创新创业课程项目

甘 果

201900460058

Git仓库的网页地址：https://github.com/TIANMUERYA/GUOGUO.git

1. 小组成员：甘果

git账户名称：TIANMUERYA

2. 所做项目：（1）名称：Merkle tree
简介：Merkle proof
完成人:甘果

（2）名称：ECDSA
简介：PoC impl of the following pitfall
完成人：甘果

3. 清单：（1）完成的项目：①Merkle tree
②ECDSA

（2）未完成的项目：其它

（3）有问题的项目及问题：SM3实现，如何优化

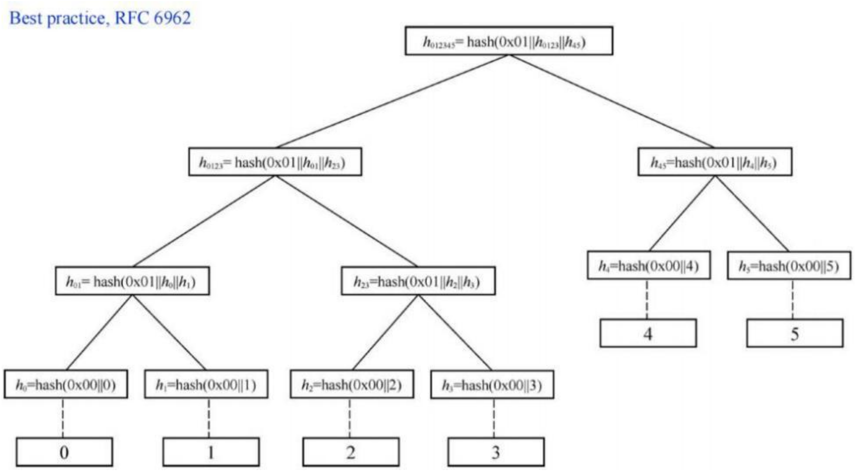
目录

Merkle proof：

1.Implement Merkle Tree as of RFC 6962 in any programming language that you prefer：

首先简要介绍一下 Merkle Tree，Merkle Tree 具有以下特征：

1. Merkle Tree 一般是二叉树的形式。
2. Merkle Tree 的叶子节点的 value 是数据集合的单元数据或者单元数据 HASH。
3. Merkle Tree 的中间节点的 value 是其所有子节点 value 的 HASH。



在代码实现之前，根据 RFC 6962 文档的内容，Merkle Tree 的实现具有以下几点要求：
要求一：在进行节点的 Hash 之前，要根据节点类型的不同加入不同的前缀：
叶子节点：前缀 0x01;
中间节点：前缀 0x00。
要求二：在构造 Merkle Tree 结构之前，需要将消息 D 拆分成左右两部分，并且满足左部分消息大小为最大且严格小于 n 的 2 次幂，不妨记为 k，则右部分消息大小为 n-k，这样消息就分为 D[0:k]与 D[k:n]两部分。

代码实现：

在这里我采用 python 构造 Merkle Tree，主要的实现代码分为以下几个部分：
叶子节点的 hash 处理：

```
def hash_leaf(data,hash_function = 'sha256'):#merkle树叶节点
    hash_function = getattr(hashlib, hash_function)
    data = b'\x00'+data.encode('utf-8')
    return hash_function(data).hexdigest()
```

中间节点的 hash 处理：

```
def hash_node(data,hash_function = 'sha256'):#merkle树中间节点
    hash_function = getattr(hashlib, hash_function)
    data = b'\x01'+data.encode('utf-8')
    return hash_function(data).hexdigest()
```

以上两部分的节点生成满足要求一。
Merkle Tree 的实现代码：

```
def Create_Merkle_Tree(lst, hash_function = 'sha256'):
    lst_hash = []
    for i in lst:
        lst_hash.append(hash_leaf(i))
    merkle_tree = [copy.deepcopy(lst_hash)]
    assert len(lst_hash)>2, "no tracnsactions to be hashed"
    n = 0 #merkle树高度
    while len(lst_hash) >1:
        n += 1
        if len(lst_hash)%2 == 0:#偶数个叶节点
            v = []
            while len(lst_hash) >1 :
                a = lst_hash.pop(0)
                b = lst_hash.pop(0)
                v.append(hash_node(a+b, hash_function))
            merkle_tree.append(v[:])
            lst_hash = v
        else:#奇数个叶节点
            v = []
            last_leaf = lst_hash.pop(-1)
            while len(lst_hash) >1 :
                a = lst_hash.pop(0)
                b = lst_hash.pop(0)
                v.append(hash_node(a+b, hash_function))
            v.append(last_leaf)
            merkle_tree.append(v[:])
            lst_hash = v
    print(merkle_tree)
    return lst_hash, n+1 ,merkle_tree
```

通过对消息个数是偶数还是奇数进行划分，使得 Merkle Tree 的生成满足要求二。
Create_Merkle_Tree 函数的输出分为三部分：最终生成的 Merkle Tree 根节点的 HASH 值，Merkle Tree 的高度及 Merkle Tree 每一层节点的 HASH 值组成的 merkle_tree 数组。

测试结果：

输入的消息 D 为：

```
lst = ['a','b','c','d','e','f','g','h','i']
```

生成的 Merkle Tree 根节点的 HASH 值及高度为：

```
根结点hash值： ['ed07096540061a22a04f0a901ee46890ee790c78ddd784a70251ff01988258c
2']
Merkle树的高度： 5
```

2. Construct a merkle tree with 100k leaf nodes：

代码实现：

生成 100000 个结点的 Merkle Tree 代码如下：

```
lst = []
for i in range(100000):
    lst.append(str(i))
```

测试结果：

生成的 Merkle Tree 根节点的 HASH 值及高度为：

```
根结点hash值： ['6191ccc9f8fbe88c36b105e600a3bb45e955a5a7468d00af9b0cbdc98705d9f
5']
Merkle树的高度： 18
```

3. Construct the existence proof for randomly chosen leaf node & verify the proof：

首先是生成叶子节点的证明，所谓叶子节点的证明即是叶子节点所对应的路径上的HASH 值，即对于叶子节点 N，我们需要计算出根节点 R 的 HASH 值及叶子节点 N 到根节点路径上的每一个 HASH 值，值得注意的是，计算某个节点对应路径上一层的 HASH 值，需要用到该层节点的兄弟节点作为证据。

代码实现：

```
def Generate_Proof(merkle_tree,h,n,message,hash_function = 'sha256'):
    proof_list =[]
    hash_value = hash_leaf(message,hash_function)
    proof_list.append(hash_value) #叶子结点的hash值
    i = 1
    while i<h:
        L = len(merkle_tree[i-1])
        if L%2 == 1 and L-1 == n:
            break
        elif n%2 == 1:
            hash_value = hash_node(merkle_tree[i-1][n-1]+hash_value,hash_functic
            proof_list.append(hash_value)
        elif n%2 == 0:
            hash_value = hash_node(hash_value+merkle_tree[i-1][n+1],hash_functic
            proof_list.append(hash_value)
        n = n//2
        i += 1
    return proof_list
```

Generate_Proof 函数的输出 proof_list 代表根节点 R 到当前叶子节点 N 的路径上的 HASH 值数组。

测试结果：

我们生成的 Merkle Tree 输入的消息 D 为：

```
lst = ['a','b','c','d','e','f','g','h','i']
```

我们生成消息 D 中节点 a 的路径证明：

```
proof_list = Generate_Proof(merkle_tree,h,0,'a'
print("路径证明为：")
```

最终生成的节点 a 的路径证明为：

```
路径证明为：
022a6979e6dab7aa5ae4c3e5e45f7e977112a7e63593820dbec1ec738a24f93c
4c64254e6636add7f281ff49278beceb26378bd0021d1809974994e6e233ec35
9dc1674ae1ee61c90ba50b6261e8f9a47f7ea07d92612158edfe3c2a37c6d74c
1d295224db01021123c4c0c052e415c00a4c329b752eaf16764dbf6837366ccd
ed07096540061a22a04f0a901ee46890ee790c78ddd784a70251ff01988258c2
```

其次对于给出的一个叶子节点 N 的证明，我们需要验证其合法性，方式如下，首先我们取出之前生成 merkle_tree 中对应的真实节点的路径信息，再与我们输入的分路径信息进行对比，如果从叶子节点 N 到根节点 R 这条路径上的 HASH 值都一致，那么这是一个合法的证明，否则是一个非法的证明。

代码实现：

```python
def Verify_Proof(merkle_tree,h,n,message,proof,hash_function = 'sha256'):
    count = 1
    #print(len(merkle_tree))
    print("验证的叶子节点为：",message)
    print("该叶子节点的路径证明为：",proof)
    for i in range(len(proof)):
        if merkle_tree[i][n] == proof[i]:
            break
            #print('true')
            #print(i)
        else:
            count = 0
        if n==1:
            break
        else: n = n//2
    if count == 1:
        print("该路径证明合法")
    else:
        print("该路径证明非法")
```

测试结果：

我们将刚刚生成的叶子节点 a 生成的证明与节点 a 作为输入：

```
验证的叶子节点为： a
该叶子节点的路径证明为： ['022a6979e6dab7aa5ae4c3e5e45f7e977112a7e63593820dbec1e
c738a24f93c', '4c64254e6636add7f281ff49278beceb26378bd0021d1809974994e6e233ec35'
, '9dc1674ae1ee61c90ba50b6261e8f9a47f7ea07d92612158edfe3c2a37c6d74c', '1d295224d
b01021123c4c0c052e415c00a4c329b752eaf16764dbf6837366ccd', 'ed07096540061a22a04f0
a901ee46890ee790c78ddd784a70251ff01988258c2']
该路径证明合法
```

我们将刚刚生成的叶子节点 a 生成的证明与节点 d 作为输入：

```
验证的叶子节点为： d
该叶子节点的路径证明为： ['022a6979e6dab7aa5ae4c3e5e45f7e977112a7e63593820dbec1e
c738a24f93c', '4c64254e6636add7f281ff49278beceb26378bd0021d1809974994e6e233ec35'
, '9dc1674ae1ee61c90ba50b6261e8f9a47f7ea07d92612158edfe3c2a37c6d74c', '1d295224d
b01021123c4c0c052e415c00a4c329b752eaf16764dbf6837366ccd', 'ed07096540061a22a04f0
a901ee46890ee790c78ddd784a70251ff01988258c2']
该路径证明非法
```

PoC impl of the following pitfall

0. ECDSA 算法的实现：

ECDSA 是椭圆曲线数字签名生成算法.

ECDSA 算法可以分为以下三个部分：KeyGen，Sign，Verify，如下图：

- Key Gen: $P = dG$, n is order
- Sign(m)
  - $k \leftarrow Z_n^*, R = kG$
  - $r = R_x \ mod\ n, r \neq 0$
  - $e = hash(m)$
  - $s = k^{-1}(e + dr) \ mod\ n$
  - Signature is (r, s)
- Verify (r, s) of m with P
  - $e = hash(m)$
  - $w = s^{-1} \ mod\ n$
  - $(r', s') = e \cdot wG + r \cdot wP$
  - Check if $r' == r$
  - Holds for correct sig since
  - $es^{-1}G + rs^{-1}P = s^{-1}(eG + rP) =$
  - $k(e + dr)^{-1}(e + dr)G = kG = R$

在这里，我们使用 python 库中的 ecdsa 库进行实现，在这个库中已经封装好了相关函数，下面我们用 ecdsa 库的函数实现自己想要的功能：

功能 1：生成密钥函数 KeyGen：

代码如下：

```
def KeyGen():
    sk = SigningKey.generate(curve=NIST384p)
    pk = sk.verifying_key
    return sk,pk
```

说明如下：在这里我们调用库中的两个函数，分别生成签名公钥 sk 与其对应的验证公钥 pk。功能 2：签名函数 Sign：

代码如下：

```
def Sign(sk:SigningKey, m:str, k=None):
    return sk.sign(str2bytes(m), k=k)
```

说明如下：该函数对我们需要的消息 m 进行签名。

功能 3：验证函数 Verify：

代码如下：

```
def Verify(vk:VerifyingKey, m:str, signature)
    return vk.verify(signature, str2bytes(m))
```

说明如下：该函数对我们需要的消息 m 及签名（r,s）进行验证。

最后对我们算法实现的正确性进行验证：

代码实现：

```
print("0.验证ECDSA正确性: ")
m="123456"
print("密文m =",m)
sk,vk = KeyGen()
sign = Sign(sk,m)
r,s = util.sigdecode_string(sign, sk.privkey.order)
tag = Verify(vk,m,sign)
print("签名(r,s) =",(r,s))
print("正确性: ",tag)
```

测试结果：

```
0.验证ECDSA正确性:
密文m = 123456
签名(r,s) = (375247100198291518335000168053815956285547926961988522734051587413
15225902434155532106093157298879806786503865066018, 24188188734503352832404348301
5393506314973137782451348946344297872398610761589656707640396830165009574493056O
2483194)
正确性:  True
```

1. Leaking k leads to leaking of d：

当 k 泄露时我们可以利用 k 和（r,s）计算 d 的值，过程如下：

Recover d with σ and k

- $s = k^{-1}(e + dr) \bmod n$
- $ks = e + dr \bmod n$
- $d = r^{-1}(ks - e) \bmod n$

在这里我们需要计算 r 的逆，即借助扩展欧几里得算法；其次 e=Hash（m）也是可以计算的。

代码实现：

```
print("1.泄露k导致d的泄露: ")
k = 111111
sign = Sign(sk,m,k)
n = sk.privkey.order
e = Hash(m, sk)
r,s = util.sigdecode_string(sign, n)
r_inv,_,gcd = exgcd(r,n)
print("r的逆 =",r_inv)
print("e =",e)
print("真实的d =",sk.privkey.secret_multiplier)
d = (r_inv * (k*s-e)%n)%n
print("恢复出d =",d)
```

测试结果：

```
1.泄露k导致d的泄露:
r的逆 = 501422646981322842313337084230591561086955228984294261873217166455375551
9153087807722934621590532041135310345685117
e = 709577396890496680647746443044371606875712033819
真实的d = 4171761335474435817994052724588510829744450688158434904370562390420972
220010730511540006740148538337469167226717360
恢复出d = 4171761335474435817994052724588510829744450688158434904370562390420972
220010730511540006740148538337469167226717360
```

2. Leaking Secret Key Via Reusing k：

当 k 被重复使用在两组签名m_1和（r_1,s_1）及m_2和（r_2,s_2）当中时，我们可以通过以下方式计算 d：

- Signing message m_1 with d
  - Randomly select $k \in Z_n^*$, $R = kG = (x, y)$
  - $e_1 = hash(m_1)$
  - $r_1 = x \bmod p,\ s_1 = k^{-1}(e_1 + r_1 d) \bmod n$
- Signing message m_2 with d
  - Reuse the same $k \in Z_n^*$, $R = kG = (x, y)$
  - $e_2 = hash(m_2)$
  - $r_1 = x \bmod p,\ s_2 = k^{-1}(e_2 + r_1 d) \bmod n$
- Recovering d with 2 signatures $(r_1, s_1), (r_1, s_2)$
  - $\frac{s_1}{s_2} = \frac{e_1 + r_1 d}{e_2 + r_1 d} \bmod n \rightarrow d = \frac{s_1 e_2 - s_2 e_1}{s_2 r_1 - s_1 r_1} \bmod n$

说明如下：因为 $R = kG = (x, y)$，故有 $r = x(\bmod n)$。
因为 k 的重复使用，导致$r_1 = r_2 = r$，此时又有:
$s_1 = k^{-1}(e_1 + r_1 d)(\bmod n)$以及$s_2 = k^{-1}(2 + r_2 d)(\bmod n)$可知:
$\frac{s_1}{s_2} = \frac{e_1 + dr_1}{e_2 + dr_2} = \frac{e_1 + dr}{e_2 + dr}$ 故有$d = \frac{s_1 e_2 - s_2 e_1}{s_2 r - s_1 r}$

代码实现：

```
print("2.重复使用k导致d的泄露：")
k = 111111
m1 = "123"
m2 = "456"
sign1 = Sign(sk,m1,k)
sign2 = Sign(sk,m2,k)
e1 = Hash(m1, sk)
e2 = Hash(m2, sk)
n = sk.privkey.order
r1,s1 = util.sigdecode_string(sign1, n)
r1_inv,_,gcd1 = exgcd(r1,n)
r2,s2 = util.sigdecode_string(sign2, n)
r2_inv,_,gcd2 = exgcd(r2,n)
print("真实的d =",sk.privkey.secret_multiplier)
d = (((s1*e2 - s2*e1)%n) * getinv(s2*r1 - s1*r2,n)) %n
print("恢复出d =",d)
```

测试结果：

```
2.重复使用k导致d的泄露:
真实的d = 4171761335474435817994052724588510829744450688158434904370562390420972220010730511540006740148538337469167226717360
恢复出d = 4171761335474435817994052724588510829744450688158434904370562390420972220010730511540006740148538337469167226717360
```

3. Two users, using k leads to leaking of d, that is they can deduce each other's d:
两个用户 A 和 B 分别使用私钥d_1和私钥d_2，但是由于使用同一个 k，他们可以求解对方的 d：

- Alice signed message m_1 with d_1, $\sigma_1 = (r, s_1)$
  - $s_1 = k^{-1}(e_1 + rd_1) \bmod n$, where $e_1 = hash(m_1)$
- Bob signed message m_2 with d_2, $\sigma_2 = (r, s_2)$
  - $s_2 = k^{-1}(e_2 + rd_2) \bmod n$, where $e_2 = hash(m_2)$
- Then we have
  - $k = s_1^{-1}(e_1 + rd_1) \bmod n$
  - $k = s_2^{-1}(e_2 + rd_2) \bmod n$
  - And we have
  - $s_1(e_2 + rd_2) = s_2(e_1 + rd_1) \bmod n$
- Now Alice can deduce Bob's secret key
  - $d_2 = (s_2 e_1 - s_1 e_2 + s_2 r d_1)/(s_1 r) \bmod n$
- And Bob can deduce Alice's secret key
  - $d_1 = (s_1 e_2 - s_2 e_1 + s_1 r d_2)/(s_2 r) mod n$

说明如下：因为 $R = kG = (x, y)$，故有 $r = x(\bmod n)$。
由$s_1 = k^{-1}(e_1 + rd_1)(\bmod n)$以及$s_2 = k^{-1}(2 + rd_2)(\bmod n)$可知：
$k = s_1(e_1 + rd_1) = s_2(e_2 + rd_2)(\bmod n)$，由于 A 拥有$d_1$，那么 A 可以计算$d_2$如下：
$d_2 = (s_2 e_1 - s_1 e_2 + s_2 r d_1)/(s_1 r)(\bmod n)$，同样由于 B 拥有$d_2$，那么 B 可以计算$d_1$如下：
$d_1 = (s_1 e_2 - s_2 e_1 + s_1 r d_2)/(s_2 r)(\bmod n)$

代码实现：

```
print("3.使用相同的k导致d1d2的泄露: ")
sk1,vk1 = KeyGen()
sk2,vk2 = KeyGen()
k = 111111
m1 = "123"
m2 = "456"
sign1 = Sign(sk1,m1,k)
sign2 = Sign(sk2,m2,k)
e1 = Hash(m1, sk1)
e2 = Hash(m2, sk2)
n = sk.privkey.order
r1,s1 = util.sigdecode_string(sign1, n)
r1_inv,_,gcd1 = exgcd(r1,n)
r2,s2 = util.sigdecode_string(sign2, n)
r2_inv,_,gcd2 = exgcd(r2,n)
d1 = sk1.privkey.secret_multiplier
d2 = sk2.privkey.secret_multiplier
D1 = (((s1*e2 - s2*e1 + s1*d2*r2)%n) * getinv(s2*r1,n)) %n
D2 = (((s2*e1 - s1*e2 + s2*d1*r1)%n) * getinv(s1*r2,n)) %n
print("真实的d1 =",sk1.privkey.secret_multiplier)
print("恢复出d1 =",D1)
print("真实的d2 =",sk2.privkey.secret_multiplier)
print("恢复出d2 =",D2)
```

测试结果：

```
3.使用相同的k导致d1d2的泄露:
真实的d1 = 1408699884698325593978573590842055519569153692384575165326689446280805606814316874878666515682395527399166467978340 1
恢复出d1 = 1408699884698325593978573590842055519569153692384575165326689446280805606814316874878666515682395527399166467978340 1
真实的d2 = 2378294358626974748665603417447291588514419352710593650130069888647898137741620918900077841459235071693210462233617 9
恢复出d2 = 2378294358626974748665603417447291588514419352710593650130069888647898137741620918900077841459235071693210462233617 9
```

4. Malleability of ECDSA, e.g.（r,s） and （r,−s）are both valid signatures:

如果签名（r,s）是合法的，则签名（r,-s）也是合法的：

Verification on (r, s) :

- $e \cdot s^{-1}G + r \cdot s^{-1}P = (x', y'), r = x' mod\ p$

Verification on $(r, -s)$ also succeed:

- $e \cdot (-s)^{-1}G + r \cdot (-s)^{-1}P = -(e \cdot s^{-1}G + r \cdot s^{-1}P) = (x', -y'),\ r = x' mod\ p$

说明如下：由于签名（r,s）是合法的，我们可以得到以下结论：

$es^{-1}G + rs^{-1}P = (x, y)$及 $r = x(mod n)$

那么将签名（r,-s）代入验证可得：

$e(-s)^{-1}G + r(-s)^{-1}P = -(e(s)^{-1}G + r(s)^{-1}) = -(x, y)$及 $r = x(mod n)$，由椭圆曲线的性质：$-(x, y) = (x, -y)$，因此签名（r,-s）也是合法的。

代码实现：

```
print("4.验证签名(r,-s)的合法性: ")
m="123456"
sk,vk = KeyGen()
n = sk.privkey.order
sign1 = Sign(sk,m)
r,s = util.sigdecode_string(sign1, n)
sign2 = util.sigencode_string(r, (-s)%n, n)
tag1 = Verify(vk,m,sign1)
tag2 = Verify(vk,m,sign2)
print("验证签名(r,s)的合法性：",tag1)
print("验证签名(r,-s)的合法性：",tag2)
```

测试结果：

```
4.验证签名(r,-s)的合法性:
验证签名(r,s)的合法性： True
验证签名(r,-s)的合法性： True
```

5. Pretend to be satoshi as one can forge signature if the verification does not check m:

当验证算法不检查 m 时，而是检查 e（即 h(m)）时，我们就可以构造伪造：

$\sigma = (r, s)$ 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:

$s^{-1}(eG + rP) = (x', y') = R', r' = x' mod\, n == r$?

To forge, choose $u, v \in \mathbb{F}_n^*$

Compute $R' = (x', y') = uG + vP$

Choose $r' = x' \; mod \; n$, to pass verification, we need

$s'^{-1}(e'G + r'P) = uG + vP$

- $s'^{-1}e' = u \; mod \; n \rightarrow e' = r'uv^{-1} mod \; n$
- $s'^{-1}r' = v \; mod \; n \rightarrow s' = r'v^{-1} \; mod \; n$

$\sigma' = (r', s')$ is a valid signature of e' with secret key d

说明如下：在验证算法里我们需要判断以下两个等式是否成立：

$s_{-1}(eG + rP) = (x, y)$及 $r = x(modn)$。

我们可以进行如下构造：

任取 u、v ∈ F_{n*}，计算$R^` = uG + vP = (x^`, y^`)$；选择合适的$r^` = x^` mod\, n$，要通过上述验证等式，要求如下：满足$s^{`-1}e^` = u(modn)$ 及$s^{`-1}r^` = v(modn)$，即满足$e^` = r^`uv^{-1}(modn)$与$s^` = r^`v^{-1}(modn)$，那么此时$(r^`, s^`)$即是$e^`$的合法签名。

代码实现：

```
print("5.当仅验证e时给出伪造: ")
sk,vk = KeyGen()
n = sk.privkey.order
G = vk.pubkey.generator
P = vk.pubkey.point
u=3
v=5
xy = u*G+v*P
r = xy.x() % n
s = (r*getinv(v,n))%n
e = (s*u)%n
sig = Signature(r,s)
tag = vk.pubkey.verifies(e, sig)
print("伪造的e =",e)
print("对应的签名(r,s) =",(r,s))
print("正确性验证: ",tag)
```

测试结果：

```
5.当仅验证e时给出伪造:
伪造的e = 3433693231654439589695523398521493740473904148349796493379739778396410
4346683124545597533275698207481928207543568733
对应的签名(r,s) = (1782621433117951394931301654188128186948532986869782822171542
4360312514512025277339930265818011384222992578918671912, 11445644105514798632318
41132840497913491301382783265497793246592798803478222770818186584442523273582730
9402514522911)
正确性验证:  True
```

6. Same d and k used in ECDSA & Schnorr signature, leads to leaking of d:

首先对 Schnorr 签名算法进行说明，算法实现流程如下：

- Key Generation
  - $P = dG$
- Sign on given message M
  - randomly k, let $R = kG$
  - $e = hash(R||M)$
  - $s = k + ed \; mod \; n$
  - Signature is : (R, s)
- Verify (R, s) of M with P
  - Check sG vs $R + eP$
  - $sG = (k + ed)G = kG + edG = R + eP$

当我们的 ECDSA 算法和 Schnorr 签名算法都使用相同的 d 和 k 时，我们能够计算出 d 的值：

ECDSA signing with private key d

- Randomly select k, $R = kG = (x, y)$
- $e_1 = hash(m)$
- $r_1 = x \bmod n, s_1 = (e_1 + r_1 d)k^{-1} mod\ p$
- Signature (r_1, s_1)

Schnorr signing with private key d

- Reuse the same k as ECDSA, $R = kG$
- $e_2 = h(R||m)$
- $R = kG, s_2 = (k + e_2 d) mod\ p$
- Signature (R, s_2)

With the two sigs, private key d can be recovered:

- $s_1 = (e_1 + r_1 d)(s_2 - e_2 d)^{-1}\ mod\ p$
- $d = \frac{s_1 s_2 - e_1}{(s_1 e_2 + r_1)} mod\ p$

说明如下：在 ECDSA 中，我们使用 d 和 k 对消息m_1进行签名可得（r_1, s_1），并满足：$R_1 = kG$ 、$s_1 = (e_1 + rd_1)k_{-1}(modn)$及$r_1 = x(modn)$。

在 Schnorr 签名算法中，我们使用 d 和 k 对消息m_2进行签名可得（R_2, s_2），并满足：$s_2 = k + e_2 d(modn)$及$R_2 = kG$，故 $k = s_2 - e_2 d(modn)$

由此可知 $s_1 = (e_1 + rd_1)(s_2 - e_2 d)_{-1}(modn)$，$d = s_1 s_2 - e_{1/} s_1 e_2 + r_1 (modn)$。

代码实现：

```
print("6.ECDSA和Schnorr使用相同的d和k导致d的泄露: ")
m="123456"
sk,vk = KeyGen()
n = sk.privkey.order
k = 111111
# ECDSA
e1 = Hash(m, sk)
sign = Sign(sk,m,k)
r1,s1 = util.sigdecode_string(sign, sk.privkey.order)
# Schnorr
e2 = hash(m)
sign2 = SchnorrSign(sk,e2,k)
R,s2 = sign2
print("真实的d =",sk.privkey.secret_multiplier)
d = (((s1*s2-e1)%n) * getinv(s1*e2+r1,n)) %n
print("恢复出d =",d)
```

测试结果：

```
6.ECDSA和Schnorr使用相同的d和k导致d的泄露:
真实的d = 27144326996749637172346223178842598105117205663273958571757997437558686674820645803752104737191256876101768910747812
恢复出d = 27144326996749637172346223178842598105117205663273958571757997437558686674820645803752104737191256876101768910747812
```