

# 汇总报告：Project1-22

## Project-1,implement the naïve birthday attack of reduced SM3

### 1, 实验内容

分为两部分解释：sm3实现和生日攻击

#### 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。

手动实现sm3的主体部分代码如下（即seg\_func () ）

```
def seg_func(data):
    #输入信息类型为bit stream
    IV=[]
    group_number=int(len(data)/64)+1#64byte per group
    if(group_number==1):
        padding_data=padding_func(data[:])
        IV=compression_func(IV0,padding_data)
        for i in range(0,8):
            IV[i]=hex(IV[i])[2:]
        padded_IV=[s.zfill(8) for s in IV]
        final_IV=''.join(padded_IV)
        return final_IV
    for i in range(0,group_number):
        if(i==0):
            IV=compression_func(IV0,data[0:64])
            continue
        if(i==group_number-1):
            padding_data=padding_func(data[64*i:])
            #获得填充后信息
            IV=compression_func(IV,padding_data)
            for i in range(0,8):
                IV[i]=hex(IV[i])[2:]
            padded_IV = [s.zfill(8) for s in IV]
            final_IV=''.join(padded_IV)
            return final_IV
        else:
            IV=compression_func(IV,data[64*i:64*(i+1)])
            continue
```

- 参数选取：

初始IV：查阅相关信息后可以得到初始IV

常量T：本代码中，前16轮使用一个T，后面的轮数使用另一个T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 $4 * \text{Index}$ 位。
- 通过时间测试函数，最后输出找到碰撞的时间

## 2，运行指导

- 直接用IDLE运行代码即可

## 3，实验结果及分析

## 1,手动实现sm3与库函数sm3的时间比较

运行程序，得到如下结果：

```
= RESTART: C:\Users\Mr. smile\Desktop\Project1_naive_birthday_attack_sm3\birthday
attck to SM3.py
e8db2a1b11871beb0b18753d869502aaa8f6e49c5f7010bacb8dbcf398fbce54
sm3_myself take time: 0.011730432510375977 s
329a31a2ccabe748be6fcf52b2ffb5f61622ca499caab57f8867a896313f3f21
sm3 by library function take time: 0.0015027523040771484 s
```

可以发现，手动实现的时间与库函数时间相差并不多，如果再能把padding\_func () 等函数做更多的优化，效果会更好。

## 2, 不同长度碰撞的时间

通过修改Index，可以测试不同长度碰撞，运行程序得到如下结果：

Index=2 :

可能是选择的位数过短，有时能在几次内就找到，时间非常快，测试多次后，得到如下：

```
find 68
r1 687d93b7fa42e4107db5c249cf99abd5ad6731a892209e9de7cde64c6a9b2e02
r2 687e19841bd0270585e8164cd98884daa968f0b90319bef9ae49ed4f1afb5a6f
time 3.1096415519714355 s
```

Index=3 :

```
find 3b6
r1 3b6b5c8445674e0c1bc256627f85b3a9a96f3efc963018cca5c9f28e351a050c
r2 3b6111f008cd97209743513209800669b8ee732116358ca3ee45ff89a8fb244a
time 68.63750147819519 s
```

Index=4 :

```
find 049a
r1 049a8a684e69d236a326de98729544dbaa4c216a1130babce4adb40110f246df
r2 049a594b6b64a37fd774d6ece695a038927fd4881629bcd48181ee4b70d30c68
time 124.50655674934387 s
```

Index=5 :

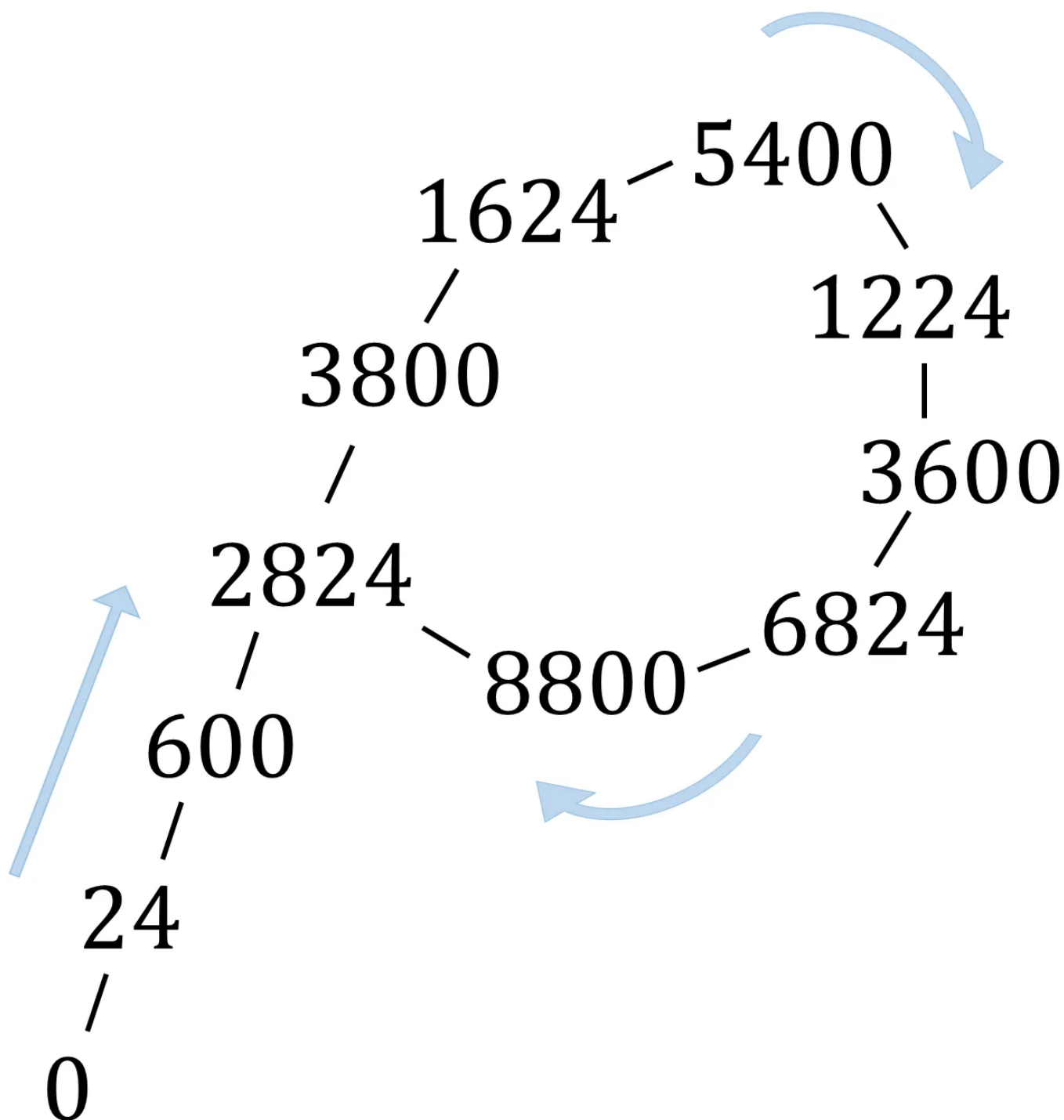
```
find 22b7e
r1 22b7e4fa181e80782fed346fe08ca4f6ad6aaa1017a52fd0a9c5da49d4bfa65e
r2 22b7e8f213d1d9b1078550b2f88106c9bd7d34c0beb2f023b3896e43d87fcf07
time 808.1945114135742 s
```

## Project-2, implement the Rho method of reduced SM3

### 1, 实验内容

同样使用手动实现sm3, 与上一部分内容相似, 不再赘述

ρ方法寻找碰撞



- $\rho$ 算法使用一个特定的循环群以及迭代函数，由于循环群中的元素数量有限，所以一定会进入一个循环，正如上图所示。
- $\rho$ 算法则将进入该循环中的数对比，寻找是否有碰撞。

体现 $\rho$ 算法逻辑的代码部分如下：

```
Index=3#标志要寻找多少位的碰撞
bit_length=64#给出一个比特长度，用于生成定长素数作为循环群的模数
n=randprime(2**(bit_length-1),2**bit_length-1)
x=2
while True:
    x=f(x,n)
    x2=f(x,n)
    x_str=str(x).encode()
    re1=seg_func(x_str)
    x2_str=str(x2).encode()
    re2=seg_func(x2_str)
    if(re1[0:Index]==re2[0:Index]):
        print('find',re1[0:Index])
        break
```

## 2，运行指导

直接使用IDLE运行代码即可，依旧会输出手动和库的sm3的测试时间，然后是 $\rho$ 算法的时间。

## 3，运行结果

参数Index用于确定要寻找多少位的碰撞

Index=2 :

```
find 77
rel 77a3b17aef9794c5a7cbb8231230e8535790cf41e8cacfdd7c7af1a78a3533bb
re2 771a11f226d6d4caedd5e7e4642fd28577998df1614f96131c5611380f0cb0a1
time 1.858950138092041 s
```

Index=3 :

多次测试后得到取较为平均结果:

```
find 0b0
rel 0b0a67f5dd790c1ef9cbadde2429bcac5690cb53e9ce97d5187619be4f14f190
re2 0b05de5e1cc37f9064d1aed22d2eba2706926f62cbefdf653e7251b24b447931
time 8.864741802215576 s
```

Index=4 :

```
find 6c43
rel 6c43052925345955b801fc14207721115490eb63f9eee7115c32913a6b04e9b1
re2 6c43c0ec6e65c5b8e8d50d2c247cfc827690d74be9cece1515f282a31b0459b8
time 120.71478414535522 s
```

可以看出在小范围内p方法的效率要好于随机选取，但是当碰撞位数到16位，时间几乎相同，p方法可能存在在循环中找不到碰撞的情况。

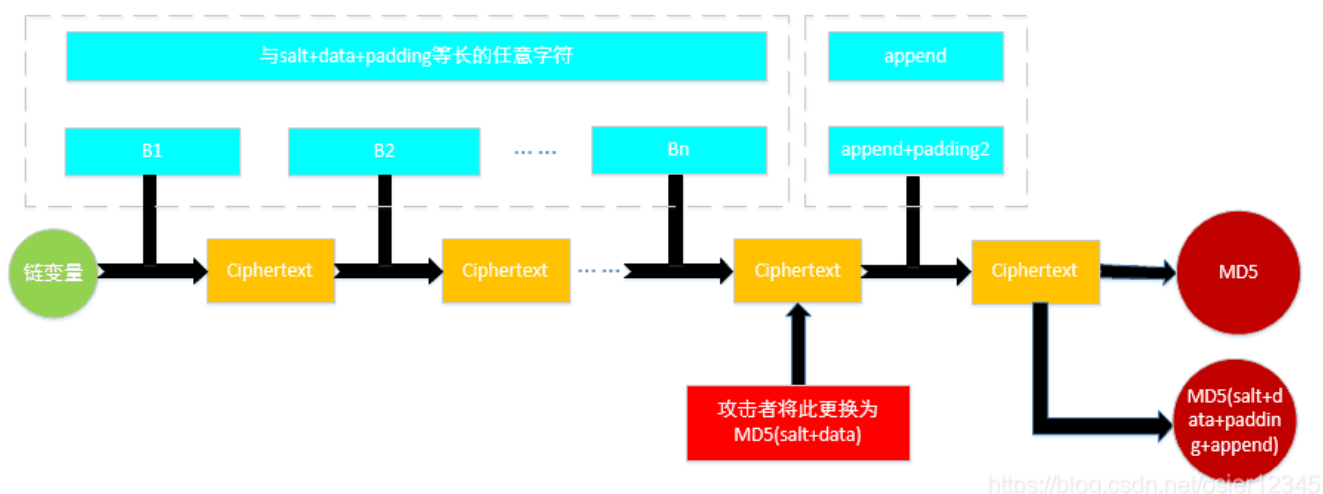
## Project-3, implement length extension attack for SM3

---

### 1, 实验内容

sm3用手动实现的，这样便于替换初始IV，实现长度扩展攻击

长度扩展攻击：



- 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。
- 长度扩展攻击代码的关键部分如下：

```
IV_fake=[]
for i in range(0,8):
    IV_i=re_hash[i:i+8]#字符串类型
    IV_i=int(IV_i,16)
    IV_fake.append(IV_i)
#经过该循环，做出假的初始IV进行替换
data0=seg_func_fake(data_after.encode(),IV_fake)
print(seg_func_fake(data_after.encode(),IV_fake))
```

上述部分可以从前一部分消息的hash值开始直接计算扩展后消息的hash值

## 2, 运行指导

直接运行代码，默认会输出两种计算方法的hash值。

## 3, 实验结果

[illegible]

第一个输出内容是直接替换初始IV，计算后半部分hash

后面输出的hash是从头开始计算拼接后消息的hash。

## Project-4, optimize SM3 implementation

## 1, 实验内容

本实验使用SIMD（Single Instruction Multiple Data），提高并行度来加速sm3的运行，并且使用循环展开进一步提速。

- 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
- 消息扩展部分如下：



```

for (j = 16; j < 68; j += 4) {
    /* w_4 = (W1[j - 3], W1[j - 2], W1[j - 1], 0) */
    w_4 = _mm_loadu_si128((__m128i*)(W1 + j - 3));
    w_4 = _mm_andnot_si128(M, X);

    w_4 = _mm_rotl_epi32(w_4, 15);
    mm_4 = _mm_loadu_si128((__m128i*)(W1 + j - 9));
    w_4 = _mm_xor_si128(w_4, K);
    mm_4 = _mm_loadu_si128((__m128i*)(W1 + j - 16));
    w_4 = _mm_xor_si128(w_4, mm_4);

    /* P1() */
    mm_4 = _mm_rotl_epi32(w_4, 8);
    mm_4 = _mm_xor_si128(K, w_4);
    mm_4 = _mm_rotl_epi32(K, 15);
    w_4 = _mm_xor_si128(w_4, mm_4);

    mm_4 = _mm_loadu_si128((__m128i*)(W1 + j - 13));
    mm_4 = _mm_rotl_epi32(K, 7);
    w_4 = _mm_xor_si128(w_4, K);
    mm_4 = _mm_loadu_si128((__m128i*)(W1 + j - 6));
    w_4 = _mm_xor_si128(w_4, mm_4);

    /* W1[j + 3] ^= P1(ROL32(W1[j + 1], 15)) */
    r_4 = _mm_shuffle_epi32(w_4, 0);
    r_4 = _mm_and_si128(r_4, M);
    mm_4 = _mm_rotl_epi32(r_4, 15);
    mm_4 = _mm_xor_si128(mm_4, r_4);
    mm_4 = _mm_rotl_epi32(mm_4, 9);
    r_4 = _mm_xor_si128(r_4, mm_4);
    r_4 = _mm_rotl_epi32(r_4, 6);
    w_4 = _mm_xor_si128(w_4, r_4);

    _mm_storeu_si128((__m128i*)(W1 + j), X);
}
/* W2 = W1[j] ^ W1[j+4] */
for (int j = 0; j < 64; j += 4) {
    w_4 = _mm_loadu_si128((__m128i*)(W1 + j));
    K = _mm_loadu_si128((__m128i*)(W1 + j + 4));
    w_4 = _mm_xor_si128(w_4, mm_4);
    _mm_storeu_si128((__m128i*)(W2 + j), w_4);
}

```

## 2, 运行指导

使用visual studio可以直接运行，默认会输出消息明文以及hash值，以及测试时间（ms）

## 3, 运行结果

代码运行结果如下：

```
Message:
O my Luve is like a red, red rose, That is newly sprung in June
Hash:
9e9d5b26 377a1258 b942d83b 56a0b6ae de1378ee eee646f4 ace6ade2 8e98e36a
Time taken: 16 microseconds
```

## Project-5, Impl Merkle Tree following RFC6962

### 1, 实验内容

在RFC6962标准下Merkle Tree，RFC6962对Merkle Tree的标准如下：

- 1. **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。

此外，RFC 6962还规定了Merkle树的哈希算法要求：

1. 使用SHA-256哈希算法作为默认的哈希函数。
2. 每个节点的哈希值都是基于其子节点的哈希值计算得到的。

根据这些规定，Merkle树可以用于验证数据的完整性。通过比较树的根哈希与预期的根哈希，可以检测到任何数据的更改或篡改。

#### 代码实现中体现RFC6962

- 在实现过程中，在 `build_merkle_tree` 函数中，将每个叶节点的数据项进行SHA-256哈希计算，并存储到Merkle树中。
- 根据RFC 6962的要求，使用填充的方式构建Merkle树。代码中，使用最近的2的幂次方来计算所需的节点数目，并构建一个大小为  $(2 * \text{num\_nodes} - 1)$  的列表来表示树的所有节点。
- 在 `build_merkle_tree` 函数中，由底部向上计算内部节点的哈希值，使用 `hash_node` 函数对相邻的左子节点和右子节点进行哈希计算，并保存到树中。
- 通过 `print_merkle_tree` 函数来检查根节点和每个节点的哈希值是否正确生成和保存。根节点的哈希值即为Merkle树的根哈希。

代码的关键部分如下：(`build_merkle_tree ()` and `print_merkle_tree ()`)

```

def build_merkle_tree(leaves):
    num_leaves = len(leaves)
    # 计算叶子节点数目最近的2的幂次方
    num_nodes = int(math.pow(2, math.ceil(math.log(num_leaves, 2))))
    tree = [None] * (2 * num_nodes - 1)

    # 填充叶子节点
    for i in range(num_leaves):
        tree[num_nodes - 1 + i] = hashlib.sha256(leaves[i].encode()).digest()

    # 计算内部节点
    for i in range(num_nodes - 2, -1, -1):
        tree[i] = hash_node(tree[2 * i + 1], tree[2 * i + 2])

    return tree

# 打印Merkle树信息
def print_merkle_tree(tree):
    num_levels = int(math.log(len(tree), 2)) + 1
    level_start_index = 0
    for level in range(num_levels):
        print("Level %d:" % level)
        level_length = int(math.pow(2, level))
        for i in range(level_start_index, level_start_index + level_length):
            node = tree[i]
            if node is not None:
                print("Node %d: %s" % (i, node.hex()))
        level_start_index += level_length
    print()

```

## 2, 运行指导

直接运行代码即可，能够打印出merkle tree的各层节点以及根节点hash

## 3, 运行结果

运行程序得到如下结果：

Level 0:

Node 0: e9e1bc4a10c502ef995ede1914b0186ed288b8dde80c8c533a0f93a96490f995

Level 1:

Node 1: d45bb77817ce1959438a171a9e67dc663e39caecb8677fd2ad2ab58aea730ff4

Node 2: e195da4c40f26b85eb2b622e1c0d1ce73d4d8bf4183cd808d39a57e855093446

Level 2:

Node 3: 5b6d4b089e2331b3e00a803326df50cdc2df81c7df405abea149421df227640b

Node 4: aa75cd5e2531f072c7007bb191febdcdbe23d9a4ae30cdbb84cb5a869ce9ab83

Node 5: e195da4c40f26b85eb2b622e1c0d1ce73d4d8bf4183cd808d39a57e855093446

Level 3:

Node 7: 5b41362bc82b7f3d56edc5a306db22105707d01ff4819e26faef9724a2d406c9

Node 8: d98cf53e0c8b77c14a96358d5b69584225b4bb9026423cbc2f7b0161894c402c

Node 9: f60f2d65da046fcaaf8a10bd96b5630104b629e111aff46ce89792e1ca11b18

Node 10: 02c6edc2ad3e1f2f9a9c8fea18c0702c4d2d753440315037bc7f84ea4bba2542

Node 11: e195da4c40f26b85eb2b622e1c0d1ce73d4d8bf4183cd808d39a57e855093446

Root hash:

e9e1bc4a10c502ef995ede1914b0186ed288b8dde80c8c533a0f93a96490f995

## Project-6, impl this protocol with actual network communication

---

### 1, 实验内容

本实验根据PPT上原理去实现，由PPT可知：

# Range Proof With Hash Function

- Alice (born 1978) wants to prove to Bob that her age  $> 21$
- Rely on trusted issuer, proof system need to be useable until 2100
- Suppose this year is 2021

- Trusted Issuer

- Pick 128-bit random seed  $s$ , compute  $s = H_0(seed)$
- $k = 2100 - 1978$ ,  $c = H_1^k(s)$ , sign over  $c$  as  $sig_c$
- Give Alice  $s$  and  $sig_c$

- Alice prove her age  $\geq 21$  to Bob

- E.g., she was born before 2000
- Compute  $d_0 = 2000 - 1978 = 22$ , compute proof  $p = H_1^{d_0}(s)$
- Give Bob  $(p, sig_c)$

- Bob verify Alice's proof

- Compute  $d_1 = 2100 - 2000 = 100$
- Compute  $c' = H_1^{d_1}(p)$ , check  $sig_c$  is for  $c'$

"Commitment"

2100

2000

1980

1979

1978

s :

Issuer picks a r

Over-21 single-chain range proof

可以看到有一个可信第三方辅助，Alice提供证明，Bob验证Alice的证明

相关文档查询[此处](#)

其中的 $H^d(p)$  函数通过查阅资料知：

So Alice's local government (issuer) will provide a signed cryptographic commitment using two collision resistant hash functions  $H_0, H_1$  (a single salted hash or HMAC function with different salt/key per  $H_0$  and  $H_1$  would also work) as follows:

可知 $H_0, H_1$ 可使用加盐的hash或者HMAC。

本次内容没有能够直接体现全部内容逻辑的短篇幅代码，给出Alice的证明代码：

```

#Alice监听消息
mes,addr=s.recvfrom(2048*8)
print(f'Received message from {addr}: {json.loads(mes.decode())}')
data=json.loads(mes.decode())
sig_c_str=data['sig_c']#提取出签名
seed=data['seed']
proof_p=H1(d0,seed)
public_key=data['public_key']
send_data={'p':proof_p,'sig_c':sig_c_str,'public_key':public_key}
json_data=json.dumps(send_data).encode()#对字典处理
s.sendto(json_data,(host, port_Bob))
s.close()

```

Bob的验证代码:

```

#Bob监听消息
mes,addr=s.recvfrom(2048*8)
print(f'Received message from {addr}: {json.loads(mes.decode())}')
data=json.loads(mes.decode())
signature_c_str=data['sig_c']#提取出签名(字符串类型)
signature_bytes = base64.b64decode(signature_c_str.encode('utf-8'))#字节流类型
proof_p=data['p']
c_1=H1(d1,proof_p)#需要对比的“原信息”
print('新生成的待签名消息:',c_1)
#生成c',进行签名对比
public_key=data['public_key']
public_key = RSA.import_key(public_key)

if verify_signature(c_1, signature_bytes,public_key):
    print("签名验证成功")
else:
    print("签名验证失败")

```

## 2, 运行指导

- 运行时我使用命令行运行，打开三个命令行，分别运行。
- 先运行Alice与Bob，二者开始时处于监听状态；最后运行Issuer。
- 如果使用多台设备模拟该过程，需要将绑定的IP分别改成各自IP，因为我使用的都是运行代码的本机IP，所以可以且只能在一台设备模拟三方交互过程。
- 默认输出结果是待签名消息，签名等一系列信息，以及最后是否证明成功的消息

## 3, 运行结果

Issuer:

```
C:\Users\Mr.smile>C:\Users\Mr.smile\Desktop\Project6_RangeProof\issuer.py  
seed: 00ef43edf065f93833fb68fc7bb5525948476ae41964707f25478695eb138bfa8b  
待签名消息: 7aeaf43edf065f93833fb68fc7bb5525948476ae41964707f25478695eb138bfa8b  
Issuer_sig: b"v\x88\xcb(\x99\x96\x18\x0c\x8f\xce\xd3\xd8\xc4\xe4\xb9\xalaX/\x140\xae\x12&\x19\xe5\xd0\xc7\xa8\x1c\xf7\xe6\x96"\x08\x8a\x02\xe1\xce)\xeeH\x93\x15\xb8\x11v\x07GN\x9e\xac\x95\b0\x1a\x86\x0c)\xf7Bxc6'\xc8V\xc4\xbf\xbb_j\x08n\x86\xef\xba\x3cxa8&\xbd_5Un=>\x1a,xce\x80B\|x8a|x87\xab\xda\xdf4\xad\x24xeb\x01Z\xal\xdb\x15 \xb6C6Q&k\x81\xbc\x81d\x92\xec\x3'\xf8\x81\xcl\x5c\x9c\xal\xaii\x93\x9d)0'\xe9\xde\x92\x1a\xfc\xa23\x98\x6d?\xfd5\xa0\x3f\x3f. @\x0f\x8d\xe1\x0c\x1d\x03Y\x3f\x85\x8a3iI50\xef\x1a\x7aTJ'f9\x5e5\x82\xfd4c\xbf5qj_0/x06\xbc\x1b1y\x4fy\xdl\x2xaa\x8aM\xcd\xdea=\xb8\xaa2S\xco\x9a\xdb\xff80\xdd\x0c\x6f\x11[\x16\xda\x02d\xed\xfc\x3f\x3d4\xdc3pY\x4c\x4c8\xea\x18R\xcc9\xbo\x9f\x0f1\xfoC\x10\xbc\xf7\x03\xae'\xe3\xfa\xfc\xaa;J\xcfp\x6c\xaoB\x968\x85NSy)\xc1c\x5b5\x8e\x0c\x8c\x4"
```

Alice:

```
C:\Users\Mr.smile>C:\Users\Mr.smile\Desktop\Project6_RangeProof\Alice.py
Received message from (2.0.1.0, 50000): {'oef43edf065f93833bf68fc7bb552594876ae41964707f25478695eb138bfa8b',
'sig_c': 'doJlKJmWGPCPzTPYxOS0sWfYFLxPrRhImSeQxg6cg9+aWJwiKaUAH0K6eIkxW4EXYHR06erJWwGoYMKfdcxn3IVsS/ul9qCG6G770oJrIgLNVuP
RrOgEJ8ioer2vrRpDlRAVqh2xUgtkM2USzrgbCzgWSS7MP4gcHfNKgaJzPzTdP3pIa/KIzmNY//TWgs/MuQA/Y4QwdAlmz5aJaTVP5sqnVer55LOY7VxS
jpPBrKxwFR50dKqik3M3me9uKJkU8Cp2/hA3qX/EvS2tBjK7WzMB9TdkCIn9YeOoYUsmwnx8x8EMQVtCD60Pq/Ko7Ss9wxqBiljIFT1N5fRy1jgyMx==', 'p
ublic_key': '-----BEGIN PUBLIC KEY-----\nMIIBjAnBgkqhkiG9w0BAQFACQAOCAGMIIBCCKAQEAAMVY2tnih4Uy+bHR+Z1b\n\nmZbiwE58SD0J
QlqdfcbCnRo43cy9ELXaBuRG3wEE3M1X5FRwmJvnh7KFx3Ighs\n\nsvs0Jj1vfwJcDH1rEv6+khv9xWgderZM7lJdKNPsfFYwG1mJr/1KRU1o3o5DZT/nR
0cJjmjrQoU7JYP9WermVf6aTgYA/4rH46IGU8p1ulRhLZgc9gaTmzJ8YvZSRaye\n\nnaZTQ6KvXsFFVufD1C3g+yIQC/IBDD+q/4ptMNHv07RSiAKCEXfRc+/
NCDQ94m+s\n\n6je6V1R4W+zYkVR3zvfcC2mfl1NnZy+7dzwvT8o0jhvy1fGB9D0yJtCskJqEY97\n\nn8QIDAQAB\n\n-----END PUBLIC KEY-----'}
```

**Bob:**

```
C:\Users\Mr.smile>C:\Users\Mr.smile\Desktop\Project6_RangeProof\Bob.py
Received message from (2.0.0.0.1, 12345): 'p': '16ef43edf065f938337b68f8c7bb5525948476ae41964707f25478695eb138bfa8b', 's
ig_c': 'dojLKGJmWGMCPztPYX0.0.0.WFYLrRPrhImGeXug6gc9+aWJwiKAuH0K65f938337b68f8c7bb5525948476ae41964707f25478695eb138bfa8b', 's
ig_eJ8ioerzVzRpDLrAvqh2Xutgm2USZrgbCzgeWSSMP4gchFnKgjaZPfzTD3p1a/K1zmNY'/TWgs/MuQA/Y4qwd4lmz5aJaTVP5sqnVer55YLOy7Vx5jpP
BrGxWR50dKqik3'3mE9uKJkU8Cp2/ha3Qx/EVxsW2gJk7WzMS9TDfFnEY0oYUsmwnv8x8EMQvPcD60P6/Ko7Ss9vwxqBiljiFT1N5fRyljgyMxA==', 'publ
ic key': '-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAAC8AQBICBCgKACeQVZTtnih4UY+bHR+Z1b/nmZbiwE5ESD0JQlgl
dfcbCnRo43c9YELxlaBuRCg5E3M1X5FRwmJvnh7KFw3Jgnsvo5J1gVJcDh1rEw6g+hkhv9xWg4rZm71JdKNPsfFYvgW1GmJdr/1KRut1o3o5DZV/nR0Cj
jmJrQkU7JYP9WermVf6aTgYA/4rH46IGU8p1uRhLZgc9gaTRmzJ8VyzSRaye\|naZtQ6KvXsFFVUfD1C3g+vIQc/IBDD+q/14ptMNHyo7RSiAKCEXFRc+/NCD
Q94m+sN6je6V1R4W+zYkVR3zvfc2mMf11Nnzy+7dzwvT8o0jhyv1fGB9D0yJtCsKJqEY97\|n8QIDAQAB\n-----END PUBLIC KEY-----',
新生成的待签名消息: 7aef43edf065f938337b68f8c7bb5525948476ae41964707f25478695eb138bfa8b
签名验证成功
```

Bob是验证签名一方，可以看出验证签名成功

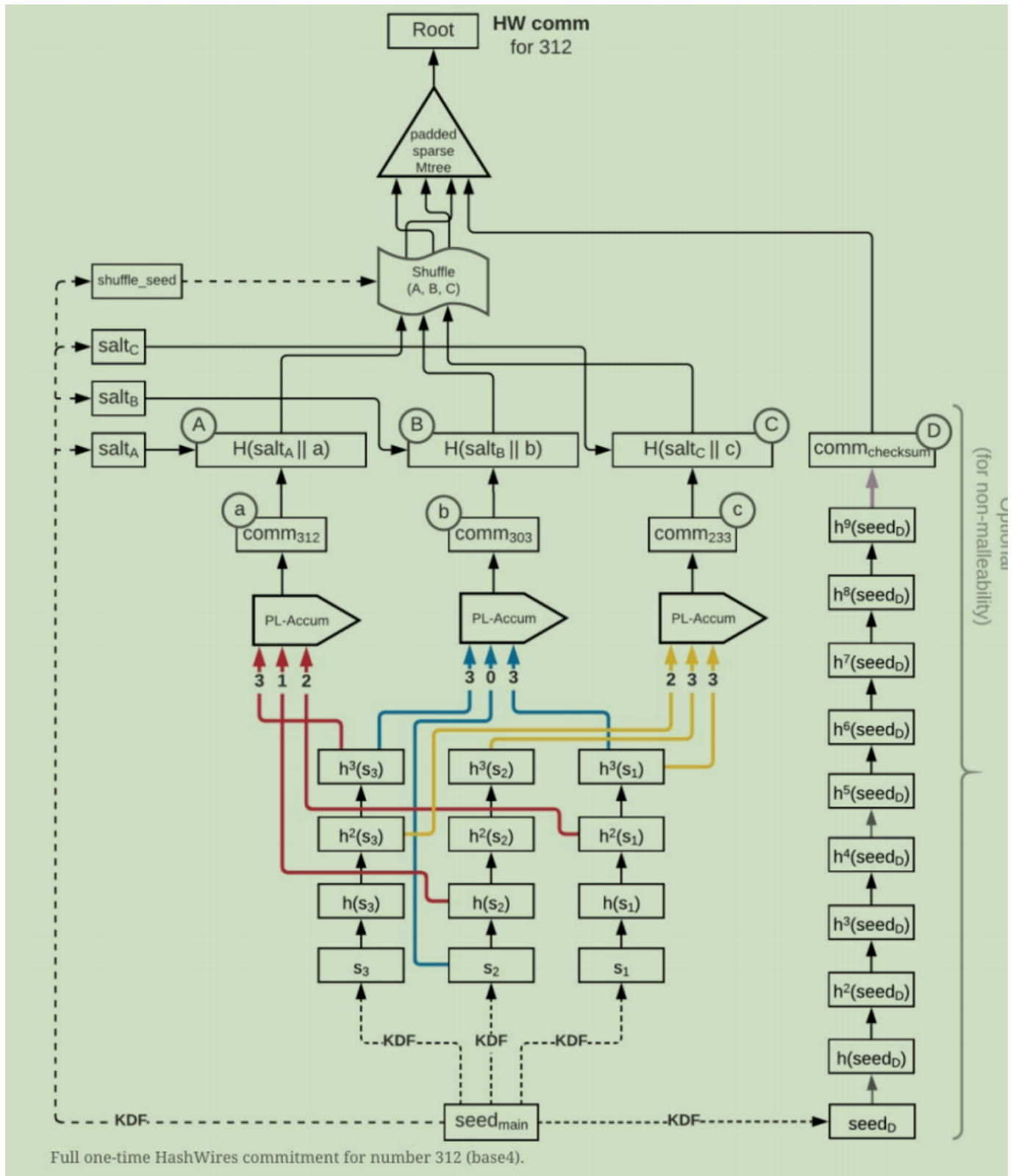
## Project-7, Try to Implement this scheme

## 1, 实验有关

这个project我并没有实现，也没有project上传，因为并不完全明白内容，也不清楚怎么实现出来。

实验的原理大致与hash 链有关，原理图示如下：





所以我给出一个hash 链的简单实现：



```

import hashlib

# 初始数据
initial_data = "Hello, World!"

# 哈希链长度
chain_length = 10

# 创建初始哈希值
current_hash = hashlib.sha256(initial_data.encode()).hexdigest()

# 创建哈希链
hash_chain = [current_hash]

# 生成哈希链
for i in range(chain_length - 1):
    current_hash = hashlib.sha256(current_hash.encode()).hexdigest()
    hash_chain.append(current_hash)

# 打印哈希链
for i, hash_value in enumerate(hash_chain):
    print(f"Block {i}: {hash_value}")

```

运行上述代码输出内容如下：

```

RESTART: C:\Users\ml.smile\Desktop\Project\impl_scheme\Project.py
Block 0: dffd6021bb2bd5b0af676290809ec3a53191dd81c7f70a4b28688a362182986f
Block 1: 2d86b9f8b1e3c767f8c886e8056b9b14ec91e4213f4898f82c3f092bb96f8ee3
Block 2: 4a113b2b08187564020e43547481e250f49a334c7adc7d80ed363825bd00bd3a
Block 3: a14c34f0c24d7d7f9956f30d10028f117fe301f65c6cf62ef33866f341e91e15
Block 4: e1ac3978ad0f7bbd2cfa8c70fbc949b5f7b2c1d6f8682d7de900a691b452078e
Block 5: 722bc9b9479d52aa14afc173b3e3672b7fcea1594724cfce76607139b312ebe2
Block 6: 8d026d5c546795ae34673e507740349ef46499e421ef197c7ed914071de08d39
Block 7: f9950cf9033f5a06933ac90522563e82856d17c6b64f724718d3e4253b569052
Block 8: 94d48fc42f4ecfad2e80c8f3fc6a4f614b95a5ff55862663dd7715e58d01f739
Block 9: e98b8bbbe54cf18cd2f1beef3626010067cb456790f48ce418dec3446c2858c7
|

```

其他部分不是很理解，所以没有完整的实现。

## Project-8, AES impl with ARM instruction

### 1, 实验内容

实验目的是在ARM架构下运行AES，需要交叉编译

交叉编译是在一个平台上生成另一个平台上的可执行代码。在非ARM 架构服务器环境下搭建ARM 的 GCC 编译环境，编译基于 ARM 架构的应用软件。交叉编译工具链是为了编译、链接、处

理和调试跨平台体系结构的程序代码。除了体系结构相关的编译选项以外，其使用方法与 Linux 主机上的 GCC 相同。搭建交叉编译环境，即安装、配置交叉编译工具链。在该环境下编译出 ARM 架构下 Linux 系统所需的操作系统、应用程序等，然后再上传到 ARM 服务器执行。

## 交叉编译器安装

### 1, 安装标准C开发环境

Ubuntu使用sudo apt-get install build-essential

### 2, 在/usr/local 下建立名为ARM-toolchain的文件夹

mkdir /usr/local/ARM-toolchain

### 3, 安装交叉编译器

```
cd /usr/local/ARM-toolchain
wget https://releases.linaro.org/components/toolchain/binaries/latest-5/aarch64-
linux-gnu/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu.tar.xzcd
/usr/local/ARM-toolchain
wget https://releases.linaro.org/components/toolchain/binaries/latest-5/aarch64-
linux-gnu/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu.tar.xz
```

也可以从网页上下载后上传到 /usr/local/ARM-toolchain 目录下, 交叉编译工具链的地址:  
[gcc-linaro-5.5.0-2017.10-x86\\_64\\_aarch64-linux-gnu.tar.xz](https://releases.linaro.org/components/toolchain/binaries/latest-5/aarch64-linux-gnu/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu.tar.xz)

解压安装包:

```
sudo apt update
sudo apt install tar xz-utils
```

进入文件所在的位置, 解压

```
tar -xf gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu.tar.xz

PATH= /usr/local/ARM-toolchain/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-
linuxgnu/bin:"${PATH}"
```

### 4, 配置环境变量

修改配置文件, 在配置文件的最后一行加入路径配置:

```
vim /etc/bash.bashrc
#Add ARM toolchain path

PATH= /usr/local/ARM-toolchain/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-
linuxgnu/bin:${PATH}"
```

实测后发现路径配置使用上面语句没有权限，加上sudo后可以进入文件但没办法修改内容，使用sudo nano /etc/bash.bashrc可以进入文件并修改。

## 5, 环境变量生效与测试

```
source /etc/bash.bashrc  
  
aarch64-linux-gnu-gcc -v
```

执行上面的命令，显示 arm-linux-gnueabi-gcc -v 信息和版本

执行后显示如下：

```
Using built-in specs.  
COLLECT_GCC=aarch64-linux-gnu-gcc  
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/aarch64-linux-gnu/9/lto-wrapper  
Target: ... (略)  
Thread model: posix  
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)
```

### 在ARM架构下编译

使用如下语句，交叉编译某个c文件，先拿一个hello.c文件试一下：

```
/usr/local/ARM-toolchain/gcc-linaro-5.5.0-2017.10-x86_64_aarch64-linux-gnu/bin/aarch64-  
linux-gnu-gcc -o Hello Hello.c
```

使用file 语句查看生成的可执行文件是什么架构下的：

**输出结果：**

```
/home/linux-nzy/Desktop/Hello: ELF 64-bit LSB executable, ARM aarch64, version 1  
(SYSV), dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, for GNU/Linux  
3.7.0, BuildID[sha1]=ab20ba9d0ce88a5f64c7dfbce6efac81a58b3613, with debug_info,  
not stripped
```

```
/home/linux-nzy/Desktop/Hello: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV),  
dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0,  
BuildID[sha1]=ab20ba9d0ce88a5f64c7dfbce6efac81a58b3613, with debug_info, not  
stripped
```

**可以看到确实是在ARM架构下**

**由此，我们已经在x86架构下使用交叉编译器编译出了ARM的可执行文件。**

并没有执行ARM下的可执行文件。

## Project-9, AES / SM4 software implementation

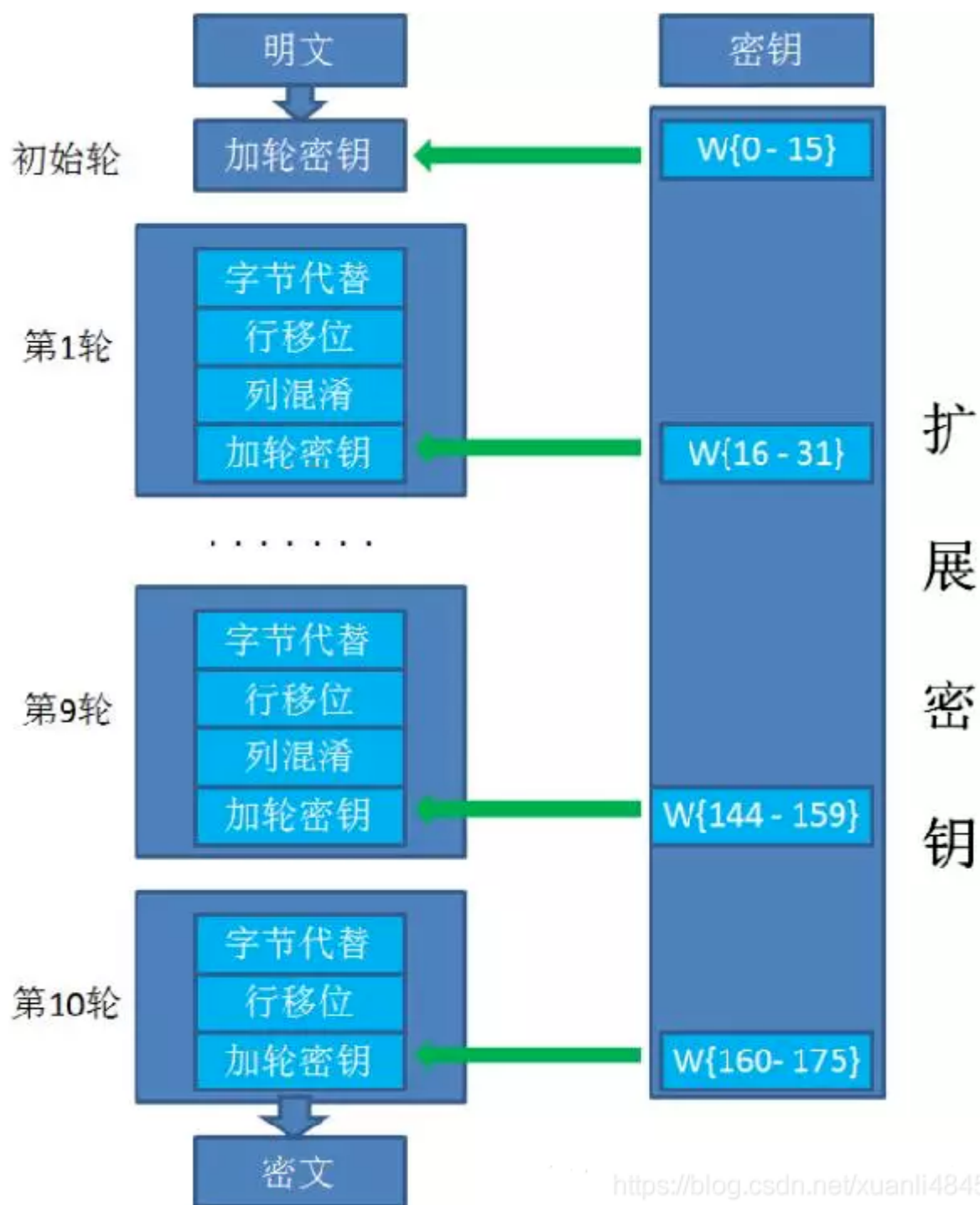
---

### 1, 实验内容

## AES

使用c++实现，是朴素的AES的实现过程。

AES的加密流程如下：



各功能的实现可以从代码中看到，能够体现整体AES流程的代码部分如下：

```

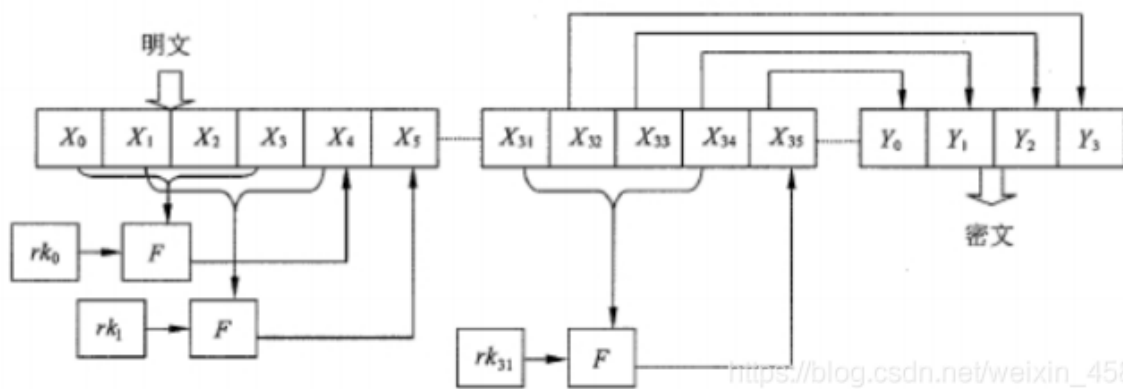
//第一轮开始
int index = 0;
key_pro = key_expression(p_key, p_sbox, p_constant, index); //第一轮轮密钥
//dump_buf(key_pro, 16);
subbyte(p_sbox, p_mes); //字节替换
dump_buf(p_mes, 16);
RowShift(p_mes); //行移位
dump_buf(p_mes, 16);
MixColumns(p_mes); //列混合
dump_buf(p_mes, 16);
key_wheel_addition(key_pro, p_mes); //密钥轮加
dump_buf(p_mes, 16);
index = 1;
//dump_buf(key_pro, 16);
for (; index < 9; index++)
{
    key_pro = key_expression(key_pro, p_sbox, p_constant, index);
    //dump_buf(key_pro, 16); //输出轮密钥
    subbyte(p_sbox, p_mes); //字节替换
    RowShift(p_mes); //行移位
    //dump_buf(p_mes, 16);
    MixColumns(p_mes); //列混合
    key_wheel_addition(key_pro, p_mes); //密钥轮加
}
index = 9;
//第十轮开始
key_pro = key_expression(key_pro, p_sbox, p_constant, index);
//dump_buf(key_pro, 16); //输出轮密钥
//subkey(p_sbox, key_pro);
subbyte(p_sbox, p_mes); //字节替换
RowShift(p_mes); //行移位
//dump_buf(p_mes, 16);
key_wheel_addition(key_pro, p_mes);
cout << "ciphertext:" << endl;
dump_buf(p_mes, 16);
return 0;

```



## sm4

sm4的加密流程大致如下：



能够体现sm4加密流程的是如下代码：

```
void sm4_encrypt(sm4_context* ctx, size_t length, unsigned char* input, unsigned char* output)
{
    uint32_t X[4+ROUND];
    uint32_t temp;

    while (length > 0) {
        GET_UINT32_BE(X[0], input, 0);
        GET_UINT32_BE(X[1], input, 4);
        GET_UINT32_BE(X[2], input, 8);
        GET_UINT32_BE(X[3], input, 12);

        for (int i = 4; i < ROUND + 4; i++)
        {
            X[i] = X[i - 4] ^ T(X[i - 3] ^ X[i - 2] ^ X[i - 1] ^ ctx->rk[i - 4]);
        }

        PUT_UINT32_BE(X[35], output, 0);
        PUT_UINT32_BE(X[34], output, 4);
        PUT_UINT32_BE(X[33], output, 8);
        PUT_UINT32_BE(X[32], output, 12);

        input += 16;
        output += 16;
        length -= 16;
    }
}
```

## 2, 运行指导

AES代码全部在AES.cpp中，其他则是sm4的头文件及源文件，AES代码可直接运行，sm4代码需要更换加密文件路径。

## 3, 运行结果

AES运行结果如下：

```
buf:
    19 A0 9A E9 3D F4 C6 F8 E3 E2 8D 48 BE 2B 2A 08
buf:
    D4 E0 B8 1E 27 BF B4 41 11 98 5D 52 AE F1 E5 30
buf:
    D4 E0 B8 1E BF B4 41 27 5D 52 11 98 30 AE F1 E5
buf:
    04 E0 48 28 66 CB F8 06 81 19 D3 26 E5 9A 7A 4C
buf:
    A4 68 6B 02 9C 9F 5B 6A 7F 35 EA 50 F2 2B 43 49
ciphertext:
buf:
    39 02 DC 19 25 DC 11 6A 84 09 85 0B 1D FB 97 32
```

sm4运行结果会保存在.txt文件中，截图如下；

```
775487a9bde51fbcca0d2d6376d04ea7dea73706a34a9a1e3298c68a
ae1cbde9defa9231acf9398110f2782683090ce9defa9231acf93981
603759e0807d59c55e7989b83c7c3c7c5062bdaf373ada8a5981b0a7
77480421835a4199a2f0005ae8afb4a23e731786867750faf07cce6
7ac0257d057e7170b8004f3451db0e5c373335f06612a737363ddd4f
5d9d25efcd8ef3aded5a7c1b84fb1d586e168493acb3b06ee39e4eaf
f8462fb5be55c7d5f732a4552d0229b1cee2b9e1aa665726fc574cf5
f82404ca5cc0d997e52533cf6192ee69306448c81ba32efeb63457ad
4946d8e0f062877add1eb8c21a62abf7825ff7743e631994cf265f7f
4ae2c052a698504e9f1bdf5cc8efd7740681201bd85c271083055f99
```

Project-10, report on the application of this deduce technique in Ethereum with ECDSA

# 运行指导

报告在Project10文件夹中，可以直接打开。

## Project-11, impl sm2 with RFC6979

### 1, 实验内容

- RFC6979中对sm2的要求如下：
  1. 无需外部随机数源：RFC 6979要求生成SM2签名所需的随机数不依赖于外部的随机数源。即在签名过程中，不需要依赖于真正的随机数生成器。
  2. 确定性：RFC 6979要求相同的私钥和消息输入时，生成的随机数必须是确定性的。这意味着对于给定的私钥和消息，每次生成的随机数都应该相同，以确保可重现性。
  3. 不可预测性：RFC 6979要求生成的随机数必须是不可预测的，以避免潜在的安全漏洞。即使攻击者可以知道私钥和消息，也不能推断出随机数的值。
  4. 安全性：RFC 6979对生成的随机数的安全性提出了要求。它使用了一种称为Hash-DRBG的伪随机数生成器来计算随机数，并确保其均匀性、不可预测性和不可操纵性。

#### sm2\_myself

sm2\_myself是自己手动实现的sm2，其中椭圆曲线参数的选取是：

```
p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFF
a = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFC
b = 0x28E9FA9E9D9F5E344D5A9E4BCF6509A7F39789F515AB8F92DDBCBD414D940E93
q = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7203DF6B21C6052B53BBF40939D54123

Px=0x32C4AE2C1F1981195F9904466A39C9948FE30BBFF2660BE1715A4589334C74C7
Py=0xBC3736A2F4F6779C59BDCEE36B692153D0A9877CC62A474002DF32E52139F0A0
```

sm2的关键部分是签名和验证签名，这两个函数如下：

```

def signature(ENTL_A, ID_A, d_A, M, k):
    G=(Px,Py)
    P_A=point_mul(G,d_A)
    x_A=P_A[0]#分别取出PA的x and y
    y_A=P_A[1]
    ZA=Precompute(ENTL_A, ID_A, x_A, y_A)
    M1=ZA+M#传入M需要也为字符串
    e=int(Hv(M1),16)#e是十进制整数
    #print('e',e)

    #k = random.getrandbits(256) % q
    #将随机生成的k改进为RFC6979生成
    kG=point_mul(G,k)#计算k*G
    x1=kG[0]
    y1=kG[1]
    #print('x1',x1)
    r=(e+x1)%q
    #print('r',r)
    s=((k-r*d_A+q)*invert((1+d_A),q))%q
    return (r,s),P_A

def verify_signature(sig,M,P_A,ENTL_A, ID_A):
    G=(Px,Py)
    ZA=Precompute(ENTL_A, ID_A, P_A[0], P_A[1])
    M1=ZA+M
    e=int(Hv(M1),16)
    #print('e1',e)
    r=sig[0]
    #print('r',r)
    s=sig[1]

    t=(r+s)%q
    #(x1,y1)=point_add(point_mul(s,G),point_mul(t,P_A))
    sG=point_mul(G,s)
    tP=point_mul(P_A,t)
    (x1,y1)=point_add(sG,tP)
    #print('x1',x1)
    R=(e+x1)%q
    #print('R:',R)
    if(R==r):
        print('签名验证通过')
    else:
        print('签名验证失败')
    return (s,t),P_A

```

## sm2 with RFC6979

加入RFC6979后，影响了随机数k的生成，RFC6979下生成k的过程如下：

```

def generate_k(mes,d):

    m=sm3_hash(mes.encode()).encode()

    h = sha256(m).digest()

    xlen = (p.bit_length() + 7) // 8
    hlen = len(h)
    v = b'\x01' * hlen
    k = b'\x00' * hlen

    k = hmac_sha256_kdf(k, v + b'\x00' + d.to_bytes(xlen, 'big') + h, hlen)
    v = hmac_sha256_kdf(k, v, hlen)
    k = hmac_sha256_kdf(k, v + b'\x01' + d.to_bytes(xlen, 'big') + h, hlen)
    v = hmac_sha256_kdf(k, v, hlen)

    while True:
        t = b''
        while len(t) < xlen:
            v = hmac_sha256_kdf(k, v, hlen)
            t += v

        k_candidate = int.from_bytes(t[:xlen], 'big')
        if 1 <= k_candidate < q:
            return k_candidate

    k = hmac_sha256_kdf(k, v + b'\x00', hlen)
    v = hmac_sha256_kdf(k, v, hlen)

```

## 2, 运行指导

直接运行sm2\_RFC6979即可，sm2\_myself作为库函数导入

## 3, 运行结果

运行程序结果如下：

```

==== RESTART: C:\Users\Mr. smile\Desktop\Project11_sm2_RFC6979\sm2_RFC6979.py ====
k_with_RFC6979 : 234779190083945406335000449864418549188460042264070278538977876
53413784195997
签名验证通过

```

**Project-12, verify the above pitfalls with proof-of-concept code**

## 1, 实验内容

实验内容旨在证明sm2,ECDSA,Shnorr签名方案中都存在的一些问题。

pitfalls	
Leaking $k$ leads to leaking of $d$	
Reusing $k$ leads to leaking of $d$	
Two users, using $k$ leads to leaking of $d$ , that is they can deduce each other's $d$	
Malleability, e.g. $(r, s)$ and $(r, -s)$ are both valid signatures, lead to blockchain network split	
Ambiguity of DER encode could lead to blockchain network split	
One can forge signature if the verification does not check $m$	
Same $d$ and $k$ with ECDSA, leads to leaking of $d$	

- 实验的过程是首先手动实现了上述三种签名方案，分别作为 ECDSA\_myself,Shnorr\_myself,sm2\_myself
- 然后给出了ECDSA中上述pitfall的证明（使用proof-of-concept code）
- 由于三种方案证明漏洞的办法相似，所以代码main.py 中针对ECDSA做了证明。
- 下面给出ECDSA实现的关键部分（签名，验签）：

```

def signature(mes,d):
    #k=random.getrandbits(256)%q

k=61052478844650362117101898807367092637533363199480174276305957447633440843510
    R=point_mul(G,k)
    r=R[0]%q
    e=sm3_hash(mes.encode())
    e=int(e,16)
    #print('e',e)
    s=(invert(k,q)*(e+d*r))%q
    return (r,s)

def verify_sig(mes,Q,sig):
    e=sm3_hash(mes.encode())
    e=int(e,16)
    r=sig[0]
    s=sig[1]
    w=invert(s,q)
    ewG=point_mul(G,e*w)
    #print('e*w',e*w)
    rwP=point_mul(Q,r*w)
    #print('r*w',r*w)
    (r1,s1)=point_add(ewG,rwP)
    if r1==r:
        print('签名验证成功')
    else:
        print('签名验证失败')

```

## verify pitfalls

- leaking k lead to leaking of d:

```
def ECDSA_leak_k(mes):
    q=ecdsa.q
    d,Q=ecdsa.key_gen()
    print('真正的d是',d)

k=610524788446503621171018988073670926375333631994801742763059574476334408435
10

    (r,s)=ecdsa.signature(mes,d)
    print('r',r)
    e=ecdsa.sm3_hash(mes.encode())
    e=int(e,16)
    print('e',e)
    d_re=((s*k-e)*invert(r,q))%q
    print('根据k还原出d是',d_re)
    if(d==d_re):
        print('leaking k lead to leaking of d')
    return d_re
```



- Two users can deduce each other's d

```
def ECDSA_deduce_other_d(mes1,mes2):
    q=ecdsa.q

    k=610524788446503621171018988073670926375333631994801742763059574476334408435
    10

    d,Q=ecdsa.key_gen()
    print('真正的d',d)
    #用户一使用k与用户二的消息推断用户二的密钥d
    (r2,s2)=ecdsa.signature(mes2,d)#(r2,s2)是用户一可以获知的用户二的签名
    e=ecdsa.sm3_hash(mes2.encode())
    e=int(e,16)
    d_re_1=((s2*k-e)*invert(r2,q))%q
    print('d_re_1',d_re_1)
    if(d_re_1==d):
        print('user 1 can deduce k of user 2')
    #用户二也可推测用户一的d
    (r1,s1)=ecdsa.signature(mes1,d)
    e=ecdsa.sm3_hash(mes1.encode())
    e=int(e,16)
    d_re_2=((s1*k-e)*invert(r1,q))%q
    print('d_re_2',d_re_2)
    if(d_re_2==d):
        print('user 2 can deduce k of user1')
    return None
```

还有其他几个pitfalls如果都贴上会显得报告冗长，这里只展示一下报告的思路和两个例子，其他不再赘述，可以在main.py中看到

## 2, 运行指导

直接运行代码即可，有些函数调用被注释掉了，需要恢复才能调用

## 3, 运行结果

```
真正的d是 17557263227441064869951035529758010784638612597823094572249304185318165423633
r 16185797599380199915883646014937883960459278700342043126416454818108589680873
e 83199874723710846093074413087829055388694010089693699555710547178428103626179
根据k还原出d是 17557263227441064869951035529758010784638612597823094572249304185318165423633
leaking k lead to leaking of d
真正的d是 17557263227441064869951035529758010784638612597823094572249304185318165423633
还原出d是 17557263227441064869951035529758010784638612597823094572249304185318165423633
resuing k lead to leaking of d
真正的d 17557263227441064869951035529758010784638612597823094572249304185318165423633
d_re_1 17557263227441064869951035529758010784638612597823094572249304185318165423633
user 1 can deduce k of user 2
d_re_2 17557263227441064869951035529758010784638612597823094572249304185318165423633
user 2 can deduce k of user1
s的逆 111561231435883526292942817581348568336639679063296180555842916895541016062355
r 16185797599380199915883646014937883960459278700342043126416454818108589680873
fake_r 111561231435883526292942817581348568336639679063296180555842916895541016062355
更正后的ew 114483247241091117308806274106849933123710547434802460539511153020393422332520
w 16185797599380199915883646014937883960459278700342043126416454818108589680873
签名验证成功
真正的d是 17557263227441064869951035529758010784638612597823094572249304185318165423633
还原出d是 17557263227441064869951035529758010784638612597823094572249304185318165423633
using same d and k leads to leaking of d
|
```

每个原因的上方是该函数执行的内容

比如 (leaking k lead to leaking of d) , 其上方的输出就是证明过程。

## Project-13, Implement the above ECMH scheme

### 1, 实验内容

ECMH:

基于椭圆曲线的因子分解算法, 当期待分解的因子不太大时, 可以快速分解因子。

- 它是一种概率算法, 即不能保证在有限时间内找到因子。
- 它适用于相对较小的合数, 对于大素数较少有效。
- 它具有较好的并行性, 可以将计算任务分配给多个处理单元进行加速计算。

能够体现ECMH流程的代码部分如下:

```

B=randprime(2**128-1,2**128-1)
#print('B',B)
k=randint(1,q-1)
P=point_mul(G,k)#计算P=k*基点
t1=time()
#counter=1
while True:
    for j in range(1,B):
        Q=point_mul(P,j)
        r=Q[0]%N
        d=gcd(r,N)
        if(d!=1 and d!=N):
            print("find",d)
            break
        continue
    if(d!=1 or d!=N):
        break
    B=(B*2)%N
    print('B',B)

```

## 关于ECMH的分解效率

- 对于较小的合数，ECM-H方法通常表现出很好的效果。它可以有效地找到该合数的非平凡因子，并完成因子分解过程。这是因为对于较小的合数，相对较少的计算资源和时间就足以找到合适的非平凡因子。
- 然而，随着待分解因子的增大，ECM-H方法的效果显著降低。尤其是当待分解因子是一个大素数时，ECM-H方法通常不会表现出很高的效率。这是因为大素数的阶很大，寻找非平凡因子的概率变得非常低，可能需要进行大量的计算才能找到其中一个因子。
- 对于较大的合数或大素数的因子分解，一般需要采用其他更适合的算法，如基于整数分解算法的方法（如QS、GNFS等）。这些算法的效果通常比ECM-H方法更好，特别是在大素数因子的分解上。
- 在实际应用中，ECM-H方法的效率通常在因子大小达到几十位数（比如50位或60位）左右时开始明显下降。在这个范围内，ECM-H方法可能仍然可以有效地找到因子并完成因子分解过程。
  - 然而，随着因子继续增大，ECM-H方法的效率迅速降低。当因子达到几百位数或更大时，ECM-H方法的效率往往会变得非常低下。

## 2, 运行指导

直接运行代码即可，默认输出待分解的大整数，使用ECMH的分解结果和测试时间

### 3, 运行结果

```
find 11213
ECMH time: 0.00799870491027832 s
i 43
time: 0.011548757553100586 s
```

## Project-14, Implement a PGP scheme with SM2

---

### 1, 实验内容

PGP协议包括非对称加密和对称加密，这里sm2应用于非对称加密，而使用AES作为对称加密的算法。

AES加密部分(明文的加密):

```
def encrypt_message(nclass,m):
    message_byte=bytes(m,encoding='utf-8')
    encrypt_mes=nclass.crypt_AES.encrypt(message_byte)
    return encrypt_mes
```

体现主要内容的代码部分如下:

```

def __init__(nclass,mes):#
    #mse is string,not byte stream
    mes=nclass.padding_func(mes)
    print('padded mes:',mes)
    nclass.iv=get_random_bytes(16)#生成初始IV
    nclass.key_AES=get_random_bytes(16)#生成AES密钥
    nclass.crypt_AES=AES.new(nclass.key_AES, AES.MODE_CBC, nclass.iv)

    nclass.private_key =
'0bedf1f07b32b1168dfd5f64862d85ea647f91edb039da1f27f8fee92b928dda'#设置ms2私钥
    #nclass.private_key = sm2.CryptSM2().generate_private_key()
    print('private_key:',nclass.private_key)
    nclass.public_key =
'6ae77fcf2fe923fd888582cf3151d9a1bc2a5ca80064e0d07f6747c78e3fa4554cbef52f11605a495821a
cdb28e9314444aee895f8ca05495cbfe861a41681cb'
    #nclass.public_key = nclass.private_key.public_key()#用私钥获取公钥
    print('public_key:',nclass.public_key)
    nclass.sm2_crypt = sm2.CryptSM2(public_key=nclass.public_key,
private_key=nclass.private_key)
    nclass.message=mes

```

通信发送部分代码不再体现。

## 2, 运行指导

直接运行代码即可.

## 3, 运行结果

运行代码结果:

```

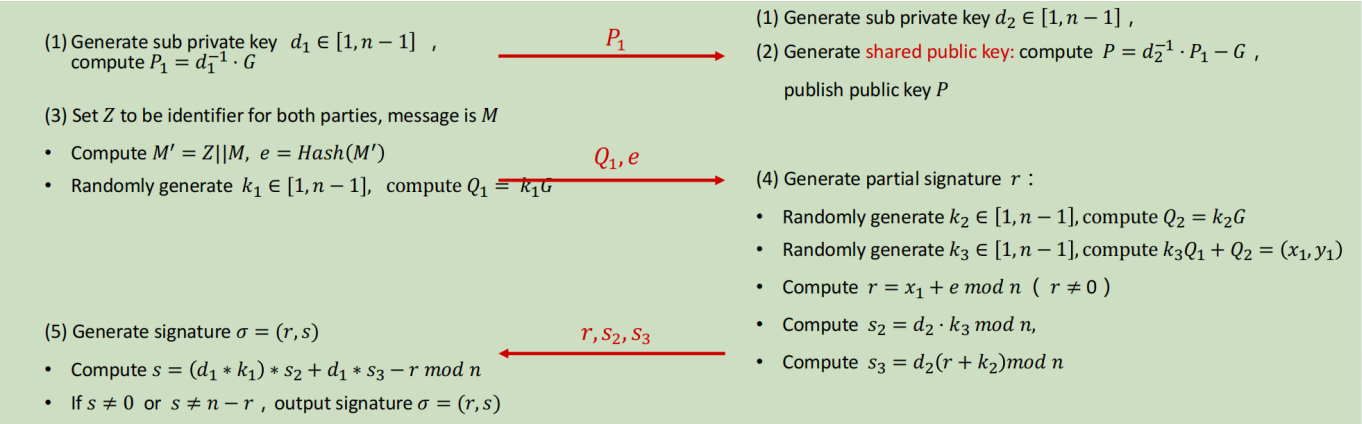
length_mes 36
padded mes: Shadow, and light down, Sunny and wind000000000000
private_key, 0bedf1f07b32b1168dfd5f64862d85ea647f91edb039da1f27f8fee92b928dda
public_key: 6ae77fcf2fe923fd888582cf3151d9a1bc2a5ca80064e0d07f6747c78e3fa4554cbe
f52f11605a495821acdb28e9314444aee895f8ca05495cbfe861a41681cb
0WhVC/sob2nE5JOBLCSzYcyN94XD3K+9xaTbTatXOnVWWBX8Y1BxwrIgPfk/rchJeo+pVkwaeMkMWKjg
g1OYG1h1G4jM14Hpj1xA4X3x7LYEC45QkMENK3cYacn4BhKvUd6P/K9gG6hbvNBmymC9FCF67TleNk2a
px7J3dC0vn0WJzedELkZcvucbQkZfMKsPEDYSSx46TTMy1GhwH0SwQ==

```

# Project-15, implement sm2 2P sign with real network communication

## 1, 实验内容

主要是在通信中实现sm2签名过程，签名原理如下：



下面给出能够体现上述原理的关键部分代码

UserA:

```

# 生成子私钥 d1
d1 = randint(1,q-1)
P=(Px,Py)

# 计算P1 = d1^(-1) * 生成元
P1 = point_mul(invert(d1,p),P)
x,y = hex(P1[0]),hex(P1[1])

# 向客户2发送P1
addr = (host, port_UserB)
s.sendto(x.encode('utf-8'), addr)
s.sendto(y.encode('utf-8'), addr)

###
m = "This is the last rose of summer"
m = hex(int(binascii.b2a_hex(m.encode()).decode(), 16)).upper()[2:]
User_A = "User_Alice"
User_A = hex(int(binascii.b2a_hex(User_A.encode()).decode(), 16)).upper()[2:]
ENTL_A = '{:04X}'.format(len(User_A) * 4)
ma = ENTL_A + User_A + '{:064X}'.format(a) + '{:064X}'.format(b) +
'{:064X}'.format(Px) + '{:064X}'.format(Py)
print('ma:',ma)
N = sm3_hash(ma.encode())
e = sm3_hash((N + m).encode())

# 生成随机数k1
k1 = randint(1,q-1)

# 计算Q1 = k1 * G
Q1 = point_mul(k1,P)
x,y = hex(Q1[0]),hex(Q1[1])

# 向客户2发送Q1,e
s.sendto(x.encode('utf-8'),addr)
s.sendto(y.encode('utf-8'),addr)
s.sendto(e.encode('utf-8'),addr)

# 从客户2接收r,s2,s3
r,addr = s.recvfrom(1024)
r = int(r.decode(),16)
s2,addr = s.recvfrom(1024)
s2 = int(s2.decode(),16)
s3,addr = s.recvfrom(1024)
s3 = int(s3.decode(),16)

# 计算s_s(避免与socket混淆)
s_s=((d1 * k1) * s2 + d1 * s3 - r)%q
if s_s!=0 or s_s!= n - r:
    print("Sign:")
    print((hex(r),hex(s_s)))

```

UserB:

```
G=(Px,Py)#生成元(在UserA中定义P为生成元，此处定义G避免与公钥P混淆定义)
# 生成子私钥 d2
d2 = randint(1,q)

# 从客户1接收P1=(x,y)
x,addr = s.recvfrom(1024)
x = int(x.decode(),16)
y,addr = s.recvfrom(1024)
y = int(y.decode(),16)

# 计算共享公钥P
P1 = (x,y)
P = point_mul(invert(d2,p),P1)

P = point_add(P,(Px,-Py))

# 从客户1接收Q1=(x,y)与e
x,addr = s.recvfrom(1024)
x = int(x.decode(),16)
y,addr = s.recvfrom(1024)
y = int(y.decode(),16)
Q1 = (x,y)
e,addr = s.recvfrom(1024)
e = int(e.decode(),16)

# 生成随机数k2,k3
k2 = randint(1,q-1)
k3 = randint(1,q-1)

# 计算Q2 = k2 * G
Q2 = point_mul(k2,G)

# 计算(x1,y1) = k3 * Q1 + Q2
x1,y1 = point_mul(k3,Q1)
x1,y1 = point_add((x1,y1),Q2)
r =(x1 + e)%q
s2 = (d2 * k3)%q
s3 = (d2 * (r+k2))%q

# 向客户1发送r,s2,s3
s.sendto(hex(r).encode(),addr)
s.sendto(hex(s2).encode(),addr)
s.sendto(hex(s3).encode(),addr)
```



## 2, 运行指导

### 3, 运行结果

```
C:\Users\Mr. smile>C:\Users\Mr. smile\Desktop\Project15_sm2_2Psign\UserA.py
ma: 0050557365725F416C696365FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC2C28E9FA9E9D9F5E3448DE6470264F
98A48F8C70A9D578D948C539EBD10162E5DB32C4AE2C1A86E372FEF09E0A81FFB9602486073CEE0D73FA7575F085051561DCA78F61D07314897B263C
4E84D2F910B0863896E661D4EBDABAF41B6E87DADA36
Sign:
('0x522bc23105445c4ad058a4cf56c992c1ae575aae45050f6743a42f398f352209', '0xd370ac4b26ef6eda4c34854378e27172ea72a1ed16f760
9994809ad6ca672e7')
```

```
C:\Users\Mr. smile>C:\Users\Mr. smile\Desktop\Project15_sm2_2Psign\UserB.py
close
```

## Project-16, implement sm2 2P decrypt with real network communication

## 1, 实验内容

(1) Generate sub private key  $d_2 \in [1, n - 1]$



A horizontal line with an arrow pointing to the right, labeled  $T_1$  above it.

 $T_2$ 

- If  $u = C_3$ , output  $M''$

UserA:

```

while True:
    d1=random.getrandbits(256)%q
    #1
    mes,addr=s.recvfrom(4096*16)
    print("Received message from :",addr)
    d2=int(mes.decode(),10)

    private_key,public_key=Key_Gene(d1,d2)
    M='RuoYan'#M定义为字符串类型
    private_key,public_key=Key_Gene(d1,d2)
    C_dict,k=Encrypt(public_key,M)
    klen=k.bit_length()
    T1=point_mul(C_dict['C1'],invert(d1,q))
    #2
    s.sendto(str(T1).encode(),addr)
    #3
    mes,addr=s.recvfrom(4096*16)
    print("Received message from :",addr)
    T2=tuple(mpz(x) for x in eval(mes.decode()))
    #T2=int(T2,10)
    Decrypt(T2,C_dict['C1'],C_dict['C2'],C_dict['C3'],klen)
    break
s.close()

```

UserB:

```

while True:
    d2=random.getrandbits(256)%q
    #1
    s.sendto(str(d2).encode(),addr)
    #2
    mes,addr=s.recvfrom(4096*16)
    print("Received message from :",addr)

    T1=tuple(mpz(x) for x in eval(mes.decode()))
    T2=point_mul(T1,invert(d2,q))
    #3
    s.sendto(str(T2).encode(),addr)
    break
s.close()

```

## 2, 运行指导

可以使用命令行, 先运行Alice, 再运行Bob。

sm2\_myself作为库函数导入

如果在两台设备进行通信，需要修改IP，一台设备则不用。

### 3，运行结果

Alice：

```
Received message from : ('2.0.0.1', 10001)
Received message from : ('2.0.0.1', 10001)
解密成功
RuoYan
```

Bob：

```
C:\Users\Mr. smile>C:\Users\Mr. smile\Desktop\Project16_sm2_2P_decrypt\Bob.py
Received message from : ('2.0.0.1', 12345)
```

## Project-17，比较Firefox和谷歌的记住密码插件的实现区别

---

### 运行指导

---

在project17中，可以直接打开报告。




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

---

### 1，实验内容

刚开始的尝试是先下载[electrum]([Electrum Bitcoin Wallet](#)) 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。

文件(F) 钱包(W) 视图(V) 工具(T) 帮助(H)

 历史  发送  接收  地址 (A)  通道  选币 (I) 

30 个地址

类型	地址	标签
收款	bc1qhcmtrkyes0ky8409yfd8jcuvfwey2k2jvkjnm8	
收款	bc1q0u3dd37au8tnhgm8nrmwn4n1jz6m1c32ty6hxx	
收款	bc1qkta9k21u42684qe99j5ezc0uk6acwemm94w81	
收款	bc1qgr6x9ewrpwyn3uc71k8z3xtddu171gx7qxjd4z	
收款	bc1quj5ktuhcmu89xehh2r29w3md1pdw29w4vddj0n	
收款	bc1qfw3dvmfwzm06du0ar7cccfmm9zfhyhzd52ff1c	
收款	bc1q6v83d8tqxem5xxcuak5m2vujwefzau2ukhtzg3	
收款	bc1qnqau3ttqkc7yuzwrj4vdengztejdmae8u32rfa	
收款	bc1q86ukrqqs2dau2qsv8c4fwwczdqkvguqtnc8fa	
收款	bc1qyu82gm5fpffrr3g49un63yas4wmfspr9xpsky4	
收款	bc1qwde26sga52fft5gpc7puprhzyryqmvn9rp6d8t	
收款	bc1q3zyuf74az444wx7kgda8y8qgw7e95r5t6ck7k3	
收款	bc1qqmesd6544wh8q6uey02j79kq141eg582qw9qje	
收款	bc1qmf1fc0yjn6xpzad0p72683t1r8sxchfy1mdwk9	
收款	bc1qmkaz5n2z8rmy52qnx53f0jsp91rar91cu0uqg4	
收款	bc1qvfezxxntdv0hh5tzjzxd4gsqkqwve6m9qzzdcj	
收款	bc1q6jtgmr9w2yuuw6fre6qdtYu75hqwjwu2kwqtd9	
收款	bc1qxm3qmsaxzpvM78ehqy65ze5uhg4rdhkvacfmj1	
收款	bc1q7hwtnt8d9wv1lftn5v5d9v9v7c75tm0m	

之后可以通过[水龙头]

1. <https://coinfaucet.eu/en/btc-testnet/>
2. <https://testnet.coinfaucet.eu/>
3. <http://tpfaucet.appspot.com/>
4. <http://kuttler.eu/bitcoin/btc/faucet/>

如上是一些可能可用的水龙头，获取比特币测试

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



Open Source JavaScript Client-Side Bitcoin Wallet Generator

Single Wallet	Paper Wallet	Bulk Wallet	Brain Wallet
Vanity Wallet	Split Wallet	Wallet Details	

Generate New Address

Print

Bitcoin Address	Private Key
-----------------	-------------

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](#)) 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：（mv15LxNVtZ2uaNCmzZ8RL4RzL9oNUPQYp7）

可以看到上面的余额情况：

收到	已发送	余额
0.01759267 BTC	0.0 比特币	0.01759267 BTC

再通过查询一笔交易的txid可以查到该交易。

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txid：

f96ba3e152b89a7318f88566d192ef37fcd44dd25b5305b5e1dedcd6b2c7861e

我们可以看到交易的信息：

高级详细信息

区块哈希	00000000000001a936af25620e16f54ff15939c02c4b242c2950af8d83f2a83
块高度	2,469,856
交易指数	3 ( <a href="#">永久链接</a> )
大小	228 字节
虚拟尺寸	147 V 字节
锁定时间	2469855
版本	2

小接口调用

大接口调用

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)
```

该爬虫将解析后的文本放入output after parse 文本中，部分内容如下：

Block Hash

0000000000000001a936af25620e16f54ff15939c02c4b242c2950af8d83f2a83  
See Block

Block Height

2,469,856

Transaction Index

3  
([permalink](#))

Size

228 bytes

Virtual Size

147 vbytes

即该笔交易的信息

## Project-19, forge a signature to pretend that you are Satoshi

---

### 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：

```
P_Satoshi=
(26877259512020005462763638353364532382639391845761963173968516804546337027093,4856694
4205781153898153509065115980357578581414964392335433501542694784316391)

sig_Satoshi=
(41159732757593917641705955129814776632782548295209210156195240041086117167123,
57859546964026281203981084782644312411948733933855404654835874846733002636486)
```

伪造签名的原理如下：

- $\sigma = (r, s)$  is valid signature of  $m$  with secret key  $d$
- If only the hash of the signed message is required
- Then anyone can forge signature  $\sigma' = (r', s')$  for  $d$
- (Anyone can pretend to be someone else)
- Ecdsa verification is to verify:
- $s^{-1}(eG + rP) = (x', y') = R', r' = x' \bmod n == r ?$
- To forge, choose  $u, v \in \mathbb{F}_n^*$
- Compute  $R' = (x', y') = uG + vP$
- Choose  $r' = x' \bmod n$ , to pass verification, we need
- $s'^{-1}(e'G + r'P) = uG + vP$ 
  - $s'^{-1}e' = u \bmod n \rightarrow e' = r'uv^{-1} \bmod n$
  - $s'^{-1}r' = v \bmod n \rightarrow s' = r'v^{-1} \bmod n$
- $\sigma' = (r', s')$  is a valid signature of  $e'$  with secret key  $d$

代码中实现该部分内容的代码如下：



```
def signature_forge(P_Satoshi):
    u=random.getrandbits(256)%q
    v=random.getrandbits(256)%q
    R_forge=point_add(point_mul(G,u),point_mul(P_Satoshi,v))
    Rx_forge=R_forge[0]
    e_forge=(Rx_forge*u*invert(v,q))%q
    s_forge=(invert(v,q)*Rx_forge)%q
    print('伪造的签名是',(Rx_forge,s_forge))
    return (Rx_forge,s_forge),e_forge
```

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：

```
def verify_sig(e,Q,sig):
    #e=sm3_hash(mes.encode())
    #e=int(e,16)
    r=sig[0]
    s=sig[1]
    w=invert(s,q)
    ewG=point_mul(G,e*w)
    #print('r',r)
    rwP=point_mul(Q,r*w)
    (r1,s1)=point_add(ewG,rwP)
    if r1==r:
        print('签名验证成功')
    else:
        print('签名验证失败')
```

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。

## 2，运行指导

直接运行代码即可，能够输出伪造的签名，以及该伪造签名是否通过验证。

## 3，运行结果

运行程序结果如下：

```
= RESTART: C:\Users\Mr.smile\Desktop\Project19_forge_Satoshi\forge_signature.py
伪造的签名是 (mpz(73745170883181049929796213540689201243845985571299337273824755
591542968204827), mpz(8367928926648359275573974790948948797602182108516651138987
2608733263517603112))
签名验证成功
```

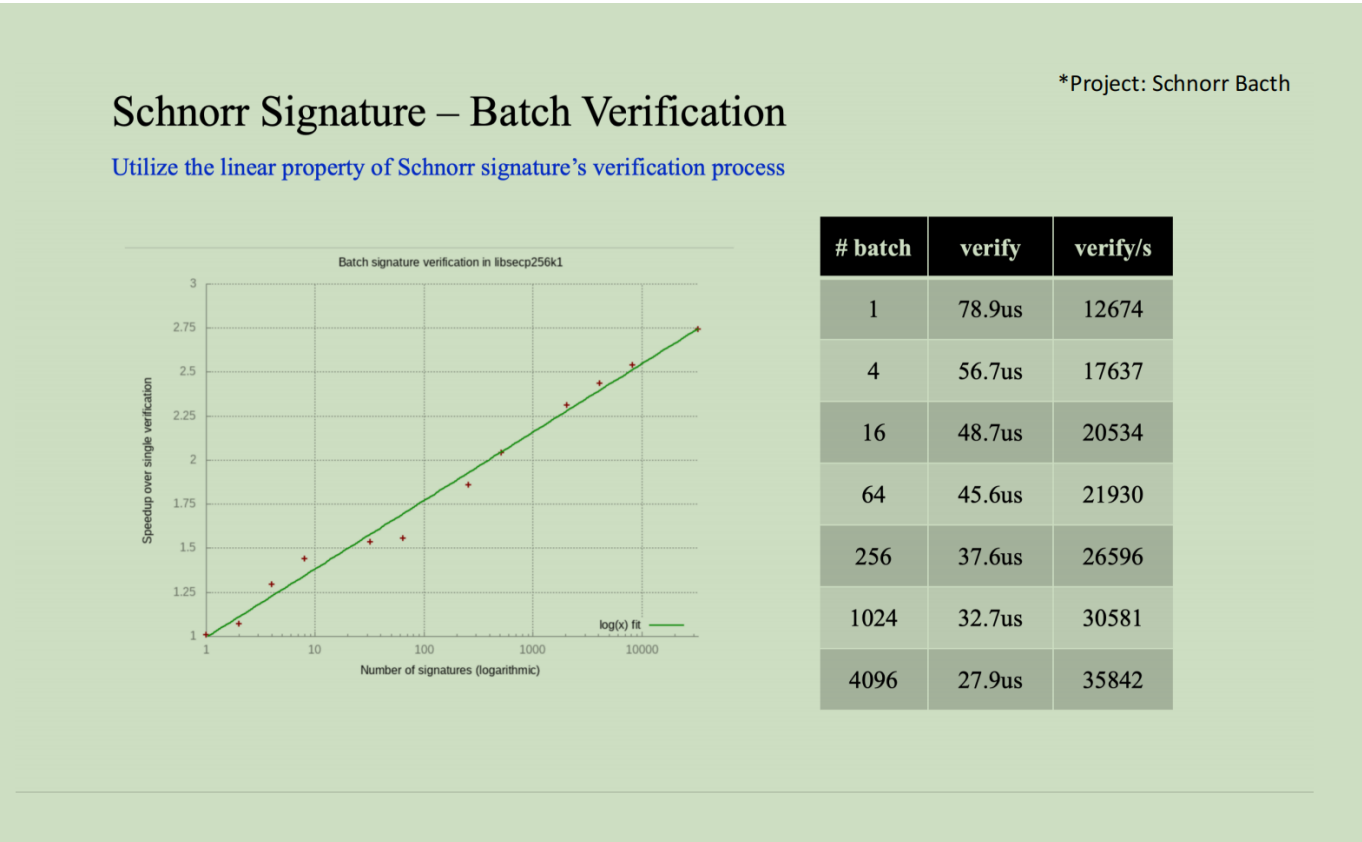
# Project-20, ECMH-POC

与上述ECMH重复，不再赘述

# Project-21, Schnorr Bacth

## 1, 实验内容

实现Schnorr签名的批量验证



Schnorr signature batch verify的原理如下：

### Schnorr Signature – Batch Verification

Utilize the linear property of Schnorr signature’s verification process

- Recall Schnorr signature’s verification:  $sG = (k + ed)G = kG + edG = R + eP$
- Batch verification equation is :
  - $(\sum_{i=1}^n s_i) * G = (\sum_{i=1}^n R_i) + (\sum_{i=1}^n e_i * P_i)$
  - Attacker can forge signature to pass the batch verification

代码实现中，batch verify部分的实现：

```
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:

```
== RESTART: C:\Users\Mr. smile\Desktop\Project21_Schnorr_Batch\
input Maksdjhf
input Mqlkejrfnzsbdgvs
input Mqkjwehnr f jknzd
input Madskhrh jf qwentfq
签名验证成功
time: 0.037055253982543945 s
```

Index=4 :

- single verify time\*Index:

```
===== RESTART: C:\Users\Mr. smile\Desktop\Schnorr_m
签名验证成功
n倍的签名验签时间: 0.09681987762451172
```

Index=8 :

- Batch verify:

```
== RESTART: C:\Users\Mr. smile\Desktop\Project21_Schnorr_Batch\  
input Msleajktwetg  
input Masfdjklgq  
input Msfdkljghksfetgwe  
input Msfjhtkgkfesn  
input Mwqreuitoywe  
input Msfdkjhgjkaz  
input Masrkuhtqker  
input Masdfkjthgqw  
签名验证成功  
time: 0.07157206535339355 s
```

Index=8 :

- single verify time\*Index:

```
===== RESTART: C:\Users\Mr. smile\Desktop\Schnorr_myself.  
签名验证成功  
n倍的签名验证时间: 0.24407958984375
```

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。

## Project-22, research report on MPT

### 运行指导

报告在Project22中，直接打开即可。