写入数据库；最终新的根节点哈希将被作为MPT的最新状态被返回。

一次MPT树提交是一个递归调用的过程，在介绍MPT提交过程之前，我们首先介绍单个节点是如何进行哈希计算和存储的。