网络空间安全创新创业实践

学院：网络空间安全学院

姓名：葛梦云

学号：202100460029

2023 年 8 月 4 日

目录

1. Project1：implement the naïve birthday attack of reduced SM3

1.1. 实现方式

生日攻击原理：利用"两个集合相交"问题的原理生成散列函数碰撞，达到目的的攻击称为生日攻击，也称为平方根攻击。生日攻击方法没有利用 Hash 函数的结构和任何代数弱性质，它只依赖于消息摘要的长度，即 Hash 值的长度。

对于 SM3 的生日攻击，就是要不断地计算 SM3，下一次计算时，将本次结果和之前的得到的计算结果比较，如果找到相同的值，就说明产生了一个碰撞，攻击成功。

本项目需要完成对 SM3 的生日攻击，可以利用 Python 中的 gmssl 完成。关于 gmssl，它是一个提供国密 SSL 相关软件/工具/服务的网站，网站简称国密 SSL 实验室。可以利用里面的 SM2/SM3/SM4/ZUC 等国密算法来帮助我们，本项目使用了 gmssl 里的 SM3 算法。

为了对 SM3 进行攻击，编写函数数 birthday_attack()。函数如下。

```
1.def birthday_attack(collision):
2.    global col_list, msg_sm3, i, msg
3.    # 进行碰撞储存的数组
4.    col_list = [-1] * pow(2, collision)
```

```
    msg = int(pow(2, collision))#消息

    for i in range(msg):
        #SM3 计算
        message = sm3.sm3_hash(func.bytes_to_list(bytes(str(i),
encoding='utf-8')))
        msg_sm3 = int(message[0:int(collision / 4)], 16)

        if col_list[msg_sm3] == -1:
            col_list[msg_sm3] = i
            if i + 1 == msg:
                print("遍历完成，未找找到碰撞。")
                break
        else:
            break
```

下面解释函数中的一些细节。数组 col_list 是用来储存碰撞值的，其中的每个值都被初始化为-1，msg 为消息空间。然后遍历消息空间 msg 中的每个消息，进行 SM3 计算，并进行比较，看能否找到碰撞。因为测试时使用 20bit，碰撞的只有消息的前 20bit，所以只用取出消息的前 20bit。与之前的计算结果进行比较时，用取出的 20bit 作为索引查找数组 col_list，若该索引处的值为-1，则进行更新，若该索引处的值不是-1，则说明它已经被更新过，即找到了一个不同于当前消息但前 20bit 经过 SM3 计算有相同结果的消息，即成功找到了一个碰撞。但有时，遍历完整个消息空间都不会发生这种情况，即未成功找到碰撞。如果找到碰撞，则将发生碰撞的两个消息存下来以便后续输出。同时，用 python 自带的 time（）函数进行计时，以获得找到碰撞所用的时间。

1.2 实现效果

代码运行结果如下：

碰撞的 bit 数为 20，运行速度为 0.4262728691101074 s

```
输入要碰撞的比特数(测试使用20bit即可):20
找到碰撞.
消息 1402 与消息 1091 的前 20 bit的SM3哈希值碰撞了
碰撞部分的16进制表示为 0x7e037
碰撞的时间为 0.4262728691101074 s
```

碰撞的 bit 数为 30，运行速度为 17.67283535003662 s

```
输入要碰撞的比特数(测试使用20bit即可):30
找到碰撞.
消息 41780 与消息 38760 的前 30 bit的SM3哈希值碰撞了
碰撞部分的16进制表示为 0x7523b64
碰撞的时间为 17.67283535003662 s
```

2. Project2：implement the Rho method of reduced SM3

2.1 实现方式

基本原理：Pollard 于 1975 年提出著名的ρ方法，该方法的基本思想是通过多项式迭代产生数列，从中寻找整数 x1，x2 满足 gcd(x1 – x2，N) 是 N 的一个非平凡因子.设一个数的素数分解为 N = pq. 该算法考虑伪随机序列 {xi} = {x0，f(x0)，f(f(x0))，• • • }，其中 f 是多项式函数，通常选择 f(x) = xˆ2 + c mod N，c = 1. f(x) 在模 N 意义下必然会形成一个环，所以该序列迟早会进入一个循环。该算法从生成的数列中通过某种低存储的方法寻找整数 x1，x2，使得 gcd(x1 – x2，N) 是 N 的一个非平凡因子。

■ Pollard ρ算法

原理： 寻找两个元素x_1与x_2,满足$x_1 \neq x_2 \bmod n$，$x_1 = x_2 \bmod p$

利用$\gcd(x_1\text{-}x_2, n)$计算n的因子p

时间复杂度：生日攻击 $1.17\sqrt{p}$

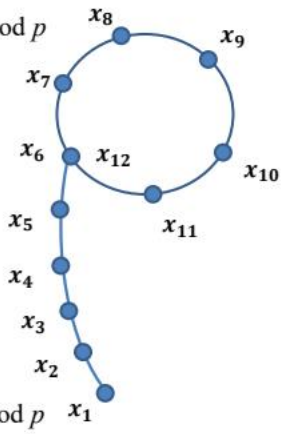
方法：假定映射F类似于一个随机映射。设$x_1 \in Z_n$，

考虑序列$x_1, x_2, \ldots$，其中$x_i = F(x_{i-1})$，对所有$i \geq 2$

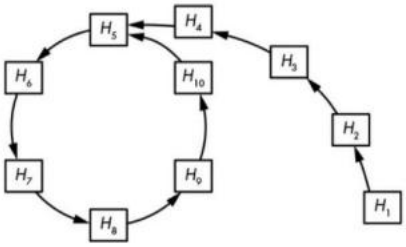
X为Z_n的m个元素的随机子集

寻找两个不同的$x_i, x_j \in X$，使得$\gcd(x_j - x_i, n) > 1$

$x_1 \bmod p \to x_2 \bmod p \to \cdots x_i \bmod p \to \cdots \to x_j \bmod p = x_i \bmod p$



至于 SM3 的 Rho 方法攻击，如下图所示：



由图可知，要找的为 H5，即发生碰撞处。它由 H4、H10 都能得到。

攻击的主要思路为：

①随机选取初始值，即 col_initial

②用 SM3 计算第一个值 col_m1，同时把它对应碰撞比特的值放进集合 set_collision 中

③不停进行 col_m1 的 SM3 计算，并检查计算结果对应碰撞比特的值是否在集合 set_collision 中；若在，则是找到碰撞，返回碰撞结果，跳出循环；若不在，继续算直至找到碰撞

④输出结果

相关步骤代码如下：

- 随机选取初始值，即 col_initial
- 用 SM3 计算第一个值 col_m1，同时把它对应碰撞比特的值放进集合 set_collision 中

```
1.    num = int(collision / 4)
2.        col_initial = hex(random.randint(0, pow(2, (collision + 1)) - 1))
[2:]
3.        set_collision = set()
4.        times = 0
5.
6.         col_m1 = sm3.sm3_hash(func.bytes_to_list(bytes(str(col_initial),
 encoding='utf-8')))
7.        set_collision.add(col_m1[:num])
```

③不停进行 col_m1 的 SM3 计算，并检查计算结果对应碰撞比特的值是否在集合 set_collision 中；若在，则是找到碰撞，返回碰撞结果，跳出循环；若不在，继续算直至找到碰撞

```
    while True:
            # 碰撞次数
            times += 1
             col_m1 = sm3.sm3_hash(func.bytes_to_list(bytes(str(col_m1), encoding='utf-8')))
            if col_m1[:num] in set_collision:
                return col_m1[:num], times
            else:
                set_collision.add(col_m1[:num])
```

④输出结果

```
    #以 30bit 碰撞为例
      collision = int(input("输入要碰撞的比特数(测试使用 30bit 即可):"))

      #开始碰撞
      start_time = time.time()
      collision_hex, collision_times = Rho_Method(collision)
      end_time = time.time()
      print("找到碰撞。")
       print("碰撞的消息的前", collision, "bit 相同，即为（十六进制表示）:", collision_hex)
       print("碰撞的计算次数:", collision_times, "\n 碰撞需要的时间:", end_time - start_time, "s")
```

2.2 实现效果

代码运行结果如下：

```
y
输入要碰撞的比特数(测试使用30bit即可):30
找到碰撞。
碰撞的消息的前 30 bit相同，即为（十六进制表示）： afd0b8b
碰撞的计算次数: 992
碰撞需要的时间: 0.5960679054260254 s
```

注：项目要利用 python 中的 gmssl。

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

3.1 实现方式

基本原理：

长度扩展攻击（length extension attack），是指针对某些允许包含额外信息的加密散列函数的攻击手段。对于满足以下条件的散列函数，都可以作为攻击对象：

① 加密前将待加密的明文按一定规则填充到固定长度（例如 512 或 1024 比特）的倍数；

② 按照该固定长度，将明文分块加密，并用前一个块的加密结果，作为下一块加密的初始向量（Initial Vector）。

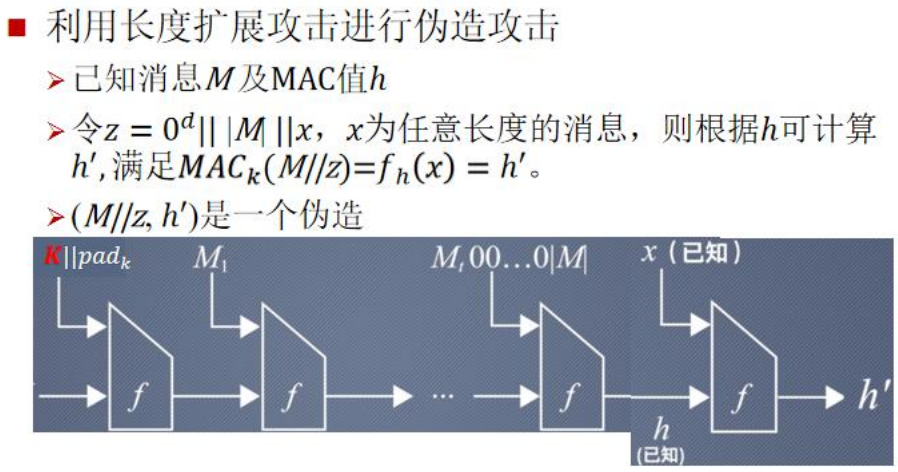
满足上述要求的散列函数称为 Merkle－Damgård 散列函数，在以下条件满足的情况下，攻击者可以通过该方法获取 H(salt+一定规则构造的 data)：

① 知道密文的加密算法且该算法满足 Merkle－Damgård 散列函数特征；

② 不知道 salt，但知道 salt 的长度，并可控制 data 的值；

③ 可以得到一个 H(salt+data)的值。

长度扩展攻击示意图如下：



对于 SM3 的长度扩展攻击，我们需要：

①随机生成消息 secret，用 SM3 计算其 hash 值 hash1；

②生成一个附加消息 append_m

③用 hash1 推出本次加密结束后 8 个向量的值，然后把它当作初始向量，加密由与 secret 等长的任意字符串（填充后）级联上 append_m 的字符串，得到另一个 hash 值 hash2

④用 SM3 计算 secret + padding + append_m 的 hash 值 hash3

若攻击成功，应有 hash2＝hash3。

因为 SM3 的消息长度为 64 字节或为 64 的倍数，所以若消息长度不满足条件就要进行填充。具体填充规则为：先填充一个 1，然后填充 0 直至消息长度为 56（或加上 k*64，k 为整数）字节，最后 8 字节表示原消息的长度。

SM3 计算时，先对消息进行分组，一组为 64 字节，进行分组加密，然后更新初始向量，新的向量用于下次加密，可以据此实现长度扩展攻击：得到首次加密后的向量 IV 后，构造“secret”+padding+append_m，下次加密时使用，如此，就在 secret 未知的情况下得到 hash 值，且该值合法。

相关步骤代码如下：

随机生成 secret，计算其 hash 值 hash1，随机设置附加消息 append_m。

```
secret = str(random.random())#随机生成消息
  #附加消息
  append_m = "lalaland"
  pad_str = ""
  pad = []
   secret_hash = sm3.sm3_hash(func.bytes_to_list(bytes(secret, encoding='utf-8')))
  secret_len = len(secret)
```

把加密完 secret 得到的 hash 值按 每组 8 字节分组，把每组的值转换为 int 型。

```
#分组，转换为整数
   for r in range(0, len(old_hash), 8):
       vectors.append(int(old_hash[r:r + 8], 16))
```

构造消息，secret 部分可以用等长的任意字符代替，然后进行填充，再加上附加信息 append_m 即可。

```
#伪造消息
    for i in range(0, secret_len):
        message += 'a'
    message = func.bytes_to_list(bytes(message, encoding='utf-8'))
```

```
        message = padding(message) #填充
         message.extend(func.bytes_to_list(bytes(append_m, encoding='utf-8')))
```

填充函数如下：

```
    #填充函数
    def padding(msg):
        mlen = len(msg)
        msg.append(0x0080)
        mlen += 1
        tail = mlen % 64
        range_end = 56
        if tail > range_end:
            range_end = range_end + 64
        for i in range(tail, range_end):
            msg.append(0x0000)
        bit_len = (mlen - 1) * 8
        msg.extend([int(x) for x in struct.pack('>q', bit_len)])
        for j in range(int((mlen - 1) / 64) * 64 + (mlen - 1) % 64, len(msg)):
            global pad
            pad.append(msg[j])
            global pad_str
            pad_str += str(hex(msg[j]))
        return msg
```

加密，因为无需知道 secret 的值，所以只需对 append_m 加密，故修改 gmssl 中 SM3 的 hash 函数 sm3_hash()，增加 new_IV 参数，代表加密 secret 后的向量值，所以加密次数减一，且从第 64 字节开始，最终得到构造消息的 hash 值 hash2。下面为 sm3_hash()修改之处。

```
    #此处进行修改，把 IV 换为第一步 hash 时各寄存器的值
    def sm3_hash(msg, new_IV):
        len1 = len(msg)
        reserve1 = len1 % 64
        msg.append(0x0080)
        reserve1 = reserve1 + 1
        range_end = 56
        if reserve1 > range_end:
            range_end = range_end + 64

        for i in range(reserve1, range_end):
```

```
            #填充 0，直到到消息长度为 56 或再加整数倍的 64 字节
            msg.append(0x0000)

        bit_length = len1 * 8
        bit_length_str = [bit_length % 0x0100]
        for i in range(7):
            bit_length = int(bit_length / 0x0100)
            bit_length_str.append(bit_length % 0x0100)
        for i in range(8):
            msg.append(bit_length_str[7 - i])

        #此处进行修改，把加密次数减少一次
        group_count = round(len(msg) / 64) - 1

        B = []
        for i in range(0, group_count):
            #加密从第 64 字节开始
            B.append(msg[(i + 1) * 64:(i + 2) * 64])

        #初始值为更新后的向量值
        V = [new_IV]
        for i in range(0, group_count):
            V.append(sm3_cf(V[i], B[i]))

        y = V[i + 1]
        result = ""
        for i in y:
            result = '%s%08x' % (result, i)
        return result
```

利用 generate_guess_hash 函数加密构造的消息。

```
    def generate_guess_hash(old_hash, secret_len, append_m):
        vectors = []
        message = ""
        #分组，转换为整数
        for r in range(0, len(old_hash), 8):
            vectors.append(int(old_hash[r:r + 8], 16))

        #伪造消息
        for i in range(0, secret_len):
            message += 'a'
        message = func.bytes_to_list(bytes(message, encoding='utf-8'))
        message = padding(message) #填充
```

```
        message.extend(func.bytes_to_list(bytes(append_m, encoding='utf-8')))
    return sm3_hash(message, vectors)
```

验证，需要有正确的消息的 hash 值即 hash3，然后判断 hash2 是否等于 hash3。

```
start_time = time.time()
    #hash2
    guess_hash = generate_guess_hash(secret_hash, secret_len, append_m)
    #new_msg 为 secret+padding+append_m
    new_msg = func.bytes_to_list(bytes(secret, encoding='utf-8'))
    new_msg.extend(pad)
    new_msg.extend(func.bytes_to_list(bytes(append_m, encoding='utf-8')))
    new_msg_str = secret + pad_str + append_m
    new_hash = sm3.sm3_hash(new_msg)  # hash3
    end_time = time.time()

    print("原消息：", secret)
    print("原消息的长度：", len(secret))
    print("原消息的 hash 值（hash1）：" + secret_hash)
    print("附加消息：", append_m, "\n")
    print("伪造消息的 hash 值（hash2）：", guess_hash)
    print("正确消息（secret+padding+append_m）：", new_msg_str)
    print("正确消息的 hash 值（hash3）：", new_hash)

    if new_hash == guess_hash:
        print("\n 成功")
    else:
        print("\n 失败")

    print("所用时间为：", end_time - start_time, "s")
```

3.2 实现效果

代码运行结果如下：

```
原消息： 0.7336402874428173
原消息的长度： 18
原消息的hash值（hash1）：56aeebe01623fc289386d48b4492cd8ad158d5f03b77dd8c766146f710409f7d
附加消息： lalaland

伪造消息的hash值（hash2）： 6c7a3079a64920629c5cda51c2b9bca5f61c94280a273e16487965f8f0d8948b
正确消息（secret+padding+append_m）： 0.73364028744281730x800x00x00x00x00x00x00x00x00x00x00x00x00x0
0x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x901a
laland
正确消息的hash值（hash3）： 6c7a3079a64920629c5cda51c2b9bca5f61c94280a273e16487965f8f0d8948b

成功
所用时间为： 0.0009014606475830078 s
```

4.Project4:do your best to optimize SM3 implementation (software)

未做

5.Project5:Impl Merkle Tree following RFC6962

5.1 实现方式

基本原理：默克尔树常见的结构是二叉树，但它也可以是多叉树，它具有树结构的全部特点。 默克尔树的基础数据不是固定的，想存什么数据都可以，因为它只要数据经过哈希运算得到的哈希值。 默克尔树是从下往上逐层计算，每个中间节点是根据相邻的两个叶子节点组合计算得出的，而根节点是根据两个中间节点组合计算得出的，所以叶节点是基础。因此，底层数据的任何变动，都会传递到其父节点，一直到树的根节点。默克尔树是区块链技术中用于保障数据不被篡改的重要安全手段之一，有着非常重要的作用。

它具有以下几个特点：

①数据结构是一个树，可以是二叉树，也可以是多叉树（本 BLOG 以二叉树来分析）

②叶子节点的 value 是数据集合的单元数据或者单元数据 HASH。

③非叶子节点 value 是其所有子节点 value 的 HASH 值。

- Construct a Merkle tree with 10w leaf nodes

生成 Merkle 树的过程如下：

①该层结点有偶数个，两两配对，加上前缀级联生成父结点的 hash 值

②该层结点有奇数个，最后一个结点作为父结点，别的结点两两配对，加上前缀级联生成父结点的 hash 值

③每层的结点都由靠近叶结点的下面一层生成，不断生成结点直至生成根结点

相关步骤代码如下：

用 hash_leaf()建立叶子结点，然后用 hash_node()建立别的结点，其中叶子结点前缀为 0x00，其余结点前缀为 0x01。

```
#Merkle 树的叶子结点
def hash_leaf(data, hash_function='sha256'):
    hash_function = getattr(hashlib, hash_function)
    data = b'\x00' + data.encode('utf-8')
    return hash_function(data).hexdigest()

#Merkle 树的其余结点
def hash_node(data, hash_function='sha256'):
    hash_function = getattr(hashlib, hash_function)
    data = b'\x01' + data.encode('utf-8')
    return hash_function(data).hexdigest()
```

用函数 Create_Merkle_Tree()创建 Merkle 树。

先把列表 lst 中的元素全当做叶子结点，并用多维数组表示 Merkle 树；然后判断结点数量， 若结点数量＜2，则不能建 Merkle 树。

反之，结点数量≥2：当有偶数个结点时，两两配对即可，每次从 lst_hash 中弹出两个结点，求其数值的和，用 hash_node()函数处理；当有奇数个结点数时，弹出 lst_hash 的最后一个元素作为根结点，剩余元素同偶数处理，完成后把最后一个元素加入列表，循环，直至只余下一个结点即根结点。

```
#生成 Merkle 树
def Create_Merkle_Tree(lst, hash_function='sha256'):
    lst_hash = []
    for i in lst:
        lst_hash.append(hash_leaf(i))#先全都设成叶子结点
    merkle_tree = [copy.deepcopy(lst_hash)]
    #如果结点数太少，则生成 Merkle 树失败
    if len(lst_hash) < 2:
        print("结点数太少，生成 Merkle 树失败！")
        return 0
    h = 0 #Merkle 树的高度
    while len(lst_hash) > 1:
        h += 1
        if len(lst_hash) % 2 == 0:#如果为偶数结点
            v = []
            while len(lst_hash) > 1:
                #pop 两个结点
```

```
                    a = lst_hash.pop(0)
                    b = lst_hash.pop(0)
                    v.append(hash_node(a + b, hash_function))
                merkle_tree.append(v[:])#Merkle 树更新
                lst_hash = v
            else: #如果为奇数结点
                v = []
                last_node = 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_node)
                #Merkle 树更新一层
                merkle_tree.append(v[:])
                lst_hash = v
        return merkle_tree, h
```

Show_Merkle_Tree 函数，用于输出 Merkle 树。

```
    def Show_Merkle_Tree(merkle_tree, h):
        print("Merkle Tree 高度:", h)
        print("Merkle Tree 从下到上每一层节点的哈希值为:")
        for i in range(h + 1):
            print("第", i + 1, "层:")
            len_tree = len(merkle_tree[i])
            for j in range(len_tree):
                print(merkle_tree[i][j])
            print()
        print()
```

- Build inclusion(exclusive) proof for specified element
- Build exclusion proof for specified element

项目要求我们实现如下功能：对某一个 Merkle 树，给出需要查找的结点、结点序号，即可判断该结点是否在该 Merkle 树中。

具体实现过程如下：先据给出的叶子结点，计算其 hash 值；据结点序号 n 的奇偶性，判断查找的结点 n-1 或结点 n+1，若该层有奇数个结点并且结点为最后一

个，则父结点是它本身；然后取 1/2 的 n，就为生成的父结点，然后根据这个进行下一层的查找；重复操作，不断向上查找，最后判断生成的哈希值与根结点是否相等，若相等则说明该结点在 Merkle 树中，否则则不在，或者可能是对应的序号错误。

相关步骤代码如下：

```
#以下为构造第 n 个叶子节点存在性和验证
#Merkle 树高度为 h，查找的序号为 n
def Audit_Proof(merkle_tree, h, n, leaf, hash_function='sha256'):
    if n >= len(merkle_tree[0]):
        print("结点序号错误！")
        return 0
    print("序号:", n, "字符:", leaf, "\n 查找路径:")
    j = 0 #第 j 层,最底层要计算叶子结点的哈希值
    L = len(merkle_tree[0])
    #叶子结点有奇数个，n 是最后一个结点
    if L % 2 == 1 and L - 1 == n:
        hash_value = hash_leaf(leaf)
        print("第", j + 1, "层的 Hash 值:", hash_value)
    elif n % 2 == 1:
         hash_value = hash_node(merkle_tree[0][n - 1] + hash_leaf(lea
f), hash_function)
        print("第", j + 1, "层的查找值:", merkle_tree[0][n - 1], "\n\t
生成的 Hash 值:", hash_value)
    elif n % 2 == 0:
         hash_value = hash_node(hash_leaf(leaf) + merkle_tree[0][n +
1], hash_function)
        print("第", j + 1, "层的查找值:", merkle_tree[0][n + 1], "\n\t
生成的 Hash 值:", hash_value)
    n = n // 2
    j += 1

    #查找兄弟结点哈希值，并生成新的哈希值
    while j < h:
        L = len(merkle_tree[j])
        #结点数为奇数，且 n 是最后一个节点
        if L % 2 == 1 and L - 1 == n:
            print("第", j + 1, "层的 Hash 值:", hash_value)
        elif n % 2 == 1:
             hash_value = hash_node(merkle_tree[j][n - 1] + hash_valu
e, hash_function)
                                print(" 第 ", j + 1, " 层 的 查 找
值:", merkle_tree[j][n - 1], "\n\t 生成的 Hash 值:", hash_value)
```

```
            elif n % 2 == 0:
                hash_value = hash_node(hash_value + merkle_tree[j][n + 1], hash_function)
                print(" 第 ", j + 1, " 层 的 查 找 值:", merkle_tree[j][n + 1], "\n\t 生成的 Hash 值:", hash_value)
            n = n // 2
            j += 1
        print('\n 根结点哈希值:', merkle_tree[h][0])
        if hash_value == merkle_tree[h][0]:
            print("结点", leaf, "在 Merkle 树中")
        else:
            print("结点", leaf, "不在 Merkle 树中，或者是序号错误")
```

5.2 实现效果

运行代码，建立 10w 个叶子结点的 Merkle 树，部分运行结果如下：

```
第 14 层:
c089da8360be0503314367c6dfabb2a2be25d327456703eb310208f2fb72e8e9
c5be2acd278fd317eadc1e984a41415f18893bf21d1fd96afcf7959308543ff8
2e56ae84a9f133612a2fb4e17e5c803b07a8447552afcb645bfba201540aeb8f
07c6d5d0a6a2ac8d0bc6c74d8bf5cb8dcea8b5d4bf3c28efa3353517edea8f6c
a2f4a7cd5b7e50d6b7b9abf3c2a986a0bb5f6df3848bd5bd98097af684978f18
583054c0e13a84b815434e3b6dec85b1dde8c35782023d7fb0a515a67eb641d3
1b6c00c6697a973157c9176a39c1bd9605bbd2d4b9961fec3b0cd6ac0b6cb82b
4fb6a5547cc50eec1b179616037feff141ee98bf5c54983a7d3bed7c9f013143
d8abe52ee7eedc1914ceb971b17af1462e47b76cc0f15315c65e9c021ac4c1ff
d18f21ab230bf2fa0f9deeac8f7a879eca038c817220123955ae0bcacce8ce8e
430ae3803a98c82f4bdc40af6228d3c6d68a771d2fc1ddd9ad8213fa9dbd03fe
e337b279a856ec17ca45ab620f1efee60dc033361d43a7cd77fd41f18e24c650
99e77afbf1f9fe676db687d47b8ff1d6752e9594d41505bc82f862ec36a3e11f

第 15 层:
86bbefecf4d9b38a73ea799976338840b0afa35abb51026de1725524bd64bda3
e187d4d73a3c14bd9e36f32f172ed8e0c916243cc50b3aabc035b9515541bc39
bfc4bebeea53be9dd0fbb5ac5d881e6b5e90c961e235f5df0455dd95c6fa2506
fcf26d76536731ba974fc3441e5269b90307a9487145899c8181bb56ddb58aee
af43e509b9fe32d4eba27937b0dec2271b6a3bb9c6b2f1dca05c19a0e814cd2e
8f71bc9785b6903f7dc4c6b70ad9571a73442f4d8b1be8f3177c7723d979b197
99e77afbf1f9fe676db687d47b8ff1d6752e9594d41505bc82f862ec36a3e11f

第 16 层:
caf1c034fdd6e533cfc6e9c4842499841ec99aea9127f894ead533dc9c4f3b70
1504312e5d90ffea4c8de94ad5e04465ffb509c29e254e498433ee20dc74e51c
d8b44a55b842774ee810a510c788b2d8691132bf7590e6f5b85b07244a245d01
99e77afbf1f9fe676db687d47b8ff1d6752e9594d41505bc82f862ec36a3e11f

第 17 层:
d5db17981bbae0f753fc66ac04ecb1badb6b09a954f38ad5ea584a9549a34060
9940b80b01132e06e7111c92a6095d52afda304cf05f2578016311a8b8710ba5

第 18 层:
6191ccc9f8fbe88c36b105e600a3bb45e955a5a7468d00af9b0cbdc98705d9f5


生成Merkle树所用的时间: 889.1836037635803 s
```

进行查找，运行结果如下：

```
输入结点：18
输入结点的序号：18
序号：18 字符：18
查找路径:
第 1 层的查找值：7184f58245bc58f1e0f59200bdeedec7017ef35a71ef54de0f9629b4fef730a0
        生成的Hash值：bdcc66fe4a6cee4358809f091137ddbc116295d52c8d973d6d799a7ad7114da7
第 2 层的查找值：31d0e2b5f411aa2ab113c0752ccdc47ccbcb0c4496f1f8216f96a77493cc56ec
        生成的Hash值：0dfeb2314186e2961145704e4defe22d9d8a52be10056ba028c05239c91f4f4f
第 3 层的查找值：441abababeb97f088be1da9239bf1c32e4a220098b0c5ea7d61a93c10432a9aa5e
        生成的Hash值：980d7ae3084bc62e4f6928314e7be79a04c3532897ba5b774c7f7fcd72e4b684
第 4 层的查找值：a230f50723789826f04ee4048d32814b90a4bdfbeecdf7759478ee1794acf64f
        生成的Hash值：27b5ea5265b18f9938dac41ab31df24c91ab778013205c3fb50d07a365d84763
第 5 层的查找值：19040f40a98aa82b160bd63a2d1a3ccf1421ee19d9b339b90cff8f66e8830525
        生成的Hash值：5c065ee189c121f6fbb998ad1a8f55c70329b0bf6dfdba84489599dc9408366c
第 6 层的查找值：20bc8c70d66d426524f269001cc47c70f0a53f39f09f016127f75d5176987da2
        生成的Hash值：08fe4160949301a5fb321bd2895c58c840e098d46f6722f449c1c8025ccee120
第 7 层的查找值：64cae0469a19ec5b7f30b6f016b912970c564df616e5b0d4fe1c7fe9a945df2f
        生成的Hash值：8ebc573ac215a2d37ada2e57674bada9f2ce878d2afc24058adc45fc3941a9e9
第 8 层的查找值：ad1e67a7450e51b6997eed94b20bbdb50aa8a2348bcb55988b1936671bfffda7
        生成的Hash值：0221f1ad57c0419bb2a105d7bd665b436cd0657e82d3d56ca1d00d5f57c967c5
第 9 层的查找值：51e255e8c1020898e3bd38263ecd905d3955e828c964a7f7eaf74074031cada6
        生成的Hash值：b7fa906b79abbc56babf2d520a52e58d8973833c49cfa2ff9491347e4cb3510d
第 10 层的查找值：b11f458b4958ddcc2b50b02982ffe79176550f496071b8b37ee5e767336a3109
        生成的Hash值：09c375d93c0d343385ef14a9e0fe5c567ed52b6d24a027618409ef1134dad544
第 11 层的查找值：1fa7bbddab3b0f1ca126a18efa4bb02ff0290eb9b2e003648d1a222d015ba5ca
        生成的Hash值：6db5f9c75bd50236213ad641ceac77f2d56d496fb0f70d00d48d70a31d5066cc
第 12 层的查找值：a0c101c792e52e6214150849c2f192380e14d86a4c1db147f4ce43d146c3bb16
        生成的Hash值：94e619642bd3b8f226af5a6c875a1c5146c2f4a9529ec8b6e987fc40d84771dc
第 13 层的查找值：94fc7230afa396376019869b730af24211628d528e19ec0cc30a598c485f8060
        生成的Hash值：c089da8360be0503314367c6dfabb2a2be25d327456703eb310208f2fb72e8e9
第 14 层的查找值：c5be2acd278fd317eadc1e984a41415f18893bf21d1fd96afcf7959308543ff8
        生成的Hash值：86bbefecf4d9b38a73ea799976338840b0afa35abb51026de1725524bd64bda3
第 15 层的查找值：e187d4d73a3c14bd9e36f32f172ed8e0c916243cc50b3aabc035b9515541bc39
        生成的Hash值：caf1c034fdd6e533cfc6e9c4842499841ec99aea9127f894ead533dc9c4f3b70
第 16 层的查找值：1504312e5d90ffea4c8de94ad5e04465ffb509c29e254e498433ee20dc74e51c
        生成的Hash值：d5db17981bbae0f753fc66ac04ecb1badb6b09a954f38ad5ea584a9549a34060
第 17 层的查找值：9940b80b01132e06e7111c92a6095d52afda304cf05f2578016311a8b8710ba5
        生成的Hash值：6191ccc9f8fbe88c36b105e600a3bb45e955a5a7468d00af9b0cbdc98705d9f5

根结点哈希值：6191ccc9f8fbe88c36b105e600a3bb45e955a5a7468d00af9b0cbdc98705d9f5
结点 18 在Merkle树中
查找所用时间为： 1.562164068222046 s
```

当输入的结点序号不对应时，有如下结果：

```
生成Merkle树所用的时间：747.8196933269501 s
输入结点：18
输入结点的序号：19
序号：19 字符：18
查找路径:
第 1 层的查找值：e013d94419e62efaabeb11e5e7143c430e4464b9a0899836dccbf19727a1967f
        生成的Hash值：1851f485fcbeccc64be64f0aae9ccc70f157f8801e6c08a2cbfdca890c0dc778
第 2 层的查找值：31d0e2b5f411aa2ab113c0752ccdc47ccbcb0c4496f1f8216f96a77493cc56ec
        生成的Hash值：82bca112c90a4dee6d4a6e4bba3158209bc0e3634e95e4e41c77d97f4c842578
第 3 层的查找值：441abababeb97f088be1da9239bf1c32e4a220098b0c5ea7d61a93c10432a9aa5e
        生成的Hash值：a99a88e28d7b2d5e1aa4ede44c2915829472b3ee9805a5e0690c2c3fe4761d8c
第 4 层的查找值：a230f50723789826f04ee4048d32814b90a4bdfbeecdf7759478ee1794acf64f
        生成的Hash值：87b5e4c2274dc0c8c4d63f78b7b83b523b1ae395066c2d134674786773073a979
第 5 层的查找值：19040f40a98aa82b160bd63a2d1a3ccf1421ee19d9b339b90cff8f66e8830525
        生成的Hash值：61b95bb0acb4342259d0d246dcef31fff95ec17e399c052fb6597098f1d97931
第 6 层的查找值：20bc8c70d66d426524f269001cc47c70f0a53f39f09f016127f75d5176987da2
        生成的Hash值：19c12cbbc3bd54a2366cd3d96a426483bdc62289bac2f2da7f47e74204970f20
第 7 层的查找值：64cae0469a19ec5b7f30b6f016b912970c564df616e5b0d4fe1c7fe9a945df2f
        生成的Hash值：c2a4ffbaef09aa082125aef7664af9af17a9a175d836e83d7fe595ec97d1e4ff
第 8 层的查找值：ad1e67a7450e51b6997eed94b20bbdb50aa8a2348bcb55988b1936671bfffda7
        生成的Hash值：dcc7fd52c5ba260e9087313bb1a8f7d858a279aeedc6fafef3645d9d5e967303
第 9 层的查找值：51e255e8c1020898e3bd38263ecd905d3955e828c964a7f7eaf74074031cada6
        生成的Hash值：7bdfe0c64457fa21697d108cf8d633f438e58fa3a556a75fed4576555a76af32
第 10 层的查找值：b11f458b4958ddcc2b50b02982ffe79176550f496071b8b37ee5e767336a3109
        生成的Hash值：f231a28e1f4a499e1dc6f52cde551e4e51e52bc14fc471e846f10519a2a60f8a
第 11 层的查找值：1fa7bbddab3b0f1ca126a18efa4bb02ff0290eb9b2e003648d1a222d015ba5ca
        生成的Hash值：6e2e22e96528bbdeb4aa46677b94bf28f03ccb907b6288035c28fcab8a8006a3
第 12 层的查找值：a0c101c792e52e6214150849c2f192380e14d86a4c1db147f4ce43d146c3bb16
        生成的Hash值：72e62b434468d1d8c68637e067be2bed13ad27c8303ff152912f946634f60492
第 13 层的查找值：94fc7230afa396376019869b730af24211628d528e19ec0cc30a598c485f8060
        生成的Hash值：c8a71c204cce80569b75ce926d94bc2221fa2134a84e8190672dfc879e967729
第 14 层的查找值：c5be2acd278fd317eadc1e984a41415f18893bf21d1fd96afcf7959308543ff8
        生成的Hash值：fda7e4c4d3d8ea98f7552af5f54fb4ec171c51e40c2f5c69b9846e9aaa43c3e5
第 15 层的查找值：e187d4d73a3c14bd9e36f32f172ed8e0c916243cc50b3aabc035b9515541bc39
        生成的Hash值：3dd9a96840fa9c5938014240b9bb5a7bd4c04e1ae3e8da12df8df1fec9ea313c
第 16 层的查找值：1504312e5d90ffea4c8de94ad5e04465ffb509c29e254e498433ee20dc74e51c
        生成的Hash值：ac14a2827dee7c889f71f832f0ce1664bfe7e5305c8baf41b532b6ea04ec6482
第 17 层的查找值：9940b80b01132e06e7111c92a6095d52afda304cf05f2578016311a8b8710ba5
        生成的Hash值：c59ac4bd53977677936c7027361c955124d16123dc93c22a8b534057f3c4c5b2

根结点哈希值：6191ccc9f8fbe88c36b105e600a3bb45e955a5a7468d00af9b0cbdc98705d9f5
结点 18 不在Merkle树中，或者是序号错误
查找所用时间为： 1.5551669597625732 s
```

当输入的结点序号大于 Merkle 树的规模时，即结点序号错误时，有如下结果：

```
输入结点：18
输入结点的序号：100003
结点序号错误！
查找所用时间为：  0.008509635925292969 s
```

6.Project6:impl this protocol with actual network communication

未做

7.Project7:Try to Implement this scheme

未做

8.Project8:AES impl with ARM instruction

未做

9.Project9:AES / SM4 software implementation

9.1 实现方式

本 project 要实现 SM4 的优化。

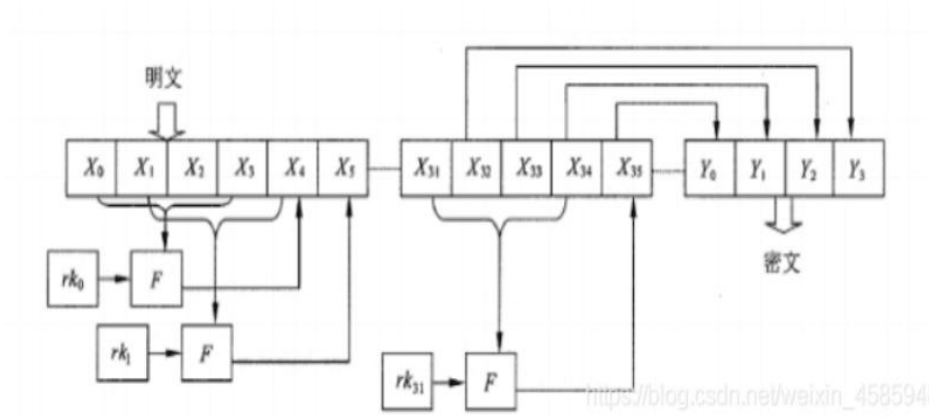
基本原理：SM4 密码算法的数据分组长度为 128 比特,密钥长度也是 128 比特，是分 组算法当中的一种。加密算法、密钥扩展算法都为 32 轮非线性迭代结构，以字 （32 位）为单位进行加密，每一次迭代运算均为一轮变换函数 F。SM4 算法加/解密算法的结构相同，只是使用轮密钥相反，其中解密轮密钥是加密轮密钥的逆序。

SM4 加密算法：

采用 32 轮迭代结构来作为加密算法，每轮使用一个轮密钥。设输入的明文为四个字(X_0，X_1，X_2，X_3)，一共有 128 位。输入轮密钥为 rk_i,i=0,1,…，31，一共 32 个字。输出密文为四个字(Y_0，Y_1，Y_2，Y_3)，128 位。

则这个加密算法可描述如下：

$$\begin{cases} X_{i+4}=F(X_i,\ X_{i+1},\ X_{i+2},\ X_{i+3},\ rki) \\ \quad =X_i \oplus T(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rki),\ i=0,1 \cdots 31 \\ (Y_0,\ Y_1,\ Y_2,\ Y_3)=(X_{35},\ X_{34},\ X_{33},\ X_{32}) \end{cases}$$



SM4 的加密算法和 DES、AES 的结构一样，均采用了基本函数迭代，但 SM4 也有一些不同的加密迭代处理特点。由图可以看出，SM4 的加密法代处理方式具有密文反馈连接和流密码的某一些特点，前一轮加密的结果与前一轮的加密数据拼接起来供下一轮加密处理。一次加密处理四个字，然后产生一个字的中间密文，这个中间密文和前三个字拼接起来后再供下一次加密处理，一共会迭代加密处理 32 轮，产生出四个字的密文。这个加密处理的整个过程就像一个宽度为 4 个字的窗口在滑动，加密处理完一轮，窗口就滑动一个字，窗口滑动 32 次后，加密迭代就结束了。

SM4 解密算法：

由于 SM4 密码算法的运算是对合运算，所以它的解密算法结构和加密算法的结构一样，但轮密钥的使用顺序相反，也就是说解密的轮密钥是加密的轮密钥的逆序。$(X_0,\ X_1,\ X_2,\ X_3)$，一共有 128 位。输入轮密钥为 $rk_i, i=0,1,\cdots,31$，一共 32 个字。输出的明文为$(Y_0,\ Y_1,\ Y_2,\ Y_3)$。

则这个解密算法可描述如下：

$$\begin{cases} X_{i+4}=F(X_i,\ X_{i+1},\ X_{i+2},\ X_{i+3},\ rki) \\ \quad =X_i \oplus T(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rki),\ i=31,30,\ \cdots 1,0 \\ (Y_0,\ Y_1,\ Y_2,\ Y_3)=(X_{35},\ X_{34},\ X_{33},\ X_{32}) \end{cases}$$

其中，轮函数 F 和合成置换 T 描述如下：

轮函数 F：SM4 密码算法的轮函数是一种以字为处理单位的密码函数。

设轮函数 F 的输入为(X_0, X_1, X_2, X_3) ，四个 32 位字，共 128 位。轮密钥为 rk，也是一个 32 位的字。其输出也是 32 位的字。

$F(X_0, X_1, X_2, X_3) = X_0 \oplus T(X_1 \oplus X_2 \oplus X_3 \oplus rk)$

合成置换 T:由非线性变换τ和线性变换 L 的转换，T (.) = L (τ (.))

非线性变换τ : 长度为 32 bits， 由 4 个并行的 8 位输入输出的 S 盒组成，表示为 S b o x (.)

$(B_0, B_1, B_2, B_3) = \tau(A) = (Sbox(A_0), Sbox(A_1), Sbox(A_2), Sbox(A_3))$

线性变换 L :B = L (A) = A ⊕ (A ＜＜＜ 2) ⊕ (A ＜＜＜ 10) ⊕ (A ＜＜＜ 18) ⊕ (A ＜＜＜ 24) 山东大学

SM4 密钥扩展算法：

SM4 密码算法采用 32 轮的迭代加密结构，拥有 128 位加密密钥，一共使用 32 轮密钥，每一轮的加密使用 32 位的一个轮密钥。SM4 算法的特点使得它需要使用一个密钥扩展算法，在加密密钥当中产生 32 个轮密钥。在这个密钥的扩展算法当中有常数 FK、固定参数 CK 这两个数值，利用这两个数值便可完成它的这一个扩展算法。

1.常数 FK

密钥扩展当中使用的常数为以下几个：

FK_0=(A3B1BAC6)，FK_1=(56AA3350),FK_2=(677D9197)，FK_3=(B27022DC)

2.固定参数 CK

一共使用有 32 个固定参数 CK_i，这个 CK_i，是一个字，它的产生规则是：

设 $ck_{i,j}$ 为 CK_i 的第 j 字节 (i=0 ，1 ，… ，31;j=0,1,2,3) ，

即 $CK_j=(ck_{i,0}, ck_{i,1}, ck_{i,2}, ck_{i,3})$,则 $ck_{i,j}=(4i+j)\times 7 (\mathrm{mod}\ 256)$

这 32 个固定参数如下(十六进制) :

00070e15， 1c232a31， 383f64d， 545b6269，

70777e85， 8c939aa1， a8afb6bd， c4cbd2d9，

e0e7eef5， fc030a11， 181f262d， 343b4249，

50575e65， 6c737a81， 888f969d， a4abb2b9，

c0c7ced5， dce3eaf1， f8ff060d， 141b2229，

30373e45， 4c535a61， 686f767d， 848b9299，

a0a7aeb5， bcc3cad1， d8dfe6ed， f4fb0209，

10171e25， 2c333a41， 484f565d， 646b7279。

假设输入的加密密钥为 $MK=(MK_0, MK_1, MK_2, MK_3)$，输出的轮密钥为 rk_i，$i=0,1\cdots,30,31$，中间的数据为 $K_i=0,1,\cdots,34,35$。

则密钥扩展算法可描述如下：

①$(K_0, K_1, K_2, K_3)=(MK_0\oplus FK_0, MK_1\oplus FK_1, MK_2\oplus FK_2, MK_3\oplus FK_3)$

②For $i=0,1,\cdots,30,31$ Do

$rk_i=K_{(i+4)}=K_i\oplus T'$（$K_{(i+1)}\oplus K_{(i+2)}\oplus K_{(i+3)}\oplus CK_i$）

说明:其中的 T’变换与加密算法轮函数中的 T 基本相同，只将其中的线性变换 L 修改为以下的 L’:$L'(B)=B\oplus(B<<<13)\oplus(B<<<23)$

从密钥扩展算法中我们分析后可以发现，密钥扩展算法与加密算法在算法结构方面类似，同样都是采用了 32 轮类似的迭代处理，需要特别注意的是密钥扩展算法采用了非线性变换 T，这个措施将会使密钥扩展的安全性大大增强。

要实现对 SM4 的优化，可以对常规的密钥扩展方案进行优化，不再每次都进行密钥扩展的计算，而是通过查表即 S 盒，从而节省了时间，提高加解密的效率。

由以上原理，可以实现 SM4 加解密算法及其优化，**相关代码如下：**

加解密函数：

```
1.    //函数定义
2.    //加密/解密
3.    void SM4(uint8_t plaintex[16], uint8_t ciphertext[16], uint32_t RK[32], bool decryption = 0)
4.    {
5.        uint32_t X[4];
6.        uint32_t X_temp;
7.        uint32_t B;
```

```
        Byte_to_Ulong(X[0], plaintex, 12)
            Byte_to_Ulong(X[1], plaintex, 8)
            Byte_to_Ulong(X[2], plaintex, 4)
            Byte_to_Ulong(X[3], plaintex, 0)

            for (int i = 0; i < 32; i++)
            {

                X_temp = (decryption == 0) ? RK[i] : RK[31 - i];
                X_temp = X[1] ^ X[2] ^ X[3] ^ X_temp;
                B = X[0];
                B ^= S_Box0[(X_temp >> 24) & 0xff];
                B ^= S_Box1[(X_temp >> 16) & 0xff];
                B ^= S_Box2[(X_temp >> 8) & 0xff];
                B ^= S_Box3[(X_temp >> 0) & 0xff];

                X[0] = X[1];
                X[1] = X[2];
                X[2] = X[3];
                X[3] = B;
            }
        Ulong_to_Byte(X[0], ciphertext, 12)
            Ulong_to_Byte(X[1], ciphertext, 8)
            Ulong_to_Byte(X[2], ciphertext, 4)
            Ulong_to_Byte(X[3], ciphertext, 0)
    }
```

密钥扩展 F 函数：

```
    //密钥拓展 F 函数
    static uint32_t SM4_F_key(uint32_t K0, uint32_t K1, uint32_t K2, uin
t32_t K3, uint32_t ck)
    {
        uint8_t b[4];
        uint32_t res;
        uint32_t Ulong = K1 ^ K2 ^ K3 ^ ck;
        Ulong_to_Byte(Ulong, b, 0)
            b[0] = S_Box[b[0]];
        b[1] = S_Box[b[1]];
        b[2] = S_Box[b[2]];
        b[3] = S_Box[b[3]];
        res = ((uint32_t)b[0] << 24) | ((uint32_t)b[1] << 16) | ((uint32
_t)b[2] << 8) | ((uint32_t)b[3]);
        return (K0 ^ res ^ (Shift_left(res, 13)) ^ (Shift_left(res, 23)))
;
```

```
}
```

根据密钥扩展函数实现密钥扩展，即普通的密钥扩展方法：

```
//不查表优化密钥拓展
void SM4_generate_key_v1(uint32_t RK[32], uint8_t key[16])
{
    uint32_t MK[4], K[36];
    Byte_to_Ulong(MK[0], key, 12)
        Byte_to_Ulong(MK[1], key, 8)
        Byte_to_Ulong(MK[2], key, 4)
        Byte_to_Ulong(MK[3], key, 0)
        K[0] = MK[0] ^ FK[0];
    K[1] = MK[1] ^ FK[1];
    K[2] = MK[2] ^ FK[2];
    K[3] = MK[3] ^ FK[3];
    for (int i = 0; i < 32; i++)
    {
        K[i + 4] = SM4_F_key(K[i], K[i + 1], K[i + 2], K[i + 3], CK[i]);
        RK[i] = K[i + 4];
    }
}
```

优化后的密钥扩展方案：

```
//查表（“S 盒”）优化密钥拓展
void SM4_generate_key_v2(uint32_t RK[32], uint8_t key[16])
{
    uint32_t MK[4], K[4], K_temp, temp;
    Byte_to_Ulong(MK[0], key, 12)
        Byte_to_Ulong(MK[1], key, 8)
        Byte_to_Ulong(MK[2], key, 4)
        Byte_to_Ulong(MK[3], key, 0)
        K[0] = MK[0] ^ FK[0];
    K[1] = MK[1] ^ FK[1];
    K[2] = MK[2] ^ FK[2];
    K[3] = MK[3] ^ FK[3];
    for (int i = 0; i < 32; i++)
    {
        K_temp = K[1] ^ K[2] ^ K[3] ^ CK[i];
        temp = K[0];
        temp ^= SS_Box0[(K_temp >> 24) & 0xff];
        temp ^= SS_Box1[(K_temp >> 16) & 0xff];
        temp ^= SS_Box2[(K_temp >> 8) & 0xff];
        temp ^= SS_Box3[(K_temp >> 0) & 0xff];
```

```
21.
22.             K[0] = K[1];
23.             K[1] = K[2];
24.             K[2] = K[3];
25.             K[3] = temp;
26.             RK[i] = temp;
27.         }
28.     }
```

具体实现过程：

```
1.     //SM4 加密、SM4 解密、密钥扩展优化
2.     int main()
3.     {
4.         clock_t start_time, end_time;//时间
5.
6.         uint8_t plaintext[16] = { 0x10,0x32,0x54,0x76,0x98,0xBA,0xDC,0xF
   E,0xEF,0xCD,0xAB,0x89,0x67,0x45,0x23,0x01 };//明文
7.         uint8_t ciphertext[16];
                                                    //密文
8.         uint8_t key_seed[16] = { 0x10,0x32,0x54,0x76,0x98,0xBA,0xDC,0xFE,
   0xEF,0xCD,0xAB,0x89,0x67,0x45,0x23,0x01 };//密钥
9.         uint32_t RK[32];
                                                    //密钥扩展
10.
11.        //生成轮密钥
12.        SM4_generate_key_v1(RK, key_seed);
13.
14.        //SM4 加密
15.        SM4(plaintext, ciphertext, RK, 0);
16.        cout << "加密结果:" << endl;
17.        for (int i = 0; i < 16; i++)
18.            cout << "0x" << hex << setw(2) << setfill('0') << (int)ciphe
   rtext[i] << "\t";
19.
20.        //SM4 解密
21.        cout << endl << "解密结果:" << endl;
22.        SM4(plaintext, ciphertext, RK, 1);
23.        for (int i = 0; i < 16; i++)
24.            cout << hex << "0x" << (int)(plaintex[i] >> 4) << (int)(plai
   ntext[i] & 0xf) << "\t";
25.        cout << "\n 正确的结果:" << endl;
26.        for (int i = 0; i < 16; i++)
27.            cout << "0x" << hex << setw(2) << setfill('0') << (int)plain
   text[i] << "\t";
```

```
28.
29.
30.         //使用查表优化的 SM4 加密
31.         start_time = clock();
32.         for (int i = 0; i < pow(10, 6); i++)
33.         {
34.             SM4_generate_key_v2(RK, key_seed);
35.             SM4(plaintext, ciphertext, RK, 0);
36.         }
37.         end_time = clock();
38.          cout << endl << "\n 使用查表优化:\nSM4 加密 pow(10,6)次总时间为
    " << (double)(end_time - start_time) / CLOCKS_PER_SEC << "s";
39.
40.         //不使用查表优化的 SM4 加密
41.         start_time = clock();
42.         for (int i = 0; i < pow(10, 6); i++)
43.         {
44.             SM4_generate_key_v1(RK, key_seed);
45.             SM4(plaintext, ciphertext, RK, 0);
46.         }
47.         end_time = clock();
48.          cout << endl << "\n 不使用查表优化:\nSM4 加密 pow(10,6)次总时间为
    " << (double)(end_time - start_time) / CLOCKS_PER_SEC << "s";
49.
50.         cout << endl;
51.
52.         return 0;
53.     }
```

9.2 实现效果

运行代码，有如下结果：

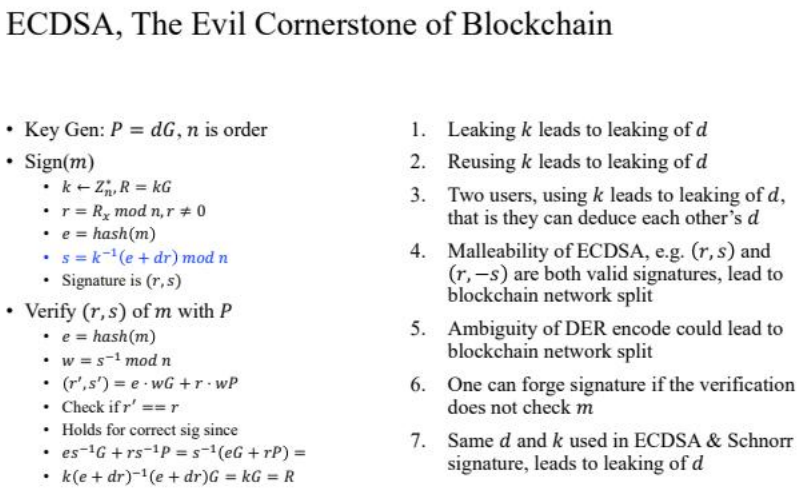
```
加密结果:
0x68    0x1e    0xdf    0x34    0xd2    0x06    0x96    0x5e    0x86    0xb3    0xe9    0x4f    0x53    0x6e    0x42    0
x46
解密结果:
0x10    0x32    0x54    0x76    0x98    0xba    0xdc    0xfe    0xef    0xcd    0xab    0x89    0x67    0x45    0x23    0
x01
正确的结果:
0x10    0x32    0x54    0x76    0x98    0xba    0xdc    0xfe    0xef    0xcd    0xab    0x89    0x67    0x45    0x23    0
x01

使用查表优化:
SM4加密pow(10,6)次总时间为0.688s

不使用查表优化:
SM4加密pow(10,6)次总时间为1.634s
```

可以发现，对密钥扩展方案的修改，使得 SM4 加密速度变快，效率提高，从而成功实现了 SM4 的优化。

10. Project10:report on the application of this deduce technique in Ethereum with ECDSA

10.1 实现方式

基本原理：关于 ECDSA 签名、签名的验证，示意图如下：



关于从签名获取公钥，可以由 ECDSA 签名、验证的过程逆推得到。

相关步骤代码如下：

ECDSA 签名：

```
#ECDSA 签名
def ECDSA_sig(d, m):
    k = random.randint(1, n - 1)
    kG = ECG_k_point(k, point_G)
    #求 r 和 s
    r = ele_to_int(kG.x)
    s = ((config.inverse(k, n)) * (int(hash_sha3_256(m), 2) + d * r)) % n
    return r, s
```

ECDSA 签名的验证：

```
def ECDSA_ver(r, s, P, m):#验证
    e = int(hash_sha3_256(m), 2)
    w = config.inverse(s, n)
    #计算(r',s')
```

```
        r_ = ECG_k_point((w * e) % n, point_G)
        s_ = ECG_k_point((w * r) % n, P)
        r_s_ = ECG_ele_add(r_, s_)
        r_ver = ele_to_int(r_s_.x)
        if r == r_ver:#验证
            return True
        else:
            return False
```

从签名获取公钥：

```
    #从签名获取公钥
    def ECDSA_deduce_pk_sig(m, r, s):
        e = int(hash_sha3_256(m), 2)
        s_inverse = config.inverse(s, n)
        #计算 r 对应的域元素，即带入椭圆曲线方程 x^3+ax+b
        r_ele = (pow(r, 3) % q + (a * r) % q + b) % q
        #解集合 y，根据公钥点 r_ele 值通过 Cipolla 算法求解 y 值，二次同余方程两个解
        P_possible = [Cipolla.Cipolla(r_ele, q), q - Cipolla.Cipolla(r_ele, q)]
        #将字节串 bin(r)转为域元素，输入的是模数 n 和字节串 S
        R = bytes_to_ele(n, bits_to_bytes(bin(r)))
        #猜测的公钥集合
        P_Guess = []
        for p_possible in P_possible:
            #将字节串 bin(p_possible)转为域元素，输入的是模数 n 和字节串 S
            p_possible = bytes_to_ele(n, bits_to_bytes(bin(p_possible)))

            #求解 r*s_inverse*P(Point_r_s_inverse_P)的椭圆曲线上的点
            Point_r_s_inverse_P = ECG_ele_add(Point(R, p_possible),
                                              ECG_k_point(((n - 1) * e * s_inverse) % n, point_G))
            #求解可能的公钥 P_g=[(r*s_inverse)^(-1)]Point_r_w_P
            P_g = ECG_k_point(config.inverse(r * s_inverse, n), Point_r_s_inverse_P)
            P_Guess.append(P_g)
        #返回可能的公钥集合
        return P_Guess
```

验证部分代码如下：

```
    if __name__ == "__main__":
    #初始化
    config.set_default_config()
    point_G = Point(config.get_Gx(), config.get_Gy())
```

```
q = config.get_q()
a = config.get_a()
b = config.get_b()
n = config.get_n()
h = config.get_h()
m = "Ge Mengyun"
#生成公私钥对
d = randint(1, n - 1)
P = ECG_k_point(d, point_G)
print("私钥 d:", d, "\n 公钥 P:", P)
#生成签名
start_time = time.time()
r, s = ECDSA_sig(d, m)
end_time = time.time()
print("\n 签名成功！\nr:", r, "\ns:", s)
print("生成签名所用时间为:", end_time - start_time, "s")
#验证签名
start_time = time.time()
if ECDSA_ver(r, s, P, m):
    end_time = time.time()
    print("\n 验证通过！")
    print("验证签名所用时间为:", end_time - start_time, "s")
#由签名值恢复公钥
start_time = time.time()
P_guess = ECDSA_deduce_pk_sig(m, r, s)
if P.x == P_guess[0].x and P.y == P_guess[0].y:
    end_time = time.time()
    print("\n 公钥恢复成功！\n 恢复的公钥为:", P_guess[0])
elif P.x == P_guess[1].x and P.y == P_guess[1].y:
    end_time = time.time()
    print("\n 公钥恢复成功！\n 恢复的公钥为:", P_guess[1])
print("恢复公钥所用时间为:", end_time - start_time, "s")
```

注：其中，有限域的运算、椭圆曲线上点的运算直接使用了开源的 SM2。

10.2 实现效果

代码的运行结果如下（部分）：

```
私钥d: 121936471882739024156204051254309809264649794284714112887234014720897684721
1
公钥P: (99922511236625999239540398394154679671617752917804590101331775412276889190628, 92132670799160167939035450340089576364803144274460807030707335895556903575828)

签名成功!
r: 54112506988199820089231210716552191182860925205956197813966380628078705598138

s: 11392463600861752597387533887330066260696254337101993067741355955538069190383
1
生成签名所用时间为: 0.2198793888092041 s

验证通过!
验证签名所用时间为: 0.4376997947692871 s
恢复公钥所用时间为: -0.01349496841430664 s
```

11.Project11:impl sm2 with RFC6979

11.1 实现方式

基本原理：

以下为 RFC6979 的主要步骤：

①用哈希函数 H 计算 h1 = H(m)；

②设 V = 0x01 0x01 0x01 ... 0x01，此处 V 的长度等于 8 * ceil（hlen / 8），单位为比特；

③设 K = 0x00 0x00 0x00 ... 0x00，这里 K 的长度与 V 相等，即 32 个 8 位字节；

④算 K = HMAC_K(V || 0x00 || int2octets(x) || bits2octets(h1))， x 为私钥；

⑤算 V = HMAC_K(V)；

⑥算 K = HMAC_K(V || 0x00 || int2octets(x) || bits2octets(h1))；

⑦算 V = HMAC_K(V)；

⑧将 T 设置为空序列，进行判断：若 lent＜256，则算 V = HMAC_K(V)并设置 T = T || V，直至 lent ＝256；

⑨若 k∈[1,q-1]成功找到了随机数，即 k 满足 r!=0 和 r+s!=n 在签名算法中判断；

⑩若不满足 k∈[1,q-1]，则 K = HMAC_K(V || 0x00)，重新生成 T，进行新一轮的寻找。

此 project 在 SM2 的签名与验证的基础上实现。可以把 SM2 签名算法中生成 k 值的随机算法替换为使用 RFC6979 生成随机值。

相关步骤代码如下：

用 RFC6979 生成随机数：

```
#长度为 256bit
f RFC6979_gen_k(key, m):
  # 1、h1 = H(m)
  h1 = hash_sha3_256(m)
  # 2、V = 0x01 0x01 0x01 ... 0x01
  V = "00000001" * 32
  # 3、K = 0x00 0x00 0x00 ... 0x00
  K = "00000000" * 32
  key_no0b = remove_0b_at_beginning(bin(key))
  key_pad_zero = padzeore_to_len(key_no0b, 256)
  # 4、K = HMAC_K(V || 0x00 || int2octets(x) || bits2octets(h1))
  K = HMAC_K(K, V + "00000000" + key_pad_zero + h1)
  # 5、V = HMAC_K(V)
  V = HMAC_K(K, V)
  # 6、K = HMAC_K(V || 0x01 || int2octets(x) || bits2octets(h1))
  K = HMAC_K(K, V + "00000001" + key_pad_zero + h1)
  # 7、V = HMAC_K(V)
  V = HMAC_K(K, V)

  while True:
      T = ""
      while len(T) < 256:
          V = HMAC_K(K, V)
          T = T + V
      k = int(T, 2)
      #若 8、k∈[1,q-1]则成功
      #是否满足 r!=0、r+s!=n 于签名算法中判定

      if 0 < k < q:
          print("\n 找到合适的随机数 k。\n 生成的随机数为:", k)
          break
      #9、K = HMAC_K(V || 0x00)，重新生成 T
      K = HMAC_K(K, V + "00000000")
      V = HMAC_K(K, V)
```

```
    return k
```

SM2 签名与验证，注意其中签名算法中 k 的生成不同于原本的 SM2 算法，原算法使用随机算法生成随机数 k，此处使用 RFC6979 生成随机数 k，代码如下：

```
#SM2 生成签名
def SM2_sig(IDA, dA, PA):
    ZA = get_Z(IDA, PA)
    #令 M1=ZA // M
    M1 = ZA + M
    #计算 e = Hv(M1)，同时进行类型转换
    e = bytes_to_int(bits_to_bytes(hash_function(M1)))
    while True:
        #用 RFC6979 生成 k∈[1,n-1]
        k = RFC6979_gen_k(dA, M)
        #算椭圆曲线点(x1,y1)=[k]G
        X = ECG_k_point(k, point_g)
        #进行 x1 的类型转换
        x1 = bytes_to_int(ele_to_bytes(X.x))
        #算 r=(e+x1) mod n，如果 r=0 或 r+k=n，就重新选择 k
        r = (e + x1) % n
        if r == 0 or r + k == n:
            continue
        #算 s = ((1 + dA)-1 ·(k-r·dA)) mod n，如果 s=0 就重新选择 k
        s = (config.inverse(1 + dA, n) * (k - r * dA)) % n
        if s == 0:
            continue
        break
    return r, s #消息 M 的签名为(r,s)


#SM2 签名验证
def SM2_ver(ID, r, s, PA):
    #验证 r,s∈[1,n-1]是否成立，如果不成立，验证不通过
    if r < 0 or r > n:
        print("r 验证不通过！")
        return -1
    if s < 0 or s > n:
        print("s 验证不通过！")
        return -1
    ZA = get_Z(ID, PA)
    #置 M1=ZA // M
    M1 = ZA + M
    #算 e = Hv(M1)，进行 e 的数据类型转换
```

```
    e = bytes_to_int(bits_to_bytes(hash_function(M1)))
    #算 t=(r + s)mod n，若 t = 0，验证不通过
    t = (r + s) % n
    if t == 0:
        return -1
    #计算椭圆曲线点(x1,y1)=[s']G+[t]PA
    X = ECG_ele_add(ECG_k_point(s, point_g), ECG_k_point(t, PA))
    #进行 x1 的数据类型转换，计算 R=(e+x1)mod n,看 R=r 是否成立
    x1 = bytes_to_int(ele_to_bytes(X.x))
    R = (e + x1) % n
    if R == r:
        return 1
    else:
        return -1
```

对使 RFC6979 生成 k 值的 SM2 的签名、验证算法进行验证：

```
if __name__ == '__main__':
    #初始化
    config.set_default_config()
    parameters = config.get_parameters()
    point_g = Point(config.get_Gx(), config.get_Gy())
    q = config.get_q()
    n = config.get_n()

    ID = "1938607452@qq.com"
    M = "Ge Mengyun"

    #生成公钥、私钥
    sk = random.randint(1, n - 1)
    pk = ECG_k_point(sk, point_g)
    print("私钥为:", sk)
    print("公钥为:", pk)

    #签名
    start_time = time.time()
    r, s = SM2_sig(ID, sk, pk)
    end_time = time.time()
    print("\n 成功生成签名，所用时间:", end_time - start_time, "s")
    print("r:", r)
    print("s:", s)

    #验证
    print("\n 验证签名")
    start_time = time.time()
```

```
29.         if SM2_ver(ID, r, s, pk):
30.             end_time = time.time()
31.             print("验证通过，所用时间:", end_time - start_time, "s")
```

11.2 实现效果

代码运行结果如下（部分）：

```
私钥为: 239417780209211983523389844293556366237115067066503393200258888915643116
76500
公钥为: (333821565771471344833636457132831865943808429259727485220752427751141128
731069, 989266826483809582292843913342194570391743196640196025246681771798628014
94082)

找到合适的随机数k。
生成的随机数为: 868403548610655429412508422396580042297854432248707720339524271 0
2220648045340

成功生成签名，所用时间: 0.24435830116271973 s
r: 424655551598091948202601338465406732435514777729337701169091626846838149767 27
s: 754634527436873738982277719099428921147144326074390800684436987288189367374 44

验证签名
验证通过，所用时间: 0.4363570213317871 s
```

12. Project12:verify the above pitfalls with proof-of-concept coded

12.1 实现方式

基本原理：本 project 要求用概念验证代码验证上述缺陷，既能够找出相关缺陷，实现对 ECDSA、Schnorr 和 SM2 签名的攻击。

针对 ECDSA 的攻击：

关于 ECDSA 的缺陷，主要有以下四项：

①k 的泄露也会导致私钥 d 的泄露；

②如果重复使用 k ，会导致私钥 d 的泄露；

③如果两个用户使用相同的 k ，会导致双方的私钥 d 的泄露；

④如果不验证 m，就可以实现签名的伪造；

基于以上缺陷，可以做出如下攻击：

①针对缺陷一，只需把签名 s 的过程逆过来，就能求出 d；

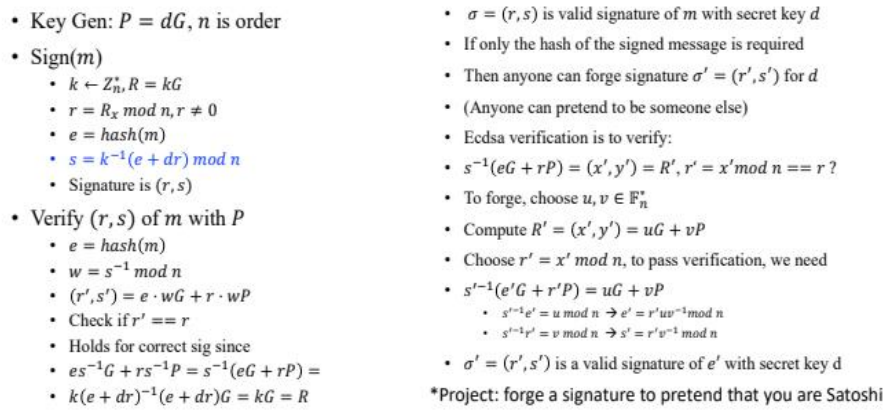
②针对缺陷二，只需把两个求签名 s 的式子联立起来，就能求出 k，从而同① 一样求出 d；

③针对缺陷三，如果两个用户使用相同的 k ，就相当于 k 已经泄露，则与①相似，只需按照①的方法求解；

④针对缺陷四，可以使用正确的 Satoshi 签名伪造 Satoshi 签名。

首先，需要生成正确的 Satoshi 签名，然后进行 Satoshi 签名的伪造，最后是通过正确的 Satoshi 签名对伪造的签名进行验证。

其流程图如下：



相关步骤代码如下：

定义 ECDSA 签名函数：

```
#指定 k 的 ECDSA 签名
def ECDSA_sign_with_certain_k(d, m, k):
    kG = ECG_k_point(k, point_G)
    #求 r 和 s
    r = ele_to_int(kG.x)
    s = ((config.inverse(k, n)) * (int(hash_sha3_256(m), 2) + d * r)) % n
    return r, s
```

针对缺陷一的攻击：

```
print("\n1、泄露 k 会导致泄露私钥 d:")
            start_time = time.time()
            m = "Ge Mengyun"
            k = random.randint(1, n - 1)
            print("泄露的 k 为", k)
            #ECDSA 签名
            r, s = ECDSA_sign_with_certain_k(d, m, k)
```

```
                #求私钥 d
                d_break = ((k * s - int(hash_sha3_256(m), 2)) * config.i
nverse(r, n)) % n
                if d_break == d:
                    print("攻击成功,求出的私钥 d 为:", d_break)
                    end_time = time.time()
                    print("花费时间为:", end_time - start_time, "s")
```

针对缺陷二的攻击：

```
    print("\n2、重复使用 k 会导致泄露私钥 d:")
                start_time = time.time()
                #使用同一个 k 签名两次
                m1 = "GeMeng yun"
                m2 = "Gemeng Yun"
                k = random.randint(1, n - 1)
                print("重复使用的 k 为", k)
                #ECDSA 签名
                r1, s1 = ECDSA_sign_with_certain_k(d, m1, k)
                r2, s2 = ECDSA_sign_with_certain_k(d, m2, k)
                #求私钥 d
                #k=(e2*r1-e1*r2)*(s2*r1-s1*r2)^(-1)
                k = ((int(hash_sha3_256(m2), 2) * r1 - int(hash_sha3_256(
m1), 2) * r2) *
                    config.inverse(s2 * r1 - s1 * r2, n)) % n
                d_break = ((k * s1 - int(hash_sha3_256(m1), 2)) * config.
inverse(r1, n)) % n
                if d_break == d:
                    print("攻击成功，求出的私钥 d 为:", d_break)
                    end_time = time.time()
                    print("花费时间为:", end_time - start_time, "s")
```

针对缺陷三的攻击：

```
    print("\n3、两个用户使用相同的 k 会导致泄露双方的私钥 d:")
                start_time = time.time()
                m1 = "GeMeng yun"
                m2 = "Gemeng Yun"
                k = random.randint(1, n - 1)
                print("使用的相同的 k 为", k)
                d1 = random.randint(1, n - 1)
                d2 = random.randint(1, n - 1)
                #ECDSA 签名
                r1, s1 = ECDSA_sign_with_certain_k(d1, m1, k)
                r2, s2 = ECDSA_sign_with_certain_k(d2, m2, k)
```

```
            d1_break = ((k * s1 - int(hash_sha3_256(m1), 2)) * config.inverse(r1, n)) % n
            d2_break = ((k * s2 - int(hash_sha3_256(m2), 2)) * config.inverse(r2, n)) % n
            if d2_break == d2:
                print("攻击成功，用户 1 破解用户 2 私钥 d2 为", d2_break)
            if d1_break == d1:
                print("攻击成功，用户 2 破解用户 1 私钥 d1 为", d1_break)
            end_time = time.time()
            print("花费的时间为:", end_time - start_time, "s")
```

针对缺陷四的攻击：

```
def Satoshi_sign(d, m):#生成正确的 Satoshi 签名
    k = randint(1, n - 1)
    kG = ECG_k_point(k, point_G)
    #求 e,r,s
    r = ele_to_int(kG.x)
    e = int(hash_sha3_256(m), 2)
    s = ((config.inverse(k, n)) * (e + d * r)) % n
    return e, r, s
#伪造 Satoshi 签名
def pretend_Satoshi_sig(G, P):
    #随机选择 u、v
    u = randint(1, n - 1)
    v = randint(1, n - 1)
    #计算伪造签名 R'值
    x_forge = ECG_k_point(u, G)
    y_forge = ECG_k_point(v, P)
    R_forge = ECG_ele_add(x_forge, y_forge)
    #r'=R'.x
    r_forge = R_forge.x
    #e'=r'*u*v^(-1)mod n
    e_forge = (u * r_forge * config.inverse(v, n)) % n
    #s'=r'*v^(-1)mod n
    s_forge = (r_forge * config.inverse(v, n)) % n

    return e_forge, r_forge, s_forge
#据已有签名对伪造的签名进行验证
def Satoshi_verify(e_real, r_real, s_real, G, P):
    s_inverse = config.inverse(s_real, n)
    #生成 R'=(r',s')
    r_ver = ECG_k_point((e_real * s_inverse) % n, G)
    s_ver = ECG_k_point((r_real * s_inverse) % n, P)
    R_ver = ECG_ele_add(r_ver, s_ver)
```

```
        if R_ver.x % n == r_real:
            return True
        else:
            return False
```

针对 Schnorr 的攻击：

关于 Schnorr 的缺陷，主要有以下五项：

①k 的泄露也会导致私钥 d 的泄露；

②如果重复使用 k ，会导致私钥 d 的泄露；

③如果两个用户使用相同的 k ，会导致双方的私钥 d 的泄露；

④如果不验证 m，就可以实现签名的伪造；

⑤如果与 ECDSA 使用相同的私钥 d、k ，则会导致泄露私钥 d。

基于以上缺陷，可以做出如下攻击：

①针对缺陷一，只需把签名 s 的过程逆过来，就能求出 d；

②针对缺陷二，只需把两个求签名 s 的式子联立起来，就能求出 k，从而同① 一样求出 d；

③针对缺陷三，如果两个用户使用相同的 k ，就相当于 k 已经泄露，则与①相似，只需按照①的方法求解；

④针对缺陷四，可以用正确签名的随机 x 倍当作伪造的签名，因为不验证 m，只验证 e，所以是伪造可以成功；

⑤针对缺陷五，只需把签名 ECDSA 中 r 的式子和签名 Schnorr 中 r 的式子联立起来，就能求得私钥 d。

相关步骤代码如下：

定义 Schnorr 签名函数：

```
#指定 k 的 Schnorr 签名
def Schnorr_sign_with_certain_k(d, m, k):
    R = ECG_k_point(k, point_G)#R = kG
    e = int(hash_sha3_256(str(R) + str(m)), 2) #e = hash(R||M)
    s = (k + e * d) % n #s = (k + ed)mod n
    return R, s
```

Schnorr 签名的验证：

```
#使用公钥 P 进行 Schnorr 签名验证
def Schnorr_verify(e, R, s, P):
    #sG=(k+ed)G=kG+edG=R+eP
    sG = ECG_k_point(s, point_G)
    R_eP = ECG_ele_add(R, ECG_k_point(e, P)) #R + eP
    if R_eP.x == sG.x and R_eP.y == sG.y:
        return True
    else:
        return False
```

针对缺陷一的攻击：

```
print("\n1、泄露 k 会导致泄露私钥 d:")
            start_time = time.time()
            m = "GeMeng yun"
            k = random.randint(1, n - 1)
            print("泄露的 k 为", k)
            #Schnorr 签名
            R, s = Schnorr_sign_with_certain_k(d, m, k)
            #e = hash(R||M)
            e = int(hash_sha3_256(str(R) + str(m)), 2)
            d_break = ((s - k) * config.inverse(e, n)) % n
            if d_break == d:
                print("攻击成功，求出的私钥 d 为:", d_break)
                end_time = time.time()
                print("花费的时间为:", end_time - start_time, "s")
```

针对缺陷二的攻击：

```
print("\n2、重复使用 k 会导致泄露私钥 d:")
            start_time = time.time()
            m1 = "GeMeng yun"
            m2 = "Gemeng Yun"
            k = random.randint(1, n - 1)
            print("使用的相同的 k 为", k)
            #Schnorr 签名
            R1, s1 = Schnorr_sign_with_certain_k(d, m1, k)
            R2, s2 = Schnorr_sign_with_certain_k(d, m2, k)
            # e = hash(R||M)
            e1 = int(hash_sha3_256(str(R1) + str(m1)), 2)
            e2 = int(hash_sha3_256(str(R2) + str(m2)), 2)
            # k = (s2*e1-s1*e2)*(e1-e2)^(-1)
             k = ((s2 * e1 - s1 * e2) * config.inverse(e1 - e2, n)) %
n
```

```
            d_break = ((s1 - k) * config.inverse(e1, n)) % n
            if d_break == d:
                print("攻击成功，求出的私钥 d 为:", d_break)
                end_time = time.time()
                print("花费的时间为:", end_time - start_time, "s")
```

针对缺陷三的攻击：

```
print("\n3、两个用户使用相同的 k 会导致泄露双方的私钥 d:")
            start_time = time.time()
            m1 = "GeMeng yun"
            m2 = "Gemeng Yun"
            k = random.randint(1, n - 1)
            print("使用的相同的 k 为", k)
            d1 = random.randint(1, n - 1)
            d2 = random.randint(1, n - 1)
            #Schnorr 签名
            R1, s1 = Schnorr_sign_with_certain_k(d1, m1, k)
            R2, s2 = Schnorr_sign_with_certain_k(d2, m2, k)
            #求私钥 d
            #e = hash(R||M)
            e1 = int(hash_sha3_256(str(R1) + str(m1)), 2)
            e2 = int(hash_sha3_256(str(R2) + str(m2)), 2)
            d1_break = ((s1 - k) * config.inverse(e1, n)) % n
            d2_break = ((s2 - k) * config.inverse(e2, n)) % n
            if d2_break == d2:
                print("攻击成功，用户 1 破解用户 2 私钥 d2 为", d2_break)
            if d1_break == d1:
                print("攻击成功，用户 2 破解用户 1 私钥 d1 为", d1_break)
            end_time = time.time()
            print("花费的时间为:", end_time - start_time, "s")
```

针对缺陷四的攻击：

```
print("\n4、如果不验证 m，可以伪造签名:")
              start_time = time.time()
              #先生成正确的签名
              R, s = Schnorr_sign_with_certain_k(d, m, k)
              e = int(hash_sha3_256(str(R) + m), 2)#e = hash(R||M)
              print("正确的签名:\ne:", e, "\nR:", R, "\ns:", s)
              #伪造签名：将正确的签名变为原来的 x 倍
              x = random.randint(1, n - 1)
              s_forge = (x * s) % n
              R_forge = ECG_k_point(x * k, point_G)
              e_forge = (x * e) % n
              #验证伪造的签名
```

```
            if Schnorr_verify(e_forge, R_forge, s_forge, P):
                print("通过验证，伪造成功，伪造的签名:\ne':", e_forge, "\nR':", R_forge, "\ns':", s_forge)
            end_time = time.time()
            print("花费时间:", end_time - start_time, "s")
```

针对缺陷五的攻击：

```
print("\n5、与 ECDSA 使用相同的私钥 d 和 k 会导致泄露私钥 d:")
        start_time = time.time()
        # m1 用于 Schnorr 签名
        m1 = "GeMeng yun"
        # m2 用于 ECDSA 签名
        m2 = "gemeng Yun"
        k = random.randint(1, n - 1)
        print("使用的相同的 k 为", k)
        #Schnorr 签名
        R, S = Schnorr_sign_with_certain_k(d, m1, k)
        #ECDSA 签名
        r, s = ECDSA_sign_with_certain_k(d, m2, k)
        #Schnorr 的 e=hash(R||M)
        e_Schnorr = int(hash_sha3_256(str(R) + m1), 2)
        # ECDSA 的 e=hash(m)
        e_ECDSA = int(hash_sha3_256(m2), 2)

        # d = (s*S-e_ECDSA)*(s*e_Schnorr+r)^(-1)
        d_break = ((S * s - e_ECDSA) * config.inverse(e_Schnorr * s + r, n)) % n
        if d_break == d:
            print("攻击成功,求出的私钥 d 为:", d_break)
            end_time = time.time()
            print("花费的时间为:", end_time - start_time, "s")
        print("Schnorr 验证到此结束。")
```

针对关 SM2 签名的攻击：

关于 SM2 签名的缺陷，主要有以下四项：

①k 的泄露也会导致私钥 d 的泄露；

②如果重复使用 k ，会导致私钥 d 的泄露；

③如果两个用户使用相同的 k ，会导致双方的私钥 d 的泄露；

④如果与 ECDSA 使用相同的私钥 d、k ，则会导致泄露私钥 d。

基于以上缺陷，可以做出如下攻击：

①针对缺陷一，只需把签名 s 的过程逆过来，就能求出 d；

②针对缺陷二，只需把两个求签名 s 的式子联立起来，就能求出 k，从而同① 一样求出 d；

③针对缺陷三，如果两个用户使用相同的 k ，就相当于 k 已经泄露，则与①相似，只需按照①的方法求解；

④针对缺陷四，只需把签名 ECDSA 中 r 的式子和签名 SM2 中 r 的式子联立卡里，就能求得私钥 d。

相关步骤代码如下：

定义 SM2 签名函数：

```
#指定 k 的 SM2 签名
def SM2_sign_with_certain_k(M, ID, dA, PA, k):
    ZA = get_Z(ID, PA)
    #置 M1=ZA // M
    M1 = ZA + M
        e = bytes_to_int(bits_to_bytes(hash_function(M1))) # 计算 e = Hv(M1)
    #算椭圆曲线点(x1,y1)=[k]G
    X = ECG_k_point(k, point_G)
    x1 = bytes_to_int(ele_to_bytes(X.x))
    #算 r=(e+x1) mod n，由于 k 已经确定，若 r=0 或 r+k=n 则直接退出，而不是重新选择 k
    r = (e + x1) % n
    if r == 0 or r + k == n:
        return None
    #算 s = ((1 + dA)^(-1) ·(k-r·dA)) mod n，若 s=0 则直接退出
    s = (config.inverse(1 + dA, n) * (k - r * dA)) % n
    if s == 0:
      return None
    return r, s
```

SM 签名的验证：

```
#SM2 signature
        elif method == '3':
            print("正在验证 SM2 signature。")
            #生成公私钥对
            d = random.randint(1, n - 1)
            P = ECG_k_point(d, point_G)
```

```
print("私钥 d:", d, "\n 公钥 P:", P)
ID = "GeMeng yun@qq.com"
```

针对缺陷一的攻击：

```
print("\n1、泄露 k 会导致泄露私钥 d:")
start_time = time.time()
m = "GeMeng yun"
k = random.randint(1, n - 1)
print("泄露的 k 为", k)
#SM2 签名
r, s = SM2_sign_with_certain_k(m, ID, d, P, k)
#求私钥 d
d_break = ((k - s) * config.inverse(s + r, n)) % n
if d_break == d:
    print("攻击成功,求出的私钥 d 为:", d_break)
    end_time = time.time()
    print("花费的时间为:", end_time - start_time, "s")
```

针对缺陷二的攻击：

```
print("\n2、重复使用 k 会导致泄露私钥 d:")
start_time = time.time()
m1 = "GeMeng yun"
m2 = "gemeng Yun"
k = random.randint(1, n - 1)
print("使用的相同的 k 为", k)

ID_1 = "GeMeng yun@qq.com"
ID_2 = "gemeng Yun@qq.com"
#SM2 签名
r1, s1 = SM2_sign_with_certain_k(m1, ID_1, d, P, k)
r2, s2 = SM2_sign_with_certain_k(m2, ID_2, d, P, k)
#求私钥 d
d_break = (s1 - s2) * config.inverse((r2 - r1 + s2 - s1), n) % n
if d_break == d:
    print("攻击成功,求出的私钥 d 为:", d_break)
    end_time = time.time()
    print("花费的时间为:", end_time - start_time, "s")
```

针对缺陷三的攻击：

```
print("\n3、两个用户使用相同的 k 会导致泄露双方的私钥 d:")
start_time = time.time()
m1 = "GeMeng yun"
```

```
            m2 = "gemeng Yun"
            k = random.randint(1, n - 1)
            print("使用的相同的 k 为", k)
            d1 = random.randint(1, n - 1)  # 用户 1 的私钥
            d2 = random.randint(1, n - 1)  # 用户 2 的私钥
            ID_1 = "GeMeng yun@qq.com"
            ID_2 = "gemeng Yun@qq.com"
            #SM2 签名
            r1, s1 = SM2_sign_with_certain_k(m1, ID_1, d1, P, k)
            r2, s2 = SM2_sign_with_certain_k(m2, ID_2, d2, P, k)
            d1_break = ((k - s1) * config.inverse(s1 + r1, n)) % n
            d2_break = ((k - s2) * config.inverse(s2 + r2, n)) % n
            if d2_break == d2:
                print("攻击成功，用户 1 破解用户 2 私钥 d2 为", d2_break)
            if d1_break == d1:
                print("攻击成功，用户 2 破解用户 1 私钥 d1 为", d1_break)
            end_time = time.time()
            print("花费的时间为:", end_time - start_time, "s")
```

针对缺陷四的攻击：

```
    print("\n4、与 ECDSA 使用相同的私钥 d 和 k 会导致泄露私钥 d:")
            start_time = time.time()
            # m1 用于 ECDSA 签名
            m1 = "GeMeng yun"
            # m2 用于 SM2 签名
            m2 = "gemeng Yun"
            k = random.randint(1, n - 1)
            print("使用的相同的 k 为", k)
            ID1 = "GeMeng yun@qq.com"
            #ECDSA 签名
            r1, s1 = ECDSA_sign_with_certain_k(d, m1, k)
            #SM2 的签名
            r2, s2 = SM2_sign_with_certain_k(m2, ID1, d, P, k)
            #ECDSA 的 e=SM2 的 e=hash(m)
            e = int(hash_sha3_256(m1), 2)
            #求私钥 d：联立两个签名 s 的式子即可求出
             d_break = (s1 * s2 - e) * config.inverse((r1 - s1 * s2 -
    s1 * r2), n) % n
            if d_break == d:
                print("攻击成功，求出的私钥 d 为:", d_break)
                end_time = time.time()
                print("花费的时间为:", end_time - start_time, "s")
            print("SM2 signature 验证到此结束。")
        else:
```

```
            print("输入不正确，请重新输入。")
            continue
        if_ver = input("\n 是否要继续验证（0：退出；1：继续验证）：")
        if if_ver == '1':
            continue
        else:
            break
```

12.2 实现效果

根据提示，输入不同的标号，可以实现对不同签名算法的验证，运行结果如下：

ECDSA：

```
1、ECDSA
2、Schnorr
3、SM2 signature
请输入要验证的方法对应的标号(1,2或者3):1
正在验证ECDSA。
私钥d: 22446802884926703655846385774937284021478307234762236617522162088258938234259

1、泄露k会导致泄露私钥d:
泄露的k为 10230375374837646093380354360834834733434891837540718277297576528795399444581
攻击成功,求出的私钥d为: 22446802884926703655846385774937284021478307234762236617522162088258938234259
花费时间为: 0.2526111602783203 s
```

Schnorr：

```
是否要继续验证（0：退出；1：继续验证）：1
1、ECDSA
2、Schnorr
3、SM2 signature
请输入要验证的方法对应的标号(1,2或者3):3
正在验证SM2 signature。
私钥d: 67916285906209276498885161643706231510134884748330358156112047609516039782356
公钥P: (105433256946955291566353182139202097338041826121264064497236291171259548253274, 53868661543300543923190378212092178147881992813080494900467164448866985446378)

1、泄露k会导致泄露私钥d:
泄露的k为 37093288918174051904511336624253616017358724662263268872056946650340282918337
攻击成功,求出的私钥d为: 67916285906209276498885161643706231510134884748330358156112047609516039782356
花费的时间为: 0.30620503425598145 s

2、重复使用k会导致泄露私钥d:
使用的相同的k为 252919763122870944368925884721247391666208447542451902593164221397829217720 81
攻击成功,求出的私钥d为: 67916285906209276498885161643706231510134884748330358156112047609516039782356
花费的时间为: 0.5427207946777344 s

3、两个用户使用相同的k会导致泄露双方的私钥d:
使用的相同的k为 106381317386411309831671761022253192099764528331123273157074731893296952980814
攻击成功，用户1破解用户2私钥d2为 41000739173705534687921505698739788336833034095291326852315480348134156596817
攻击成功，用户2破解用户1私钥d1为 18242167863492252092720306665194962460327972768815871528659956543405546869025
花费的时间为: 0.5458328723907471 s

4、与ECDSA使用相同的私钥d和k会导致泄露私钥d:
使用的相同的k为 833597420530061559530866218719024677290742042488704170009322307580648125383 45
攻击成功，求出的私钥d为: 67916285906209276498885161643706231510134884748330358156112047609516039782356
花费的时间为: 0.49410319328308105 s
SM2 signature验证到此结束。
```

SM2：

```
是否要继续验证（0：退出；1：继续验证）：1
1、ECDSA
2、Schnorr
3、SM2 signature
请输入要验证的方法对应的标号(1,2或者3):2
正在验证Schnorr!
私钥d: 216408132434989011270445138154426393512545739161391136368586827468241608453 92
公钥P: (100331589172264865853076837532668015162494998791868192629157582800198222018030, 43936039915935816909518207283585839594659292000599483467877541094648411325325)

1、泄露k会导致泄露私钥d:
泄露的k为 10511759174216952904842560411837561662397354832551948427048331253015559297434
攻击成功，求出的私钥d为: 216408132434989011270445138154426393512545739161391136368586827468241608453 92
花费的时间为: 0.3123300075531006 s

2、重复使用k会导致泄露私钥d:
使用的相同的k为 53927116548629878124612696630796132888270868602859637091865133995568865842 24
攻击成功，求出的私钥d为: 216408132434989011270445138154426393512545739161391136368586827468241608453 92
花费的时间为: 0.5005474090576172 s

3、两个用户使用相同的k会导致泄露双方的私钥d:
使用的相同的k为 109644652375890834355321418451896439356237428288229948627132023973303027057 60
攻击成功，用户1破解用户2私钥d2为 99036419803373970366128627591928362116927494360744738270798295911202070031625
攻击成功，用户2破解用户1私钥d1为 17679266659447967034049499528990302898131373988188661315566346255193976004434
花费的时间为: 0.5058047771453857 s

4、如果不验证m，可以伪造签名:
正确的签名:
e: 203642193008570897694989354945067017636469280942927232300020750543929343411 32
R: (114296468863752281758774569284947687461033891488334206676540063916847414351739, 913509342133545819624674031341679452405652198879433441593160232144516283025 0)
s: 76732842709836178496135312176393290460485901350047448413583600654144927604366
通过验证，伪造成功，伪造的签名:
e': 46670022219995917193345633053703011216513266948808989586920470832349859440071
R': (48153879603639523464522369939806998614048176949843781430598189680756808454384, 79388780954804492011929869166608644374955129062203694209638855950792033345716)
s': 69886352784171111495260304111434956740182097353105292892580282345621532155763
花费时间: 1.3647899627685547 s

5、与ECDSA使用相同的私钥d和k会导致泄露私钥d:
使用的相同的k为 11188643614114225248938984738288739946027038800601370509438564847218960827 1676
攻击成功,求出的私钥d为: 216408132434989011270445138154426393512545739161391136368586827468241608453 92
花费的时间为: 0.5472862720489502 s
Schnorr验证到此结束。
```

13.Project13:Implement the above ECMH scheme

未做

14.Project14:Implement a PGP scheme with SM2

未做

15.Project15:implement sm2 2P sign with real network communication

15.1 实现方式

本 project 要求实现在真实网络环境下的双方 SM2 签名。

基本原理：

首先要实现 SM2 的签名和验证。

SM2 签名的主要步骤：

设待签名的消息为M，为了获取消息M的数字签名(r,s)，作为签名者的用户A应实现以下运算步骤：

A1：置$\overline{M}=Z_A \| M$；

A2：计算$e=H_v(\overline{M})$，按本文本第1部分4.2.3和4.2.2给出的细节将e的数据类型转换为整数；

A3：用随机数发生器产生随机数$k \in [1,n-1]$；

A4：计算椭圆曲线点$(x_1,y_1)=[k]G$，按本文本第1部分4.2.7给出的细节将x_1的数据类型转换为整数；

A5：计算$r=(e+x_1) \bmod n$，若$r=0$或$r+k=n$则返回A3；

A6：计算$s=((1+d_A)^{-1} \cdot (k-r \cdot d_A)) \bmod n$，若$s=0$则返回A3；

A7：按本文本第1部分4.2.1给出的细节将r、s的数据类型转换为字节串，消息M的签名为(r,s)。

SM2 签名验证的主要步骤：

为了检验收到的消息M'及其数字签名(r', s')，作为验证者的用户B应实现以下运算步骤：

B1：检验$r' \in [1,n-1]$是否成立，若不成立则验证不通过；

B2：检验$s' \in [1,n-1]$是否成立，若不成立则验证不通过；

B3：置$\overline{M'}=Z_A \| M'$；

B4：计算$e' = H_v(\overline{M'})$，按本文本第1部分4.2.3和4.2.2给出的细节将e'的数据类型转换为整数；

B5：按本文本第1部分4.2.2给出的细节将r'、s'的数据类型转换为整数，计算$t = (r' + s') \bmod n$，若$t = 0$，则验证不通过；

B6：计算椭圆曲线点$(x'_1, y'_1)=[s']G + [t]P_A$；

B7：按本文本第1部分4.2.7给出的细节将x'_1的数据类型转换为整数，计算$R = (e' + x'_1) \bmod n$，检验$R=r'$是否成立，若成立则验证通过；否则验证不通过。

相关步骤代码如下：

SM2 签名：

```
#SM2 生成签名
def SM2_sig(ID, dA, PA):
    ZA = get_Z(ID, PA)
    #置 M1=ZA // M
    M1 = ZA + M
    #算 e = Hv(M1)，并将 e 的数据类型转换为整数
    e = bytes_to_int(bits_to_bytes(hash_function(M1)))
    while True:
        #产生随机数 k ∈[1,n-1]
        k = randint(1, n - 1)
        #算椭圆曲线点(x1,y1)=[k]G
        X = ECG_k_point(k, point_g)
        #把 x1 的数据类型转换为整数
        x1 = bytes_to_int(ele_to_bytes(X.x))
        #算 r=(e+x1) mod n，如果 r=0 或者 r+k=n，重新选择 k
        r = (e + x1) % n
        if r == 0 or r + k == n:
            continue
        #算 s = ((1 + dA)-1 ·(k-r·dA)) mod n，如果 s=0，重新选择 k
        s = (config.inverse(1 + dA, n) * (k - r * dA)) % n
        if s == 0:
            continue
        break
    #消息 M 的签名为(r,s)
    return r, s
```

SM2 签名验证：

```
#SM2 签名验证
def SM2_ver(IDA, r, s, PA):
    #验证 r,s∈[1,n-1]是否成立，若不成立，验证不通过
    if r < 0 or r > n:
        print("r 验证不通过！")
```

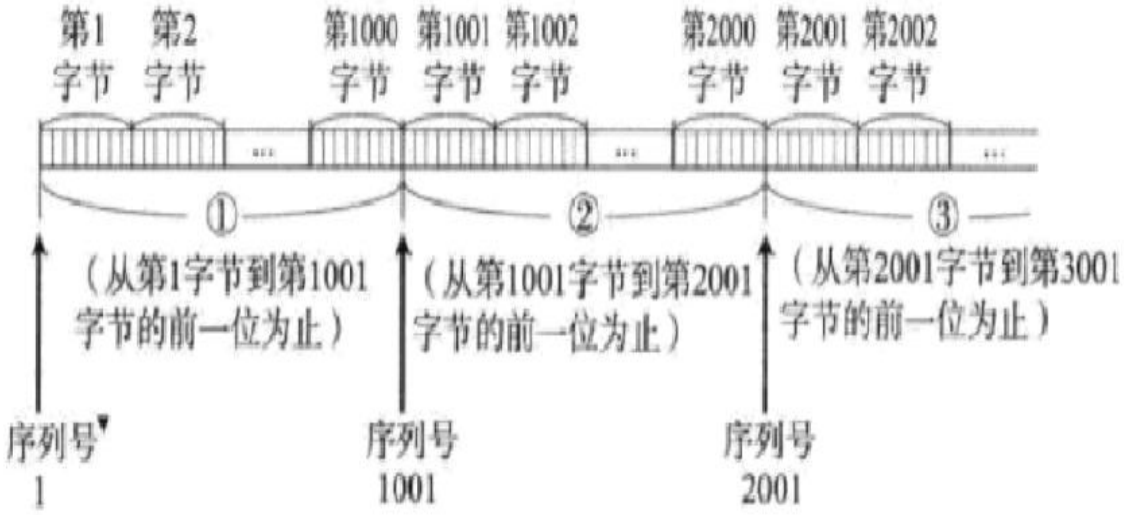
```
            return -1
        if s < 0 or s > n:
            print("s 验证不通过！")
            return -1
        ZA = get_Z(IDA, PA)
        #置 M1=ZA // M
        M1 = ZA + M
        #算 e = Hv(M1)，并将 e 的数据类型转换为整数
        e = bytes_to_int(bits_to_bytes(hash_function(M1)))
        #算 t=(r + s)mod n，若 t = 0，则验证不通过
        t = (r + s) % n
        if t == 0:
            return -1
        #算椭圆曲线点(x1,y1)=[s']G+[t]PA
        X = ECG_ele_add(ECG_k_point(s, point_g), ECG_k_point(t, PA))
        #把 x1 数据类型转换为整数，算 R=(e+x1)mod n,检验 R=r 是否成立
        x1 = bytes_to_int(ele_to_bytes(X.x))
        R = (e + x1) % n
        if R == r:
            return 1
        else:
            return -1
```

要想实现在真实网络环境下的双方 SM2 签名，可以用 TCP 协议模拟真实网络环境。

TCP 原理：TCP 对数据传输提供的管控机制，主要体现在两个方面：安全和效率。这些机和线程的设计原则是：保证数据安全的前提下，尽可能提高传输效率。

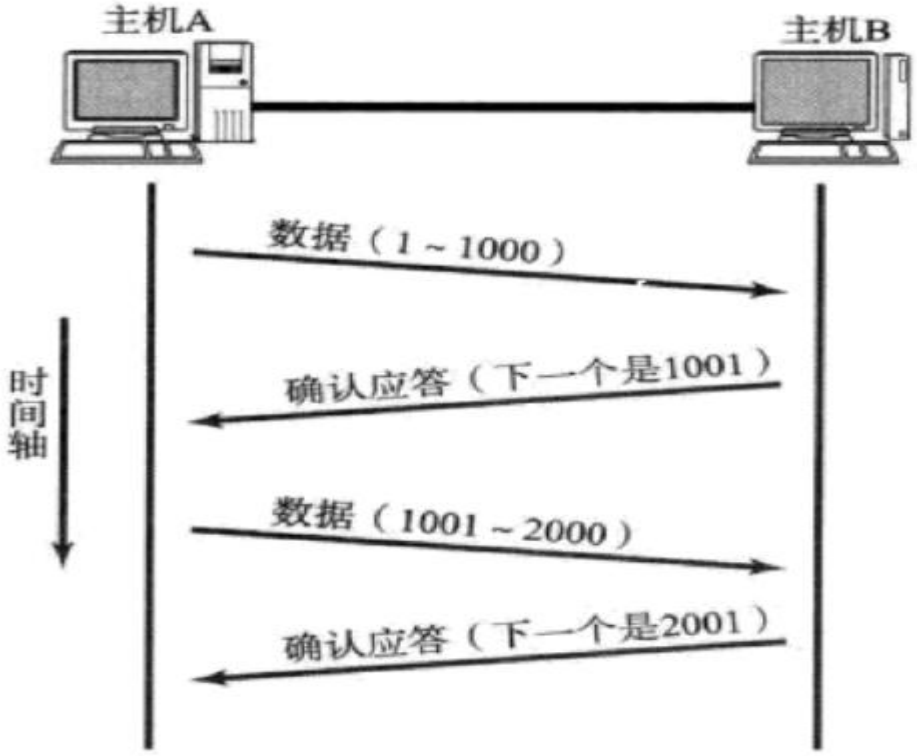
1. 确认应答机制（可靠机制）

发送方发送给接收方数据之后，接收方就会回应一个应答报文。如果发送方收到这个应答报文，那么就认为接收方已经收到了数据。由于网络上的传输，充满不确定性，因此不能通过收到数据的顺序来确定逻辑。因此就需要给应答进行编号。TCP 的序号和确认序号，是以字节为单位进行编号的。



针对每个字节进行编号，依次进行累加。（TCP 序号的起始不一定是从 1 开始的）。

工作过程：

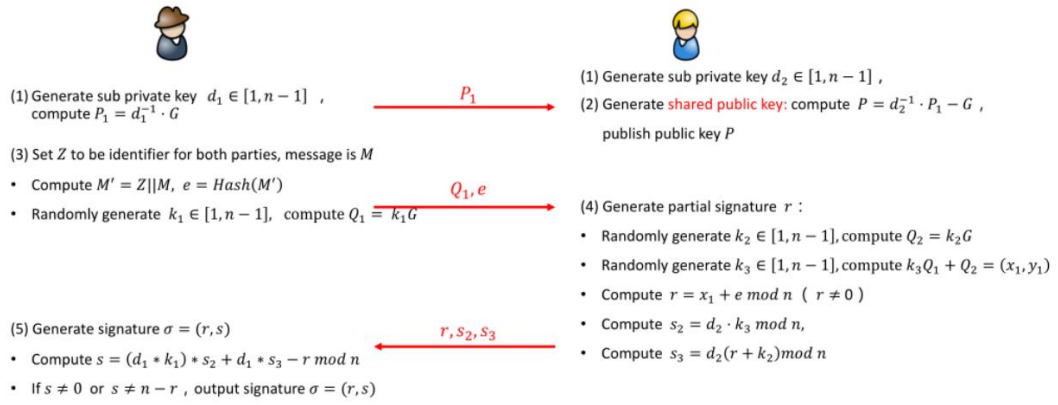


解释：第一个请求,A 给 B 发送了 1000 个字节的数据，序号就是 1~1000（假设从 1 开始编号），这就相当于是发送了一个数据报，这个数据报的序号是 1，长度为 1000；确认应答数据报，里面的确认序号是 1001，意思就是 1001 之前的数

据，B 已经收到了。也可以理解成，B 在向 A 索要 1001 开始的数据。序号 1001~2000 的数据传输和应答原理与上面的一致。

TCP 的核心是可靠性，可靠性的核心是确认应答。

真实网络环境下的双方 SM2 签名的流程图：



即服务器端先生成 d1，计算 P1 ，发送给客户端，同时从客户端接受公钥 P；接着客户端生成 d2，从服务器接受 P1，同时计算公钥 P 发送给服务器；服务器端收到公钥 P 后，就可以进行签名，按照流程计算出 Q1 和 e 发送给客户端；客户端收到 Q1 和 e 后，就可以生成部分签名 r，按照流程将 r,s2,s3 发送给服务器；服务器接收到 r,s2,s3 后，就可以计算剩余的签名 s；最后我们可以对生成的签名进行验证，使用 SM2_signature_verification.py 中的代码即可。

相关步骤代码如下：

服务器端先生成 d1，计算 P1 ，发送给客户端，同时从客户端接受公钥 P：

```
d1 = randint(1, n - 1)#服务器子私钥 d1
P1 = ECG_k_point(config.inverse(d1, n), point_G)
print("\n 服务器子私钥 d1:", d1, "\nP:", P1)
#发送 P1 至客户端
new_socket.sendall(str([P1.x, P1.y]).encode())
#从客户端接受公钥 P，用来签名、验证
p1, p2 = eval(new_socket.recv(1024).decode())
PA = Point(int(p1), int(p2))
```

客户端生成 d2，从服务器接受 P1，同时计算公钥 P 发送给服务器；

```
d2 = randint(1, n - 1)#客户端子私钥 d2
```

```
    print("\n 客户端子私钥 d2 为:", d2)
    x, y = eval(s.recv(1024).decode())#从服务器接收 P1
    P1 = Point(int(x), int(y))
    #算公钥 P
    P = ECG_ele_add(ECG_k_point(config.inverse(d2, n), P1), ECG_k_point(
q - 1, point_G))
    print("计算的公钥 P:", P)
    #发送公钥 P 至服务器，用来签名、验证
    s.sendall(str([P.x, P.y]).encode())
```

服务器端收到公钥 P 后，就可以进行签名，按照流程计算出 Q1 和 e 发送给客户端：

```
    #签名开始
    ZA = get_Z(ID, PA)
    M1 = ZA + M #令 M1=ZA // M
    #计算 e = Hv(M1)，进行 e 的数据类型转换
    e = bytes_to_int(bits_to_bytes(hash_function(M1)))
    k1 = randint(1, n - 1)
    #计算椭圆曲线点(x1,y1)=[k]G
    Q1 = ECG_k_point(k1, point_G)
    #发送 e 和 Q1 至客户端
    new_socket.sendall(str(e).encode())
    new_socket.sendall(str([Q1.x, Q1.y]).encode())
```

客户端收到 Q1 和 e 后，就可以生成部分签名 r，按照流程将 r,s2,s3 发送给服务器：

```
    #从服务器接收 e 和 Q1
    e = int(s.recv(1024).decode())
    x, y = eval(s.recv(1024).decode())
    Q1 = Point(int(x), int(y))

    k2 = randint(1, n - 1)#随机生成 k2 和 k3
    k3 = randint(1, n - 1)
    Q2 = ECG_k_point(k2, point_G)
    X = ECG_ele_add(ECG_k_point(k3, Q1), Q2)
    #算 r=(e+x1) mod n，若 r=0，则签名失败
    r = (X.x + e) % n
    if r == 0:
        print("签名失败！r==0！")
        s.close()
    #据 ppr 计算 s2、s3
    s2 = (d2 * k3) % n
```

```
    s3 = (d2 * (r + k2)) % n

    #发送 r,s2,s3 至服务器
    s.sendall(str(r).encode())
    s.sendall(str(s2).encode())
    s.sendall(str(s3).encode())
```

服务器接收到 r, s2, s3 后，就可以计算剩余的签名 s：

```
    #从客户端接受 r,s2,s3
    r = int(new_socket.recv(1024).decode())
    s2 = int(new_socket.recv(1024).decode())
    s3 = int(new_socket.recv(1024).decode())
    #算 s = ((1 + dA)-1 ·(k-r·dA)) mod n，若 s=0，则重新选择 k
    s = ((d1 * k1) * s2 + d1 * s3 - r) % n
    if s == 0 or s == n - r:
        print("签名失败！s==0 或者 s==n-r!")
        S.close()
    end_time = time.time()
    print("\n 签名完成:\nr:", r, "\ns:", s)
    print("双方签名 SM2 所需时间:", end_time - start_time, "s")
```

对生成的签名进行验证：

```
    #验证
    start_time = time.time()
    if SM2_signature_verification.SM2_ver(M, ID, r, s, PA, n):
        print("\n 签名验证成功！")
        end_time = time.time()
        print("验证 SM2 所用时间:", end_time - start_time, "s")
```

15. 2 实现效果

先运行服务器端代码，再运行客户端代码。

服务器端运行结果如下：

```
服务器端启动     主机名: LAPTOP-4000RE59          端口号: 1234
主机地址 192.168.0.110 ,端口号 65526 的客户端接入本服务器，开始执行在线双方SM2签
名

服务器子私钥d1: 863345646140696264989257576077961751526985018930325493535059187 4
2062127025980
P: (992971863683102344029239707459176135650289652132893347534469277015438027314 5
7, 298007825757010431584493888522534413520927293745156271894671718497220381784 5
8
)

签名完成:
r: 202337210498568346410074791424545288317472371438290117939072760145270566061 23

s: 270133080072774939866928173245316982927956396916135864468574440887897665060 15
双方签名SM2所需时间: 1.3917129039764404 s

签名验证通过!
验证SM2所需时间: 0.45246124267578125 s
签名完成，断开连接!
```

客户端运行结果如下：

```
连接到主机名为 LAPTOP-4000RE59 端口号为 1234 的服务器，开始执行在线双方SM2签名

客户端子私钥d2: 311910286185983404750288049346999299569097491944011519855861508 1
6177403620737
计算的公钥P: (164880038167383346320269503664944055797141410519624683370266826560
09382871290, 467989564484519326174834334830444224719686845909116773580220139222 8
137029640)
签名完成，断开连接!
```

16. Project16: implement sm2 2P decrypt with real network communication

16.1 实现方式

本 project 要求实现真实网络环境下的双方 SM2 解密，首先要实现 SM2 的解密。

基本原理：

SM2 加密流程图：

设需要发送的消息为比特串M，$klen$为M的比特长度。

为了对明文M进行加密，作为加密者的用户A应实现以下运算步骤：

A1：用随机数发生器产生随机数k∈[1,n-1]；

A2：计算椭圆曲线点C_1=[k]G=(x_1,y_1)，按本文本第1部分4.2.8和4.2.4给出的细节，将C_1的数据类型转换为比特串；

A3：计算椭圆曲线点S=[h]P_B，若S是无穷远点，则报错并退出；

A4：计算椭圆曲线点[k]P_B=(x_2,y_2)，按本文本第1部分4.2.5和4.2.4给出的细节，将坐标x_2、y_2的数据类型转换为比特串；

A5：计算t=$KDF(x_2 \| y_2, klen)$，若t为全0比特串，则返回A1；

A6：计算$C_2 = M \oplus t$；

A7：计算$C_3 = Hash(x_2 \| M \| y_2)$；

A8：输出密文$C = C_1 \| C_2 \| C_3$。

SM2 解密流程图：

设$klen$为密文中C_2的比特长度。

为了对密文C=$C_1 \| C_2 \| C_3$进行解密，作为解密者的用户B应实现以下运算步骤：

B1：从C中取出比特串C_1，按本文本第1部分4.2.3和4.2.9给出的细节，将C_1的数据类型转换为椭圆曲线上的点，验证C_1是否满足椭圆曲线方程，若不满足则报错并退出；

B2：计算椭圆曲线点S=[h]C_1，若S是无穷远点，则报错并退出；

B3：计算[d_B]C_1=(x_2,y_2)，按本文本第1部分4.2.5和4.2.4给出的细节，将坐标x_2、y_2的数据类型转换为比特串；

B4：计算t=$KDF(x_2 \| y_2, klen)$，若t为全0比特串，则报错并退出；

B5：从C中取出比特串C_2，计算$M' = C_2 \oplus t$；

B6：计算$u = Hash(x_2 \| M' \| y_2)$，从$C$中取出比特串$C_3$，若$u \neq C_3$，则报错并退出；

B7：输出明文M'。

相关步骤代码如下：

SM2 加密：

```
#SM2 加密
def SM2_enc(M, PB):
    M = remove_0b_at_beginning(M)
    point_G = Point(config.get_Gx(), config.get_Gy())
    klen = len(M)
    while True:
        #产生随机数 k∈[1, n - 1]
        k = randint(1, n - 1)
        #算椭圆曲线点 C1=[k]G=(x1,y1)
        C1 = ECG_k_point(k, point_G)
        #把 C1 的数据类型转换为比特串
        C1 = bytes_to_bits(point_to_bytes(C1))
        C1 = remove_0b_at_beginning(C1)
        #算椭圆曲线点 S=[h]PB，若 S 是无穷远点，则报错并退出
        S = ECG_k_point(h, PB)
        if S == ECG_ele_zero():
```

```
                print("S 是无穷远点！")
                return -1
            #算椭圆曲线点[k]PB=(x2,y2)
            x2 = ECG_k_point(k, PB).x
            y2 = ECG_k_point(k, PB).y
            #把坐标 x2、y2 的数据类型转换为比特串
            x2 = bytes_to_bits(ele_to_bytes(x2))
            y2 = bytes_to_bits(ele_to_bytes(y2))
            x2 = remove_0b_at_beginning(x2)
            y2 = remove_0b_at_beginning(y2)
            #算 t = KDF(x2∥y2, klen)，若 t 为全 0 比特串，则重新计算，直到不全为零再跳出
            t = KDF(x2 + y2, klen)
            if is_zero_str(t):
                continue
            break
        #算 C2 = M ⊕ t
        C2 = bin(int(M, 2) ^ int(t, 2))
        C2 = remove_0b_at_beginning(C2)
        C2 = padding_0_to_length(C2, klen)
        #算 C3 = Hash(x2 // M // y2)
        C3 = hash_function(x2 + M + y2)
        C3 = remove_0b_at_beginning(C3)
        #输出密文 C = C1 // C2 // C3
        return C1 + C2 + C3
```

SM 解密：

```
#SM2 解密
def sm2_dec(C, sk):
    c1_len = (2 * math.ceil(math.log2(q) / 8) + 1) * 8
    #C 中取出比特串 C1
    C1 = '0b' + C[:c1_len]
    C2 = C[c1_len: len(C) - 256]
    C3 = C[len(C) - 256: len(C)]
    #把 C1 的数据类型转换为椭圆曲线上的点
    C1 = bytes_to_point(a, b, bits_to_bytes(C1))
    #算椭圆曲线点 S=[h]C1，若 S 是无穷远点，则报错并退出
    S = ECG_k_point(h, C1)
    if S == ECG_ele_zero():
        print("解密失败：S 是无穷远点！")
        return -1
    #算[sk]C1=(x2,y2)
    X = ECG_k_point(sk, C1)
    #把坐标 x2、y2 的数据类型转换为比特串
```

```
        x2 = bytes_to_bits(ele_to_bytes(X.x))
        y2 = bytes_to_bits(ele_to_bytes(X.y))
        x2 = remove_0b_at_beginning(x2)
        y2 = remove_0b_at_beginning(y2)
        #算 t=KDF(x2 // y2, klen)，若 t 为全 0 比特串，则报错并退出
        klen = len(C2)
        t = KDF(x2 + y2, klen)
        if is_zero_str(t):
            print("解密失败：t 是全零比特串！")
            return -1
        #从 C 中取出比特串 C2，计算 M' = C2 ⊕ t
        M_1 = bin(int(C2, 2) ^ int(t, 2))
        M_1 = remove_0b_at_beginning(M_1)
        #算 u = Hash(x2 // M' // y2),从 C 中取出比特串 C3,若 u!=C3,报错并退出
        u = hash_function(x2 + M_1 + y2).replace('0b', '')
        if u != C3:
            print("解密失败，u! =C3")
            return -1
        #输出明文 M'
        return M_1
```

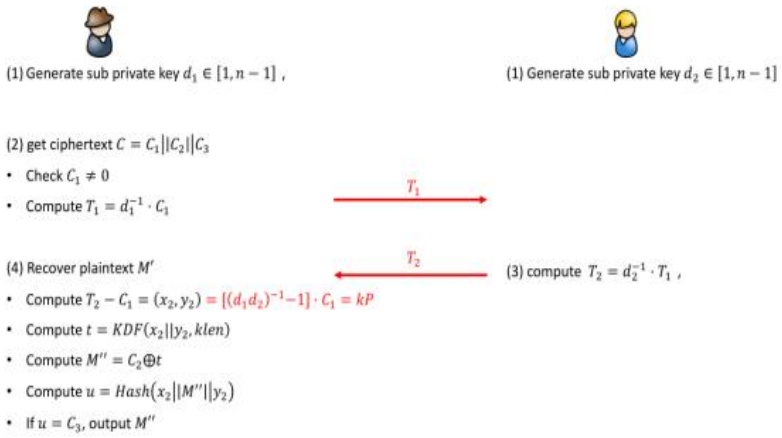
加解密的具体实现：

```
if __name__ == '__main__':
    #明文
    M = bin(int("aaacccbbbdd", 16))[2:]

    #密钥对生成
    config.set_default_config()
    parameters = config.get_parameters()
    key = key_pair_generation(parameters)
    dB = key[0]
    PB = key[1]
    print("公钥:\n", PB)
    print("私钥:\n", dB)
    #初始化
    a = config.get_a()
    b = config.get_b()
    q = config.get_q()
    h = config.get_h()
    n = config.get_n()
    #加密
    start_time = time.time()
    C = SM2_enc(M, PB)
    end_time = time.time()
```

```
        print("\n 加密密文:\n", hex(int(C, 2)))
        print("加密时间:", end_time - start_time, "s")
        #解密
        start_time = time.time()
        M_dec = sm2_dec(C, dB)
        end_time = time.time()
        print("\n 解密结果:", hex(int(M_dec, 2))[2:])
        print("原始明文:", hex(int(M, 2))[2:])
        #验证解密是否正确
        if M_dec == M:
            print("解密正确!\t 解密时间:", end_time - start_time, "s")
        else:
            print("解密错误。")
```

在此基础上，要实现真实网络环境下的双方 SM2 解密，可以使用 TCP 协议模拟真实的网络环境。TCP 的原理已经在上一个 project 中给出。

双方解密 SM2 的流程图如下：



加密环节中，公钥 $P=[(d_1d_2)^{-1}-1]G$，私钥 $d=(d_1d_2)^{-1}-1$. 即两方要在[1，n-1]内选择一个子密钥，分别记作 d_1、d_2，然后各自执行命令，过程中要进行两次消息的交换。其具体流程为：从客户端获取 d2；在服务器端进行 SM2 加密，使用代码 SM2_encryption_decryption.py 即可；获取密文 C 后，进行双方解密即可。

相关步骤代码如下：

从客户端获取 d2：

```
#P2 子密钥 d2
d2 = randint(1, n - 1)
print("P2 的子密钥 d2:", d2)
#发送 d2 至服务器
```

```
    s.sendall(str(d2).encode())
```

在服务器端进行 SM2 加密：

```
    M = bin(int("aaacccbbbdd", 16))[2:]
    d1 = randint(1, n - 1)#P1 子密钥 d1
    print("P1 的子密钥 d1:", d1)
    #从客户端接受 d2
    d2 = int(new_socket.recv(1024).decode())
    dB = config.inverse(d1 * d2, n) - 1#私钥
    PB = ECG_k_point(dB, point_G)#公钥
    C = SM2_encryption_decryption.SM2_enc(M, PB, n, h)
```

进行双方解密：

服务器端代码：

```
    T1 = config.inverse(d1, n) * C1#求 T1
    #发送 T1 至客户端
    new_socket.sendall(str(T1).encode())
    #从客户端接受 T2
    T2 = int(new_socket.recv(1024).decode())
    C1 = "0b" + C[:c1_len]
    C1 = bytes_to_point(a, b, bits_to_bytes(C1))
    #X = [sk]C1=(x2,y2)
    X = ECG_k_point(dB, C1)
    #将坐标 x2、y2 的数据类型转换为比特串
    x2 = bytes_to_bits(ele_to_bytes(X.x))
    y2 = bytes_to_bits(ele_to_bytes(X.y))
    x2 = remove_0b_at_beginning(x2)
    y2 = remove_0b_at_beginning(y2)
    #计算 t=KDF(x2 // y2, klen)，若 t 为全 0 比特串，就报错退出
    klen = len(C2)
    t = KDF(x2 + y2, klen)
    if SM2_encryption_decryption.is_zero_str(t):
        print("解密失败：t 是全零比特串！")
        S.close()
    #从 C 中取出比特串 C2，计算 M' = C2 ⊕ t
    M_1 = bin(int(C2, 2) ^ int(t, 2))
    M_1 = remove_0b_at_beginning(M_1)
    #算 u = Hash(x2 // M' // y2)，从 C 中取出比特串 C3，若 u==C3，则解密成功
    u = hash_function(x2 + M_1 + y2)
    u = remove_0b_at_beginning(u)
    if u == C3:
        print("解密成功！二进制明文:", bin(int(M_1, 2))[2:],
              "\n\t\t 十六进制明文:", hex(int(M_1, 2))[2:])
```

```
        new_socket.sendall(M_1.encode())
        end_time = time.time()
        print("双方解密 SM2 所用时间:", end_time - start_time, "s")
        print("解密完成，断开连接。")
        S.close()
```

客户端代码：

```
    #从服务器接受 T1
    T1 = int(s.recv(1024).decode())
    #计算 T2
    T2 = config.inverse(d2, n) * T1
    #发送 T2 至服务器端
    s.sendall(str(T2).encode())
    print("解密完成，断开连接。")
    s.close()
```

16.2 实现效果

运行时，先运行服务器端代码，再运行客户端代码。

服务器端运行结果如下：

```
服务器端启动    主机名: LAPTOP-4O00RE59          端口号: 1113
主机地址 192.168.0.110 ,端口号 65420 的客户端接入本服务器，开始执行在线双方SM2解
密
P1的子密钥d1: 491305634851620269912444394069511533533335431109575193165611022375
61547987407
解密成功！二进制明文为: 10101010101011001100110010111011101111011101
                    十六进制明文为: aaacccbbbdd
双方解密SM2所需时间: 1.1460130214691162 s
解密完成，断开连接！
```

客户端代码如下：

```
连接到主机名为 LAPTOP-4O00RE59 端口号为 1113 的服务器，开始执行在线双方SM2解密
P2的子密钥d2: 27833951423122856251556524171756566252527325222427898082517348438729620600458
解密完成，断开连接！
```

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

未做

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

未做

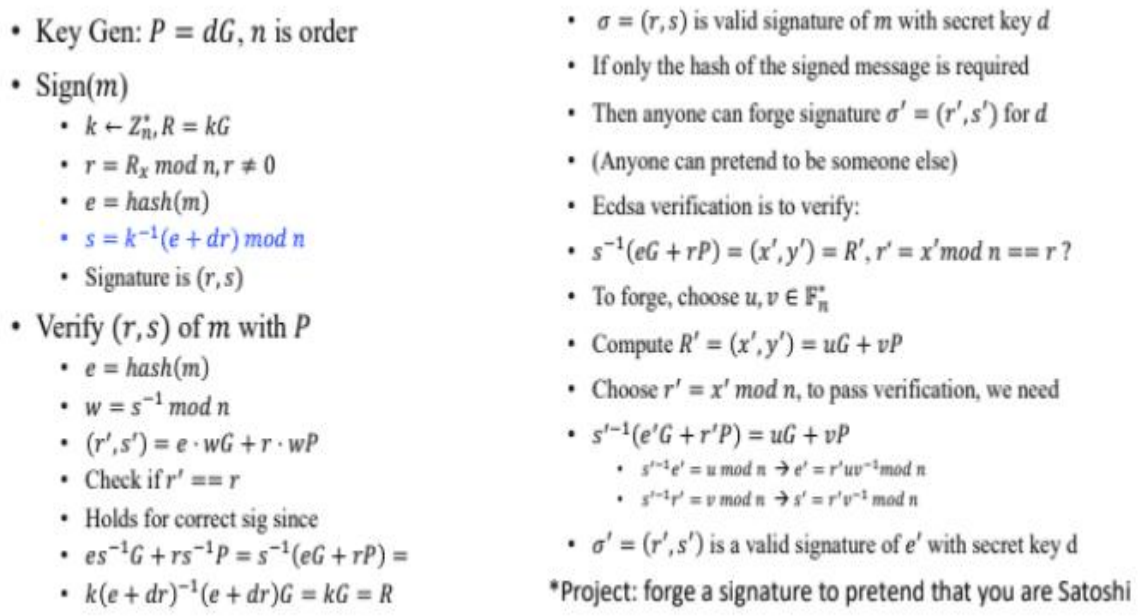
19. Project19:forge a signature to pretend that you are Satoshi

19.1 实现方式

本 project 要实现用正确的 Satoshi 签名伪造 Satoshi 签名。

首先，需要生成正确的 Satoshi 签名，然后进行 Satoshi 签名的伪造，最后是通过正确的 Satoshi 签名对伪造的签名进行验证。

其流程图如下：



相关步骤代码如下：

生成正确的 Satoshi 签名：

```
1.  def Satoshi_sign(d, m):#生成正确的 Satoshi 签名
```

```
        k = randint(1, n - 1)
        kG = ECG_k_point(k, point_G)
        #求 e,r,s
        r = ele_to_int(kG.x)
        e = int(hash_sha3_256(m), 2)
        s = ((config.inverse(k, n)) * (e + d * r)) % n
        return e, r, s
```

Satoshi 签名的伪造：

```
    #伪造 Satoshi 签名
    def pretend_Satoshi_sig(G, P):
        #随机选择 u、v
        u = randint(1, n - 1)
        v = randint(1, n - 1)
        #计算伪造签名 R'值
        x_forge = ECG_k_point(u, G)
        y_forge = ECG_k_point(v, P)
        R_forge = ECG_ele_add(x_forge, y_forge)
        #r'=R'.x
        r_forge = R_forge.x
        #e'=r'*u*v^(-1)mod n
        e_forge = (u * r_forge * config.inverse(v, n)) % n
        #s'=r'*v^(-1)mod n
        s_forge = (r_forge * config.inverse(v, n)) % n

        return e_forge, r_forge, s_forge
```

根据已有的签名对伪造的签名进行验证：

```
    #据已有签名对伪造的签名进行验证
    def Satoshi_verify(e_real, r_real, s_real, G, P):
        s_inverse = config.inverse(s_real, n)
        #生成 R'=(r',s')
        r_ver = ECG_k_point((e_real * s_inverse) % n, G)
        s_ver = ECG_k_point((r_real * s_inverse) % n, P)
        R_ver = ECG_ele_add(r_ver, s_ver)
        if R_ver.x % n == r_real:
            return True
        else:
            return False
```

具体实现：

```
    if __name__ == '__main__':
        #初始化
        config.set_default_config()
```

```
    parameters = config.get_parameters()
    point_G = Point(config.get_Gx(), config.get_Gy())
    n = config.get_n()
    m = "GeMeng yun"
    #生成公私钥对
    sk = random.randint(1, n - 1)
    pk = ECG_k_point(sk, point_G)
    print("公钥为:", pk)
    print("私钥为:", sk)
    #生成正确的 Satoshi 签名
    start_time = time.time()
    e_real, r_real, s_real = Satoshi_sign(sk, m)
    end_time = time.time()
                                    print("\n 签 名 成功:\ne:", e_real, "\nr:", r_real, "\ns:", s_real)
    print("签名所用时间为:", end_time - start_time, "s")
    #伪造新签名
    start_time = time.time()
    e_forge, r_forge, s_forge = pretend_Satoshi_sig(point_G, pk)
    end_time = time.time()
                            print("\n 伪 造 签 名 成功:\ne':", e_forge, "\nr':", r_forge, "\ns':", s_forge)
    print("伪造签名所用时间为:", end_time - start_time, "s")
    #使用已有的签名对伪造的签名验证
    start_time = time.time()
    if Satoshi_verify(e_real, r_real, s_real, point_G, pk):
        end_time = time.time()
             print("\n 验证伪造的签名成功，\n 验证签名所用时间为:", end_time - start_time, "s")
```

19.2 实现效果

运行代码，可得到如下结果：

```
py
公钥为：(84933630281381821642214136863983984204592553891812468680878637103990489
972115, 3052133804957769366014243170594892917372959144925978075903753645258818877
70788)
私钥为：8179940841989724741135806065759398578186953438400659074427958319016757781
17122

签名成功：
e：7441626144355438208132363852404896802506998168064708204937282999110494105350 2

r：7547092209738596180982877980073951456865400396375004534638398612091759798832 6

s：2208971162815361131382395313510059648425669107132611063781881518630956999309 7
签名所用时间为：0.21743106842041016 s

伪造签名成功：
e'：5578183249061292380697780311138194175004067123131536541768403585205108750790
8
r'：9825241768577522656098386995599284746167691356999724508790373581786068900322

s'：1108409620676333720496993555841575026486346452432186787868187359711973661471
6
伪造签名所用时间为：0.42479586601257324 s

验证伪造的签名成功，
验证签名所用时间为：0.4578540325164795 s
```

20. Project20:ECMH PoC

未做

21. Project21:Schnorr Bacth

未做

22. Project22:research report on MPT

22.1 实现方式

本 project 要求理解、研究已有的 MPT 项目代码。

理解以及研究结果以注释的形式穿插在代码中。

相关代码及注释如下：

```
import hashlib
#定义 branch 分支结点类
class Branch_node:
    def __init__(self):
        #十六进制编码的 16 种字符
        #children 字典中的 value 作为 branch 表的终止符
        #同时也记录当前 extension 结点的状态，
        #未更新 hash 值时，默认为 False
        self.type = 'branch'
```

```
            self.children = {'0': None, '1': None, '2': None, '3': None,
'4': None,
                             '5': None, '6': None, '7': None, '8': None,
'9': None,
                             'a': None, 'b': None, 'c': None, 'd': None,
'e': None,
                            'f': None,'value': False}
#定义 extension 拓展结点类
class Extension_node:
    def __init__(self):
        self.type = 'extension'
        self.key = None
        #把 branch 分支结点作为 extension 拓展结点的组成元素
        #对外只有 extension 和 leaf 两种类型的结点
        self.value = Branch_node()
        self.prefix = None
        #结点的 hash
        self.node_hash = None
        #结点下数据的 hash
        self.node_value = None

#定义 leaf 叶子结点类
class Leaf_node:
    def __init__(self):
        self.type = 'leaf'
        self.key_end = None
        self.value = None
        self.prefix = None
        #结点的 hash
        self.node_value = None
        #结点下数据的 hash
        self.node_hash = None

#定义 MPT 树类，包括如下操作：
#1.创建新结点；
#2.向前添加 extension 拓展结点；
#3.向后添加 extension 拓展结点；
#4.创建叶子结点；
#5.创建拓展结点；
#6.获取差异值索引；
#7.遍历 MPT 树查询；
#8.打印 MPT 树；
#9.更新 MPT 树；
#10.删除结点；
```

```
    #11.增加；
    #12.删除；
    #13.修改；
    #14.查找；
    #15.抛弃所有子结点的 value；
    #16.抛弃整棵树的 value
    class Tree:
        def __init__(self, tree=None):
            #构建对 MPT 树
            if tree is not None:
                self.root = tree
            else:#为结点创建新的 extension 拓展结点
                self.root = self.make_extension()
                #默认 root 作为根结点 prefix
                self.root.prefix = 'root'
                #定义 MPT 树的 value、hash
                self.value = None
                self.hash = None
        #创建新结点
        def add_node(self, node, key, value):
            #若父结点是 root
            if node.prefix == 'root':
                #且若父结点 root 下的 branch 分支结点是空的，即可直接插入（默认 key[0]）
                if self.root.value.children[key[0]] is None:
                    #key[1::]是后续传递的 new_key 值
                    #即去掉共同前缀的剩余部分作为下一步索引的前缀值
                     self.root.value.children[key[0]] = self.make_leaf(key[1::], key[1::], value)
                    #插入新的 leaf 叶子结点后，结点数据发生改变，更新结点状态
                    node.value.children['value'] = False
                    return
                #否则父结点 root 下 branch 表发生冲突，将冲突的结点位置作为参数进行递归
                else:
                     self.root.value.children[key[0]] = self.add_node(self.root.value.children[key[0]],
                                                                      key[1::], value)
                    return
            father = node
            #把 key 值与父结点的前缀字符比较，index 作为在当前 extension 拓展结点定位 branch 分支表位置的索引
            index = self.diff(father, key)
```

```
89.             #共同的前缀
90.             prefix = key[:index:]
91.             #去除（branch 分支索引）共同前缀后的剩余字符
92.             new_key = key[index::]
93.             #若相同字符数不等于共同前缀长度
94.             #则代表新结点与 father 结点没有共同前缀，产生冲突
95.             if index != len(father.prefix) and index < len(father.prefix)
    :
96.                 #extension 扩展结点产生冲突
97.                 if father.type == 'extension':
98.                     #向前创建新的 extension 拓展结点以解决冲突
99.                      return self.pre_extension(father, prefix, new_key, i
    ndex, value)
100.                #leaf 叶子结点产生冲突
101.                elif father.type == 'leaf':
102.                    #向后创建新的 extension 拓展结点以解决冲突
103.                     return self.pro_extension(father, prefix, new_key, i
    ndex, value)
104.            #否则就无冲突发生，进入拓展的 branch 分支结点中向下遍历
105.            else:
106.                #判断 extension 拓展结点下的 branch 分支结点对应 key 的 value 是
    否为空
107.                if father.value.children[key[index]] is None:
108.                    #若为空则添加 leaf 叶子结点
109.                     father.value.children[key[index]] = self.make_leaf(k
    ey[index + 1::], key[index])
110.                    #插入新的 leaf 叶子结点后，结点数据发生改变，更新结点状态
111.                    father.value.children['value'] = False
112.                    return father
113.                else:
114.                    #若非空，则发生字符表冲突，向下递归延展 extension 拓展结点
115.                     father = self.add_node(father.value.children[key[ind
    ex]], new_key, value)
116.                    return father
117.        #解决 extension 扩展结点与 leaf 叶子结点的冲突，向前添加 extension 拓展结
    点
118.        def pre_extension(self, node, prefix, key, index, value):
119.            node_new_prefix = node.prefix[index + 1::] #共同前缀
120.            #创建新的 extension 拓展结点
121.            tmp_node = self.make_extension()
122.            tmp_node.prefix = prefix#写入共同前缀
123.            #将旧 extension 拓展结点插入 branch 分支表中
124.            tmp_node.value.children[node.prefix[index]] = node
125.            #修改旧 extension 拓展结点的共同前缀
```

```
126.             tmp_node.value.children[node.prefix[index]].prefix = node_ne
    w_prefix
127.             #插入新 leaf 叶子结点
128.             tmp_node.value.children[key[0]] = self.make_leaf(key[1::], k
    ey[0], value)
129.             #返回新 extension 拓展结点
130.             return tmp_node
131.         #为解决 leaf 叶子结点与 leaf 叶子结点的冲突，向后添加 extension 拓展结点
132.         def pro_extension(self, node, prefix, key, index, value):
133.             leaf = node
134.             #创建新 extension 拓展结点
135.             tmp_node = self.make_extension()
136.             tmp_node.prefix = prefix#写入共同前缀
137.             #把旧 leaf 叶子结点插入 branch 分支表中
138.             tmp_node.value.children[leaf.key_end[index]] = leaf
139.             #产生共同前缀，leaf 叶子结点的 key_end 发生改变
140.             tmp_node.value.children[leaf.key_end[index]].key_end = leaf.
    key_end[index + 1::]
141.             #插入新 leaf 叶子结点
142.             tmp_node.value.children[key[0]] = self.make_leaf(key[1::], k
    ey[0], value)
143.             #返回新 extension 拓展结点
144.             return tmp_node
145.         #创建 leaf 叶子结点
146.         def make_leaf(self, key, profix, value):
147.             #初始化
148.             tmp_node = Leaf_node()
149.             tmp_node.key_end = key
150.             tmp_node.prefix = profix
151.             #添加 leaf 叶子结点的值和 hash
152.             tmp_node.value = value
153.             #对 value 进行 hash
154.             tmp_node.node_value = hashlib.sha256(value.encode('utf-8')).
    hexdigest()
155.             #对整个结点进行 hash，要在数据 hash 操作后进行
156.             tmp_node.node_hash = hashlib.sha256(str(tmp_node).encode('ut
    f-8')).hexdigest()
157.             #返回创建的 leaf 叶子结点
158.             return tmp_node
159.         #创建 extension 拓展结点
160.         def make_extension(self):
161.             #直接创建
162.             tmp_node = Extension_node()
163.             return tmp_node
```

```
164.     #获取差异值索引
165.     def diff(self, node, key):
166.         #将遍历长度定为 key 和 node.prefix 中长度较小的那一个，避免溢出
167.         if len(key) < len(node.prefix):
168.             lenth = len(key)
169.         else:
170.             lenth = len(node.prefix)
171.         count = 0
172.         while count < lenth: #遍历
173.             #如果遍历到有差异的地方，则返回差异的索引，否则继续遍历
174.             if node.prefix[count] != key[count]:
175.                 return count
176.             count += 1
177.         return count
178.
179.     #遍历 MPT 树
180.     def traverse_search(self, node, index):
181.         #返回的结点的索引
182.         result_node = None
183.         #遍历当前 extension 拓展结点的 branch 分支表
184.         for key in node.value.children:
185.             #若检测终止标志 value，则终止
186.             if key == 'value':
187.                 break
188.             #若检测到空值，则继续遍历
189.             if node.value.children[key] is None:
190.                 continue
191.             #若检测到 leaf 叶子结点，就对比 key_end 和索引值
192.             if node.value.children[key].type == 'leaf':
193.                 #若匹配
194.                 if index[1::] == node.value.children[key].key_end:
195.                     #返回该结点，并结束遍历
196.                     result_node = node.value.children[key]
197.                     break
198.                 #否则继续检测
199.                 else:
200.                     continue
201.             #若检测到 extension 扩展结点，则进入该结点的 branch 分支表向下索
    引
202.             elif node.value.children[key].type == 'extension':
203.                 #记录去除该 extension 拓展结点的共同前缀后剩余的索引值
204.                  short_key = index[len(node.value.children[key].prefi
    x) + 1::]
205.                 #递归向下索引
```

```
206.                     result_node = self.traverse_search(node.value.childr
    en[key], short_key)
207.                 #若检测到不为空的结点
208.                 if result_node is not None:
209.                     #返回该结点，结束遍历
210.                     break
211.                 #否则，继续检测
212.                 else:
213.                     continue
214.         #返回检测到的结点的索引
215.         return result_node
216.     #打印 MPT 树：遍历 MPT 树，在遍历期间打印遇到的非空结点信息
217.     def print_all(self, node):
218.         print('extension of prefix:', node.prefix)
219.         #遍历当前 extension 拓展结点的 branch 分支表
220.         for key in node.value.children:
221.             #若检测终止标志 value，终止
222.             if key == 'value':
223.                 break
224.             #若检测到空值，继续遍历
225.             if node.value.children[key] is None:
226.                 continue
227.             #若检测到 leaf 叶子结点，则打印 branch 分支结点和 key_end
228.             if node.value.children[key].type == 'leaf':
229.                 print('branch:', key)
230.                  print('leaf of key_end:', node.value.children[key].k
    ey_end)
231.             #若检测到 extension 扩展结点，则打印 branch 分支表，并递归遍历打
    印所有的非空结点信息
232.             elif node.value.children[key].type == 'extension':
233.                 print('branch:', key)
234.                 self.print_all(node.value.children[key])
235.     #更新 MPT 树：查询之前需要进行更新
236.     #即遍历MPT树，自下向上对每个extension扩展结点的value和hash进行更新
237.     def update_tree(self, node):
238.         #临时 string，用于聚合 extension 扩展结点下 branch 分支表中非空结点
    的 value 值
239.          #extension 扩展结点的 value 值产生自对聚合结果的 hash，即该临时
    string 的 hash
240.         tmp_str = ''
241.         #当前结点状态为 True，已更新，则直接返回当前值
242.         if node.value.children['value']:
243.             return node.node_value
244.         #否则遍历结点
```

```
245.             for key in node.value.children:
246.                 #若检测终止标志 value，则终止
247.                 if key == 'value':
248.                     break
249.                 #若检测到空值，则继续遍历
250.                 if node.value.children[key] is None:
251.                     continue
252.                 #若检测到 leaf 叶子结点，则聚合 leaf 叶子结点
253.                 if node.value.children[key].type == 'leaf':
254.                     tmp_str = tmp_str + node.value.children[key].node_va
    lue
255.                  #若检测到 extension 扩展结点，则递归遍历聚合 extension 扩展结
    点
256.                 elif node.value.children[key].type == 'extension':
257.                     tmp_str = tmp_str + self.update_tree(node.value.chil
    dren[key])
258.             #修改结点状态为 True
259.             node.value.children['value'] = True
260.             #利用聚合的 value 值更新结点的 value 和 hash 值
261.              node.node_value = hashlib.sha256(tmp_str.encode()).hexdigest
    ()
262.              node.node_hash = hashlib.sha256(str(node).encode()).hexdiges
    t()
263.             #打印结点的 prefix 和 value
264.             print('prefix:', node.prefix)
265.             print('node_value:', node.node_value)
266.             #返回更新的结点
267.             return node.node_value
268.         #删除结点：通过遍历找到需要删除的结点
269.         #将需要删除的结点对应的 branch 分支的位置设为 None
270.         def delete_node(self, node, hash):
271.             #进行遍历
272.             for key in node.value.children:
273.                 #若检测终止标志 value，则终止
274.                 if key == 'value':
275.                     break
276.                 #若检测到空值，则继续遍历
277.                 if node.value.children[key] is None:
278.                     continue
279.                 #若检测到 leaf 叶子结点，则对比 key_end 和索引值
280.                 if node.value.children[key].type == 'leaf':
281.                     #若匹配，则删除该结点，并将其重置为 None，并返回 True
282.                     if hash[1::] == node.value.children[key].key_end:
283.                         del node.value.children[key]
```

```
284.                         node.value.children[key] = None
285.                         return True
286.                     #否则继续遍历
287.                     else:
288.                         continue
289.                 #若检测到 extension 扩展结点，记录去除该 extension 拓展结点的共同前缀后剩余的索引值
290.                 elif node.value.children[key].type == 'extension':
291.                     short_hash = hash[len(node.value.children[key].prefix) + 1::]
292.                     #若剩余的索引值为空
293.                     if short_hash == '':
294.                         #删除该结点
295.                         del node.value.children[key]
296.                         #将其重置为 None
297.                         node.value.children[key] = None
298.                         print('delete')
299.                         #返回 True
300.                         return True
301.                     #否则，继续递归遍历，直到找到剩余的索引值为空的结点并删除
302.                     #并将其重置为 None，并返回 True
303.                     elif self.delete_node(node.value.children[key], short_hash):
304.                         return True
305.         #增加操作：后续需要将 update_tree 精准到结点上
306.         #而不是每次都从 root 开始，最后再对 MPT 树进行更新
307.         def add(self, key, value, node=None):
308.             #若结点是空的，进行构建
309.             if node is None:
310.                 node = self.root
311.             #进行递归增
312.             self.add_node(node, key, value)
313.             #更新树
314.             self.update_tree(self.root)
315.
316.         #删除操作：最后对 MPT 树进行更新
317.         def delete(self, key):
318.             print('delete from str')
319.             #进行递归删
320.             self.delete_node(self.root, key)
321.             #更新树
322.             self.update_tree(self.root)
323.
324.         #修改操作：修改 leaf 叶子结点的 value 值，最后再对 MPT 树进行更新
```

```
325.     def update(self, index, value):
326.         #若是字符串类型
327.         if type(index) == str:
328.             #则进行 MPT 树的遍历查询
329.             tmp_node = self.traverse_search(self.root, index)
330.             #修改结点的 value 值
331.             tmp_node.value = value
332.             #对 value 进行 hash
333.              tmp_node.node_value = hashlib.sha256(value.encode('utf-8
    ')).hexdigest()
334.             #对整个结点进行 hash
335.              tmp_node.node_hash = hashlib.sha256(str(tmp_node).encode
    ('utf-8')).hexdigest()
336.         #若不是字符串类型
337.         else:
338.             #直接修改 value 值
339.             index.value = value
340.             #对 value 进行 hash
341.             index.node_value = hashlib.sha256(value.encode('utf-8')).
    hexdigest()
342.             #对整个结点进行 hash
343.              index.node_hash = hashlib.sha256(str(index).encode('utf-
    8')).hexdigest()
344.         #更新树
345.         self.update_tree(self.root)
346.
347.     #查找操作：提供接口
348.     def search(self, index):
349.         #若是字符串类型
350.         if type(index) == str:
351.             #进行 MPT 的遍历查询
352.             return self.traverse_search(self.root, index).value
353.         #若不是字符串类型
354.         else:
355.             #直接返回 value 值
356.             return index.value
357.
358.     #抛弃所有 leaf 叶子结点的 value：遍历整个 MPT 树
359.     #删除 leaf 叶子结点的 value 并设置为 None
360.     def drop_all_value(self, node=None):
361.         #若结点是空的，进行构建
362.         if node is None:
363.             node = self.root
364.         #遍历
```

```
365.            for key in node.value.children:
366.                #若检测终止标志 value，终止
367.                if key == 'value':
368.                    break
369.                #若检测到空值，继续遍历
370.                if node.value.children[key] is None:
371.                    continue
372.                #若检测到 leaf 叶子结点，则直接删除 value
373.                if node.value.children[key].type == 'leaf':
374.                    del node.value.children[key].value
375.                    #把 value 设置为 None
376.                    node.value.children[key].value = None
377.                #若检测到 extension 扩展结点，则递归遍历直到叶子结点
378.                elif node.value.children[key].type == 'extension':
379.                    self.drop_all_value(node.value.children[key])
380.
381.        #抛弃整棵树的 value：保留 root 根结点的 value 和 hash
382.        def drop_tree(self):
383.            #对要进行操作的树进行跟更新
384.            self.update_tree(self.root)
385.            #把 value 设置为根结点 root 的 value
386.            self.value = self.root.node_value
387.            #把 hash 设置为根结点 root 的 hash
388.            self.hash = self.root.node_hash
389.            #删掉原来树的根结点
390.            del self.root
391.            #设置为 None
392.            self.root = None
```

23. 笔记本相关信息


设备规格
复制
设备名称 LAPTOP-4O00RE59
处理器 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
机带 RAM 16.0 GB (15.8 GB 可用)
设备 ID 142DC9CF-1506-4C5C-9731-FDB773F89354
产品 ID 00342-36159-60807-AAOEM
系统类型 64 位操作系统, 基于 x64 的处理器
笔和触控 没有可用于此显示器的笔或触控输入
相关链接 域或工作组 系统保护 高级系统设置
Windows 规格
复制
版本 Windows 11 家庭中文版
版本 22H2
安装日期 2023/3/18
操作系统版本 22621.1848
序列号 MP1ZK3PX
体验 Windows Feature Experience Pack 1000.22642.1000.0
Microsoft 服务协议
Microsoft 软件许可条款