

# 创新创业实践作业总结报告

陈子豪，程雨森，王佳乐，赵嵘晖

2023 年 8 月 4 日

## 摘要

我们组四个组员通力合作、各司其职，最后完成了 project1,2,3,4,5,6,7,11,12,13,14,15,16,17,22

**关键词：** SM2,SM3,SM4,ETH,zk-SNARK,BTC

### 任务分配：

陈子豪 202100460028： project12 项目报告总结编写

程雨森 202100460090： project1,2,3,4,5,6 以及资料整理

王佳乐 202100460028： project7,17,22 以及资料查询

赵嵘晖 202100460100： project11,13,14,15,16 并负责 github 库的运维

# 目录

<b>1</b>	<b>SM2 算法原理</b>	<b>1</b>
1.1	参数选择 . . . . .	1
1.2	密钥生成 . . . . .	1
1.3	加密 . . . . .	1
1.4	解密 . . . . .	2
1.5	签名 . . . . .	4
1.6	验证 . . . . .	5
<b>2</b>	<b>SM2 密钥协商</b>	<b>6</b>
2.1	原理 . . . . .	6
2.2	实现 . . . . .	8
<b>3</b>	<b>Project1:implement the naïve birthday attack of reduced SM3</b>	<b>9</b>
3.1	思路分析 . . . . .	9
3.2	运行结果 . . . . .	10
<b>4</b>	<b>Project2:implement the Rho method of reduced SM3</b>	<b>10</b>
4.1	思路分析 . . . . .	10
4.2	运行结果 . . . . .	10
<b>5</b>	<b>Project3:implement length extension attack for SM3, SHA256, etc.</b>	<b>11</b>
5.1	思路分析 . . . . .	11
5.2	运行结果 . . . . .	11
<b>6</b>	<b>Project4: do your best to optimize SM3 implementation (software)</b>	<b>12</b>
6.1	思路分析 . . . . .	12
6.2	运行结果 . . . . .	12
<b>7</b>	<b>Project5: Impl Merkle Tree following RFC6962</b>	<b>13</b>
7.1	思路分析 . . . . .	13
7.2	运行结果 . . . . .	15

目录	II
<b>8 Project6: impl this protocol with actual network communication</b>	<b>16</b>
8.1 思路分析 . . . . .	16
8.2 运行结果 . . . . .	16
<b>9 Project7: Try to Implement this scheme</b>	<b>17</b>
9.1 思路分析 . . . . .	17
9.2 实现方式 . . . . .	17
9.3 运行结果 . . . . .	19
<b>10 Project11: impl sm2 with RFC6979</b>	<b>19</b>
10.1 思路分析 . . . . .	19
10.1.1 RFC6979 . . . . .	19
10.1.2 具体想法 . . . . .	20
10.2 运行结果 . . . . .	20
10.3 实验效率 . . . . .	21
<b>11 Project12: verify the above pitfalls with proof-of-concept code</b>	<b>22</b>
11.1 思路分析 . . . . .	22
11.2 实现方式 . . . . .	23
11.3 运行结果 . . . . .	25
<b>12 Project13: Implement the above ECMH scheme</b>	<b>25</b>
12.1 思路分析 . . . . .	25
12.2 实现方式 . . . . .	25
12.3 运行结果 . . . . .	28
<b>13 Project14: Implement a PGP scheme with SM2</b>	<b>29</b>
13.1 思路分析 . . . . .	29
13.1.1 PGP 加密 . . . . .	29
13.1.2 PGP 解密 . . . . .	30
13.2 运行结果 . . . . .	32

<b>14 Project15: implement sm2 2P sign with real network communication</b>	<b>33</b>
14.1 思路分析 . . . . .	33
14.2 运行结果 . . . . .	34
<b>15 Project16: implement sm2 2P decrypt with real network communication</b>	<b>35</b>
15.1 思路分析 . . . . .	35
15.2 运行结果 . . . . .	36
<b>16 Project17: 比较 Firefox 和谷歌的记住密码插件的实现区别</b>	<b>37</b>
16.1 问题分析 . . . . .	37
<b>17 Project22: research report on MPT</b>	<b>38</b>
17.1 MPT 的优点 . . . . .	38
17.2 MPT 在以太坊中的用途 . . . . .	39



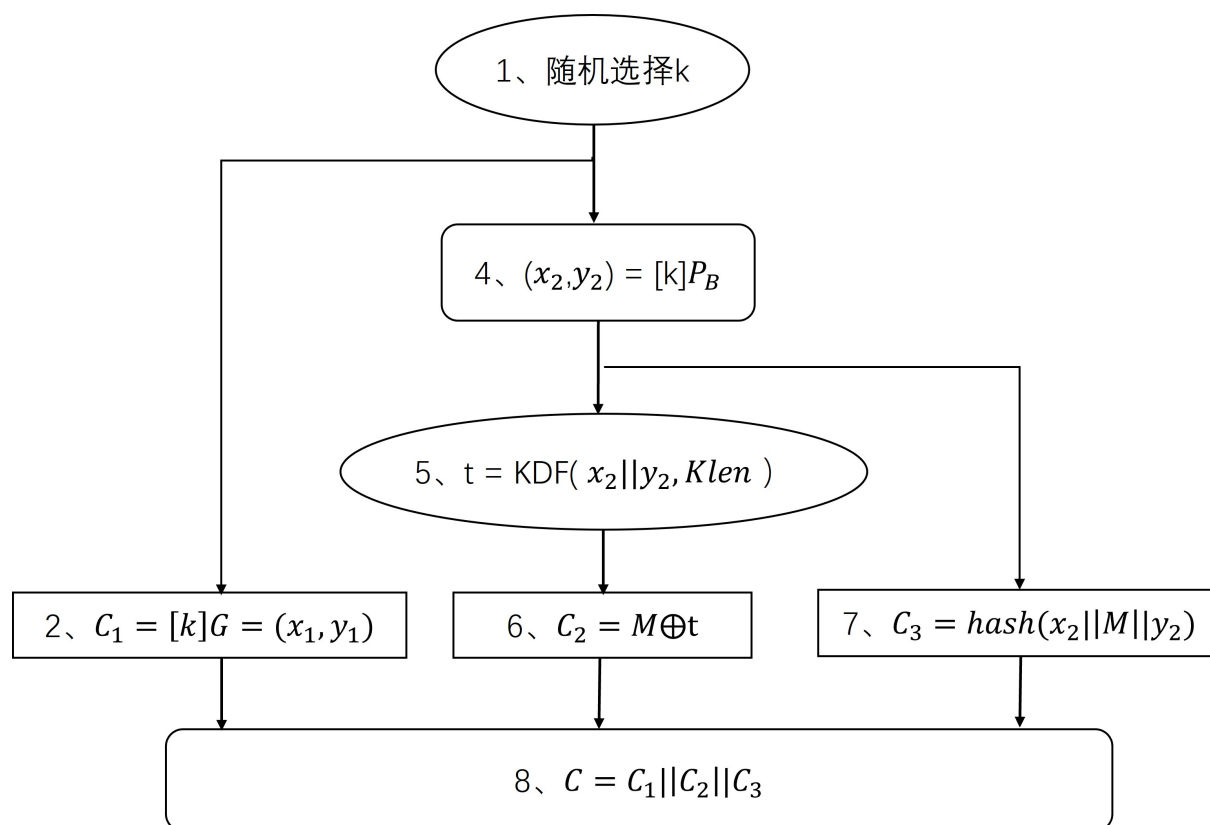


图 1: SM2 加密算法流程图

算法如下 Algorithm 1 所示。

## 1.4 解密

流程图如下图 2 所示。

**Algorithm 1** SM2 Encryption**Input:** M**Output:** C

- 1: Compute the bit length of the M:klen
- 2: Select a random number  $k \in [1, n - 1]$
- 3: Compute  $C_1 = [k]G = (x_1, y_1)$
- 4: Compute  $h = |E(F_q)|/n$
- 5: Compute  $C_1 = [k]G = (x_1, y_1)S = hP_B$
- 6: **if** S is  $\bigcirc$  **then**
- 7:     **return** Error
- 8: **end if**
- 9: Compute  $kP_b = (x_2, y_2)$
- 10: Compute  $t = \text{KDF}(x_2 || y_2, \text{Klen})$
- 11: **if**  $t == 0$  **then**
- 12:     Select a random number k again
- 13: **end if**
- 14: Compute  $C_2 = M \oplus t$
- 15: Compute  $C_3 = \text{hash}(x_2 || M || y_2)$
- 16: Compute  $C = C_1 || C_2 || C_3$
- 17: **return** C

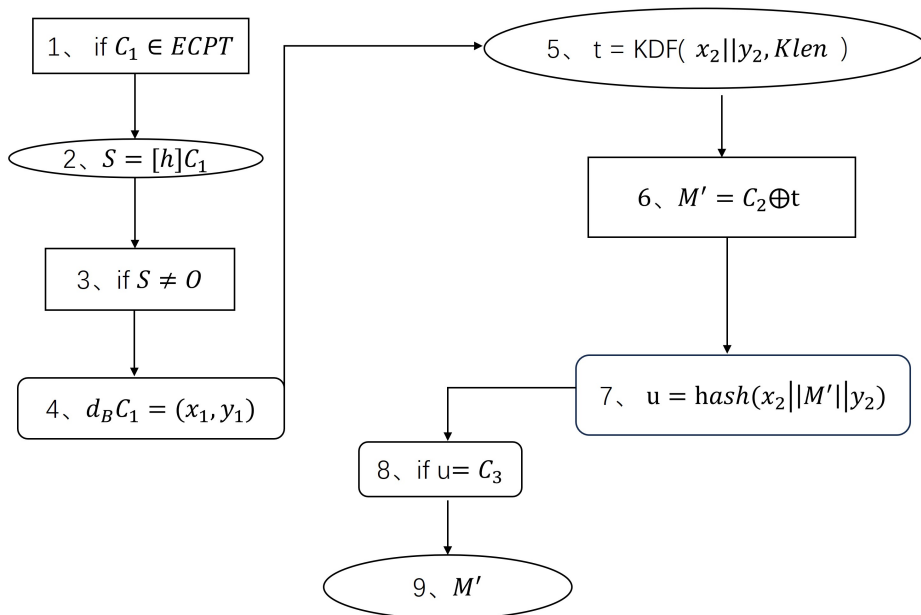


图 2: SM2 解密算法流程图

算法如下 Algorithm 2 所示。

---

**Algorithm 2** SM2 Decryption
 

---

**Input:** C

**Output:** M'

```

1: if C1∉ECPT then
2:   return ERROR
3: end if
4: Compute S=hC1
5: if S=O then
6:   return ERROR
7: end if
8: Compute dBC1
9: Compute t=KDF(x2||y2,Klen)
10: Compute M'=C2⊕t
11: Compute u=hash(x2||M'||y2)
12: if u≠C3 then
13:   return ERROR
14: end if
15: return M'

```

---

## 1.5 签名

先预计算  $Z_A = \text{Hash}(\text{ENTL}_A \| ID_A \| a \| b \| x_G \| y_G \| x_A \| y_A)$ ,  $\text{entl}_A$  是  $ID_A$  的长度,  $\text{ENTL}_A$  为  $\text{entl}_A$  的两字节表示。HASH 函数采用 SM3。

流程图如下图 3 所示。



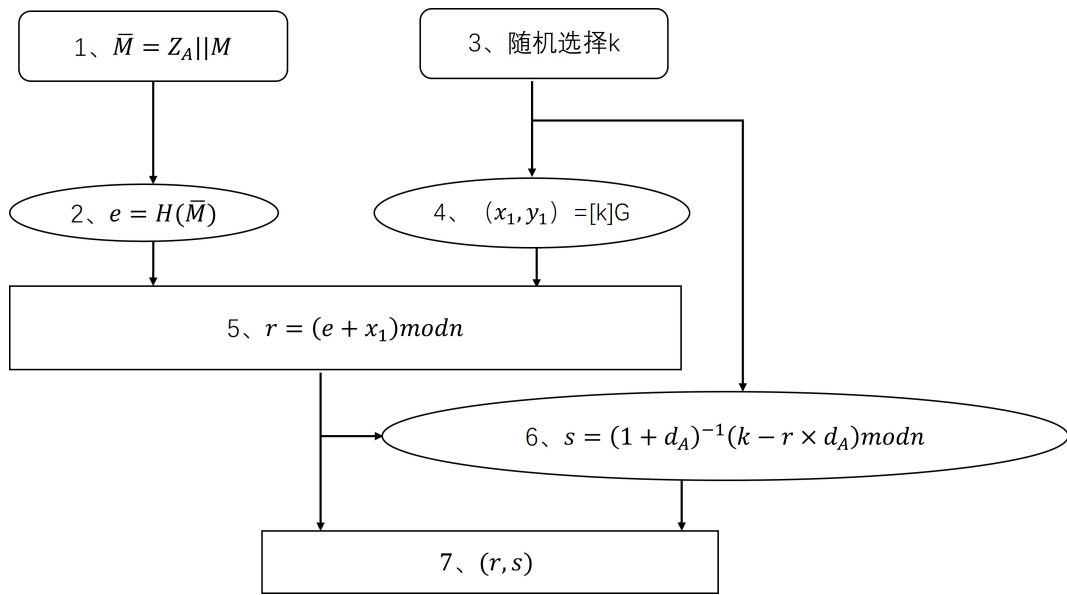


图 3: SM2 签名算法流程图

算法如下 Algorithm 3 所示。

## 1.6 验证

流程图如下图 4 所示。

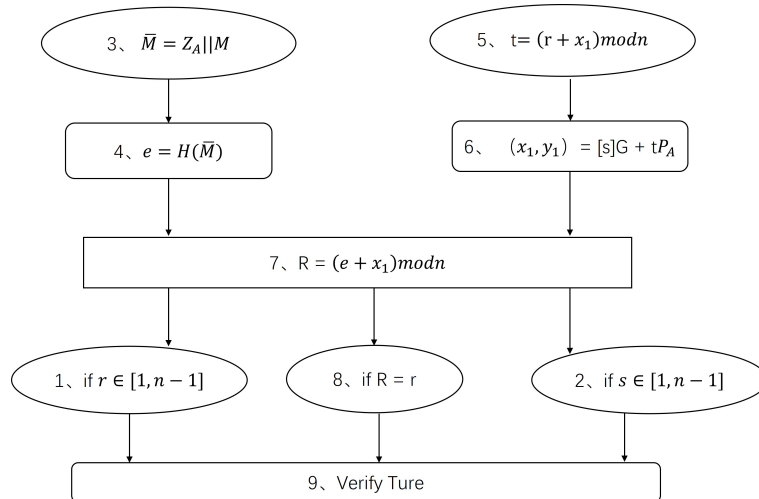


图 4: SM2 签名验证算法流程图

算法如下 Algorithm 4 所示。

---

**Algorithm 3** SM2 Signature

---

**Input:** M**Output:** (r,s)

- 1: Compute  $Z_A = \text{Hash}(\text{ENTL}_A || \text{ID}_A || a || b || x_G || y_G || x_A || y_A)$
  - 2: Compute  $\overline{M} = Z_A || M$
  - 3: Compute  $e = \text{Hash}(\overline{M})$
  - 4: Select a random number  $k \in [1, n - 1]$
  - 5: Compute  $KG = (x_1, y_1)$
  - 6: Compute  $r = (e + x_1) \bmod n$
  - 7: **if**  $r == 0$  or  $r + k == n$  **then**
  - 8:     **return** Select a random number  $k$  again
  - 9: **end if**
  - 10: Compute  $s = (1 + d_A)^{-1} (k - rd_A)$
  - 11: **if**  $s == 0$  **then**
  - 12:     Select a random number  $k$  again
  - 13: **end if**
  - 14: **return** (r,s)
- 

## 2 SM2 密钥协商

### 2.1 原理

系统参数:

- 1、域  $F_q$  的描述
- 2、椭圆曲线的两个定义元  $a, b \in F_q$
- 3、 $E(F_q)$  的基点  $G(x_G, y_G) \neq O$ , 其中  $x_G, y_G \in F_q$
- 4、 $G$  的阶  $n$ ,  $[n]G = O$
- 5、其他可选项, 如  $n$  的余因子  $h: = \frac{\#E(F_q)}{n}$
- 6、用户 A 的密钥对: 私钥  $d_A$ , 公钥  $P_A = [d_A]G = (x_A, y_A)$
- 7、用户 B 的密钥对: 私钥  $d_B$ , 公钥  $P_B = [d_B]G = (x_B, y_B)$
- 8、用户 A 的可辨别标识  $ID_A$ , 长度  $\text{Len}_A$  比特

---

**Algorithm 4** SM2 Signature

---

**Input:** (r,s)**Output:** False or TURE

```

1: Compute  $Z_A = \text{Hash}(\text{ENTL}_A || \text{ID}_A || a || b || x_G || y_G || x_A || y_A)$ 
2: Check  $r \in [1, n-1]$ 
3: Check  $s \in [1, n-1]$ 
4: Compute  $\overline{M} = Z_A || M$ 
5: Compute  $e = \text{hash}(\overline{M})$ 
6: Compute  $t = (r + x_1) \bmod n$ 
7: Compute  $(x_1, y_1) = sG + tP_A$ 
8: Compute  $R = (e + x_1) \bmod n$ 
9: if  $R \neq r$  then
10:   return FALSE
11: end if
12: return TURE

```

---

9、用户 B 的可辨别标识  $ID_B$ ，长度  $Elen_B$  比特

10、 $Z_A = H_{256}(Elen_A || ID_A || a || b || G_x || G_y || x_A || y_A)$

11、 $Z_B = H_{256}(Elen_B || ID_B || a || b || G_x || G_y || x_B || y_B)$

12、 $w = \lceil \frac{\lceil \log_2 n \rceil}{2} \rceil - 1$

主要函数:

1、密钥派生函数 KDF

2、伪随机数生成器 PRG

3、杂凑函数 SM3

流程图如下图 5 所示。

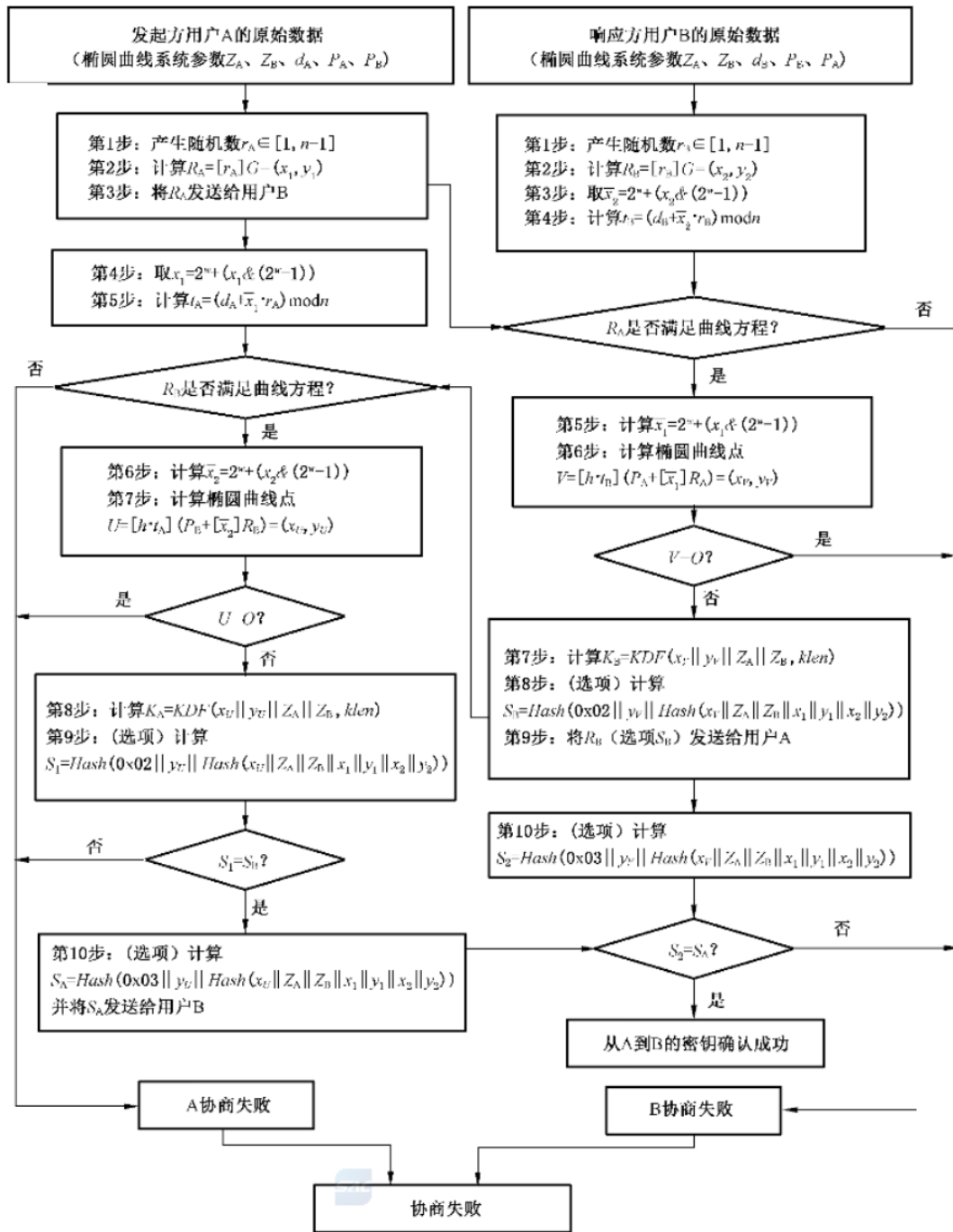


图 5: SM2 密钥协商算法

## 2.2 实现

SM2 密钥协商有两个用户 A 和 B。

其中 A 先随机选取  $r_A$ ,  $r_A \in [1, n-1]$ 。A 计算椭圆曲线上的点  $R_A = [r_A]G = (x_1, y_1)$ 。A 将  $R_A$  发给 B。

B 先随机选取  $r_B$ ,  $r_B \in [1, n-1]$ 。B 计算椭圆曲线上的点  $R_B = [r_B]G = (x_2, y_2)$ 。B 计算  $x_{B2} = 2^w + x_2 \& (2^w - 1)$ ,  $t_B = d_B + x_{B2} \cdot r_B \bmod n$ 。

B 判断  $R_A$  是否在椭圆曲线上, 如果不在那么协商失败。如果在那么 B 计算  $x_{A1} = 2^w + x_1 \& (2^w - 1)$ 。计算椭圆曲线上的点  $V = [h \cdot]t_B(P_A + [x_{A1}]R_A) = (x_V, y_V)$ 。B 计算  $K_B = KDF(x_V || y_V || Z_A || Z_B, klen)$ , 其中  $klen$  是协商密钥的比特长度。B 再计算  $S_B = Hash(0X02 || y_V || Hash(x_V || Z_A || Z_B || x_1 || y_1 || x_2 || y_2))$ 。B 将  $R_B$  和  $S_B$  发送给 A。

A 收到后先计算  $x_{A1} = 2^w + x_1 \& (2^w - 1)$ ,  $t_A = d_A + x_{A1} \cdot r_A \bmod n$ , 并判断  $R_B$  是否在椭圆曲线上, 如果不在那么协商失败。

如果在那么 A 计算  $x_{B2} = 2^w + x_2 \& (2^w - 1)$ 。计算椭圆曲线上的点  $U = [h \cdot]t_A(P_B + [x_{B2}]R_B) = (x_U, y_U)$ 。A 再计算  $S_1 = Hash(0X02 || y_U || Hash(x_U || Z_A || Z_B || x_1 || y_1 || x_2 || y_2))$ 。A 判断  $S_1$  是否与  $S_B$  相等, 如果不相等那么协商失败。

如果相等, A 计算  $K_A = KDF(x_U || y_U || Z_A || Z_B, klen)$ , 其中  $klen$  是协商密钥的比特长度。A 再计算  $S_A = Hash(0X03 || y_U || Hash(x_U || Z_A || Z_B || x_1 || y_1 || x_2 || y_2))$ 。A 将  $S_A$  发送给 B。

B 计算  $S_2 = Hash(0X03 || y_V || Hash(x_V || Z_A || Z_B || x_1 || y_1 || x_2 || y_2))$ 。B 判断  $S_2$  是否与  $S_A$  相等, 如果不相等那么协商失败。如果相等那么协商成功。

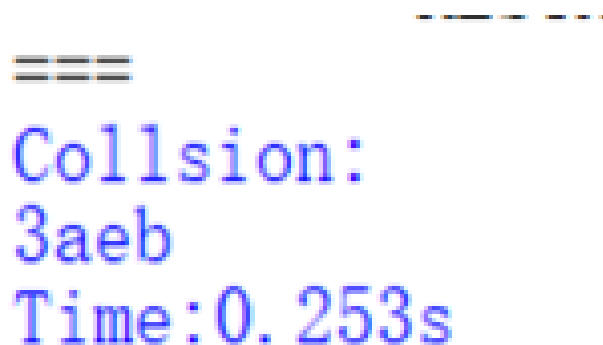
此时  $K_A = K_B$ 。

### 3 Project1:implement the naïve birthday attack of reduced SM3

#### 3.1 思路分析

生成了一个包含  $2^{16}$  个随机 64 位整数的列表。然后, 它对每个整数进行了 SM3 哈希, 并提取了哈希值的前 7 个十六进制数字 (对应哈希值的前 28 位比特) 接着使用 Python 的 Counter 类统计每个哈希值出现的次数, 如果有任意两个哈希值的前 28 位相同, 则认为发生了哈希碰撞, 并将其输出。这样, 就可以利用生日攻击破解 SM3 哈希函数。

## 3.2 运行结果



```
=====  
Collision:  
3aeb  
Time:0.253s
```

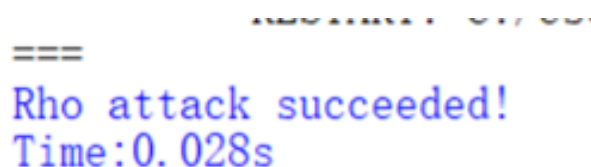
图 6: project1 结果

# 4 Project2:implement the Rho method of reduced SM3

## 4.1 思路分析

首先定义了 SM3 哈希函数的一些基本操作，然后，实现了 SM3 哈希函数的核心部分，包括填充、分块、消息扩展和消息压缩等操作。接着，定义了一个名为 rho\_attack 的函数，该函数生成随机的 64 位整数列表，并对每个整数进行哈希处理。然后，提取哈希值的第一个十六进制数字，并将该数字存储在列表中。最后，检查列表中是否存在相同的数字，如果存在，则认为发生了哈希碰撞，从而成功地执行了 Rho 攻击。

## 4.2 运行结果



```
=====  
Rho attack succeeded!  
Time:0.028s
```

图 7: project2 结果



## 6 Project4: do your best to optimize SM3 implementation (software)

### 6.1 思路分析

在代码中定义了这几个函数 `leftshift()`: 实现循环左移的函数, `FF()` 和 `GG()`: SM3 压缩函数中的两个置换函数, `P0()` 和 `P1()`: SM3 压缩函数中的两个线性函数, `compress()`: SM3 压缩函数, 其中, `FF()` 和 `GG()` 是 SM3 压缩函数中的两个置换函数, 用于对数据进行混淆; `P0()` 和 `P1()` 是 SM3 压缩函数中的两个线性函数, 用于对数据进行线性变换。 `compress()` 函数是 SM3 的核心计算部分, 用于将每个 512 位的消息块压缩成 256 位的摘要。在该函数内部, 先将消息块按照一定的规则进行填充, 然后将填充后的消息块拆分成 16 个 32 位的字, 再扩展到 68 个 32 位字, 最后进行 64 轮计算 (每轮计算包括置换、线性函数和循环左移等操作), 最终得到 256 位的摘要。

代码实现了三种不同的计算方式: 串行计算、4 线程并行计算和 8 线程并行计算。其中, 4 线程和 8 线程的并行计算使用了 C++11 中的 `std::thread` 库。

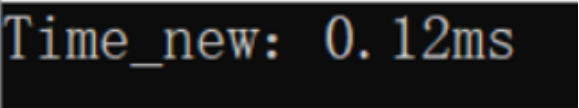
### 6.2 运行结果

初始运行时间:



```
Time: 337.69ms
```

优化后运行时间:



```
Time_new: 0.12ms
```

图 9: project4 结果



## 7 Project5: Impl Merkle Tree following RFC6962

### 7.1 思路分析

为了实现具体功能，在代码中定义了 Merkle 树的节点结构体 `merkletree`，其中包含左右子节点指针、父节点指针、节点层数、节点数据和节点字符串；定义了哈希函数 `hash` 和 `hash_nodes`，分别用于对字符串和节点数据进行哈希；定义了函数 `last_node` 和 `find_new_node`，分别用于查找最后一个节点和查找新节点，这两个函数在构建 Merkle 树时用到；定义了函数 `initial`，用于初始化 Merkle 树。该函数首先将字符串按照标点符号和空格进行分割，得到一个字符串数组。然后根据字符串数组中的每个字符串构建 Merkle 树的叶子节点，并根据节点的哈希值构建 Merkle 树的中间节点和根节点。在构建过程中，如果某个节点的左右子节点都已经存在，则需要查找新节点，并将新节点插入到 Merkle 树中。最后返回根节点。

定义了函数 `print_tree`，用于打印 Merkle 树。该函数采用递归方式遍历 Merkle 树，并根据节点的层数打印相应数量的缩进和节点数据或字符串。定义了函数 `delete_tree` 和 `delete_string`，分别用于释放 Merkle 树和字符串数组的内存空间。

在 `main` 函数中，首先定义了一个包含字符串的数组 `message`，然后调用 `divide_string` 函数将字符串按照标点符号和空格进行分割，并得到字符串数组 `s` 和字符串数量 `n`。接着调用 `initial` 函数初始化 Merkle 树，并调用 `print_tree` 函数打印 Merkle 树。最后释放字符串数组和 Merkle 树的内存空间，并返回 0。



## 7.2 运行结果

```
Initialization finish!
Merkle Tree:
-->Hello
-->29931
-->!
-->441469
-->This
-->33136
-->is
-->215355882
-->Baekhune
-->2273362
-->.
-->30323657
-->I'm
-->2058589
-->writing
-->1549874312
-->a
-->109847
-->merkle
-->864962
-->tree
-->13719
-->!
-->6054734
```

图 10: project5 结果

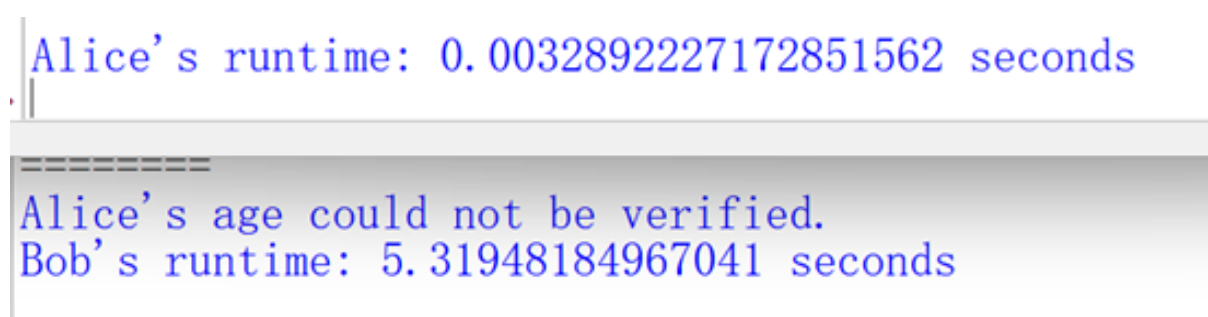
## 8 Project6: impl this protocol with actual network communication

### 8.1 思路分析

Alice 的代码中，首先定义了变量 `age` 和 `r`，并根据这两个变量计算出哈希值 `v`。然后创建一个 TCP 套接字，并连接到指定的主机和端口。接着将哈希值和年龄发送给服务器，并关闭套接字。Bob 的代码中，首先创建一个 TCP 套接字，并绑定到指定的主机和端口。然后监听连接请求，并等待客户端的连接。一旦连接建立成功，就从客户端接收哈希值和年龄。接着根据收到的年龄和预定的随机数 `r` 计算哈希值 `v_prime`，并与收到的哈希值进行比较。如果两个哈希值相等，则打印验证通过的消息，否则打印验证失败的消息。最后关闭连接和套接字。

该代码的功能是验证 Alice 的年龄是否正确。具体实现方式是，Alice 在本地计算出一个哈希值，并将这个哈希值和自己的年龄发送给 Bob。Bob 接收到这些信息后，先计算出自己预定的随机数和 Alice 的年龄的哈希值，然后将这个哈希值与 Alice 发送过来的哈希值进行比较。如果两个哈希值相等，则说明 Alice 的年龄是正确的，否则说明 Alice 的年龄可能被篡改了。这种方式可以保护 Alice 的隐私，因为 Bob 不需要知道 Alice 的确切年龄，只需要验证其年龄是否正确即可，任何对 Alice 的年龄的篡改都会导致哈希值的不一致。

### 8.2 运行结果



```
Alice's runtime: 0.0032892227172851562 seconds
=====  
Alice's age could not be verified.  
Bob's runtime: 5.31948184967041 seconds
```

图 11: project6 结果

## 9 Project7: Try to Implement this scheme

### 9.1 思路分析

在本次实验中我们使用泛化哈希链实现验证一个秘密数 `secret` 在某个区间  $[a,b]$  内。

选择一个合适的区间  $[a,b]$ ，表示想要证明的秘密整数  $x$  的范围。选择一个合适的哈希函数族  $\{H_k|k \text{ 是任意整数}\}$ 。

需要生成一个初始值 `seed`，它可以是任意的字符串或数字。计算出  $x$  在区间  $[a, b]$  中对应的二进制表示  $b_x$ ，并且将其分成  $n$  个比特位。

根据  $b_x$  中每个比特位的值来选择不同的参数  $k$ ，并且用它们来构造一个泛化哈希链。具体地说，如果  $b_x[i] = 0$ ，那么选择  $k=2i$ ；如果  $b_x[i] = 1$ ，选择  $k=-2i$ 。然后，可以用这些参数  $k$  来生成  $n$  个哈希值  $h_0, h_1, \dots, h_{n-1}$ ，其中  $h_i = H_k(s)$ 。如果  $b_x=101010$ ，并且  $s="hello"$ ，那么  $h_0 = SHA-256-1("hello")$   $h_1 = SHA-256-2("hello")$ ； $h_2 = SHA-256-4("hello")$   $h_3 = SHA-256-8("hello")$ ； $h_4 = SHA-256-16("hello")$   $h_5 = SHA-256-32("hello")$ 。

将种子  $s$  和  $n$  个哈希值  $h_0, h_1, \dots, h_{n-1}$  组成一个证明  $p$ ，并且将其发送给验证者。验证者可以根据以下方法来验证证明  $p$ ：

首先，验证者需要知道区间  $[a, b]$  和哈希函数族  $\{H_k|k \text{ 是任意整数}\}$ 。

然后，验证者需要从证明  $p$  中提取出种子  $s$  和  $n$  个哈希值  $h_0, h_1, \dots, h_{n-1}$ 。

接下来，验证者需要根据哈希值  $h_0, h_1, \dots, h_{n-1}$  来重构出  $b_x$  中每个比特位的值。如果  $h_i = H_k(s)$ ，那么验证者就可以根据  $k$  的正负号来判断  $b_x[i]$  的值。如果  $k>0$ ，那么  $b_x[i]=0$ ；如果  $k<0$ ，那么  $b_x[i]=1$ 。

最后，验证者需要根据  $b_x$  中的二进制表示来计算出  $x$  的值，并且检查是否满足  $x$  在区间  $[a, b]$  中。

### 9.2 实现方式

在实验中我们选择了 python 自带的 `hashlib` 库中的 `SHA-256` 作为哈希函数。通过将 `seed` 与二进制数  $k$  相异或得到一系列的哈希函数族。

```
1 def hash_encode(s,k):
2     # 创建一个 SHA-256 哈希对象
```

```
3 m = hashlib.sha256()
4 # 向哈希对象中添加数据
5 m.update(bxor(s,k).encode())
6 # 获取哈希值
7 h = m.digest()
8 # 将哈希值转换为 16 进制字符串
9 d = m.hexdigest()
10 #打印哈希值的 16 进制
11 print("0x",d,"\n")
12 return d
```

生成  $b_x$  时我们采用  $\text{srcert}$  在区间  $[a,b]$  的相对位置的二进制，并取  $\log(b,2)$  的下界为位数  $n$ ，表示要表示  $b_x$  需要的位数。之后生成一系列的哈希链。在比较时我们选择比较正确返回 '0'，失败则返回 '1'，从而实现对  $b_x$  的还原。要注意的是，此时验证者并不能知道  $\text{secret}$  具体是哪一个数

### 9.3 运行结果

```

The string matches the hash.
2
0x 24bc4a0130a829ee557b8794728d4b36000c347103d64e7367ddf7dcb2548c4d

The string matches the hash.
4
0x bed2d7608a6721bb8b9f43ae2d2af56a36e0a0430766dba344b3540db47894a8

The string does not match the hash.
8
0x 007db93d88f1ecc7d021f941624e89cfbd8fcb26a095f27505abb29bf46efa1a

The string matches the hash.
16
0x bf63543edc6c1239c6bd69600bab30e13e631b848f207262e66ce40c1d24091

The string does not match the hash.
Success!
32
0x ac3e2db16ff747040371d960ee0c2b9b40831eb58d96e8211280dd2979fe7867

The string matches the hash.
Success!
64
0x 7c41b69aeb659046d3b81ab795b7f3960636f85123b7df8681ef2617e866232f

The string matches the hash.
Success!
请按任意键继续. . . |

```

图 12: project7 结果

## 10 Project11: impl sm2 with RFC6979

### 10.1 思路分析

#### 10.1.1 RFC6979

根据 *RFC6979*，最初的 *ECDSA* 概念中，每个签名都需要 256 位随机数据。这会带来一个问题，因为当使用相同的随机输入签署两个不同的消息（即比特币交易）时，会泄露私钥。*RFC6979* 建议使用  $HMAC - SHA256(private\_key, message)$  的输出来代替随机数据，从而消除了这种风险。

重点是  $k$  的生成。

前提：

$qlen$  是  $q$  的二进制字符串长度。

函数 *bits2int()* 将二进制字符串转换为 *int* 类型

函数 *int2octets()* 将 *int* 类型数据转换为多个 *8bit* 组成的字符串

函数 *bits2octets()* 将二进制字符串转换为多个 *8bit* 组成的字符串。

1 :  $Compute h_1 = H(m)$ ,  $m$  is the message.

2 :  $V = 0x010x010x010x010x01...0x01$

3 :  $K = 0x000x000x000x000x00...0x00$

4 :  $K = HMAC\_K(V || 0x00 || int2octets(x) || bits2octets(h_1))$ ,  $x$  is public key.

5 :  $V = HMAC\_K(V)$ .

6 :  $K = HMAC\_K(V || 0x01 || int2octets(x) || bits2octets(h_1))$ .

7 :  $V = HMAC\_K(V)$ .

8 :  $T = ""$ ,  $hlen$  是序列  $T$  的长度, 初始化为 0。

9 : *Loop* : *While*  $hlen < qlen$  *compute*  $V = HMAC\_K(V)$   $T = T || V$   $hlen = length(T)$ .

10 : *compute*  $K = bits2int(T)$ .

11 : *if*  $K \in [1, q - 1]$  *then* *we get*  $K$ .

12 : *if*  $K \notin [1, q - 1]$ .

13 : *Loop* : *while*  $k \geq q$   $K = HMAC\_K(V || 0x00)$   $V = HMAC\_K(V)$   $K = bits2int(T)$ .

14 : 循环结束后, 得到正确的  $K$ 。

### 10.1.2 具体想法

改变随机数  $k$  的生成方式, 由 SM3 生成  $k$ , 即  $k = SM3(d, message)$ 。其他的 SM2 实现方法依照之前的原理实现。

## 10.2 运行结果

SM2 加密函数与解密函数正确性验证。



```
明文: 123456
c1: c3c8e0116ca85562ebc4291c7ccdc4052396a0ce7777bcff0c3eab1aabcc85fb2cafd506c3a2167cdbf48ad357b569650348a8949c939a040723760f0f7cee7
c2: df1f4b71f815
c3: ddee9a3870a07a7971394118a2717b310a5ad59be122a7b5ed99921f3f22464d
解密结果为: 123456
```

图 13: SM2 加解密算法验证

SM2 签名算法正确性验证。

```
消息: 123456
签名r: a7b37c38add7540d27bb4ea309696681ab94120d8ef66e2338046b9940e7ccbc
签名s: d3571576852aaf355025421934a1953f4a57a009274a4c23d7e4c5758b31f7df
验证结果: True
```

图 14: SM2 签名算法验证

### 10.3 实验效率

SM2 加密函数与解密函数时间效率测量。

```
明文: 123456
c1: c3c8e0116ca85562ebc4291c7ccdc4052396a0ce7777bcff0c3eab1aabcc85fb2cafd506c3a2167cdbf48ad357b569650348a8949c939a040723760f0f7cee7
c2: df1f4b71f815
c3: ddee9a3870a07a7971394118a2717b310a5ad59be122a7b5ed99921f3f22464d
解密结果为: 123456
测量1000次运行时间为: 5.773189306259155
测量1次运行时间为: 0.005773189306259155
```

图 15: SM2 加解密算法效率

如上图所示，一次加密与解密耗时为：0.005773189306259155s。

SM2 签名算法时间效率测量。

```

消息: 123456
签名r: a7b37c38add7540d27bb4ea309696681ab94120d8ef66e2338046b9940e7ccbc
签名s: d3571576852aaf355025421934a1953f4a57a009274a4c23d7e4c5758b31f7df
验证结果: True
测量1000次运行时间为: 6.3390820026397705
测量1次运行时间为: 0.006339082002639771
请按任意键继续. . . |

```

图 16: SM2 签名算法效率

如上图所示，一次签名及验证耗时为：0.006339082002639771s。

## 11 Project12: verify the above pitfalls with proof-of-concept code

### 11.1 思路分析

本项目意在证明目前 SM2、ECDSA 签名算法应用过程中存在的几个安全性问题。分别是：

①: leaking k.

当我们获得了签名结果  $(r, s)$  以及参数  $k$  之后，由于  $s = ((1 + d_A)^{-1} \cdot (k - r \cdot d_A)) \bmod n$ ，以及  $s(1 + d_A) = (k - r \cdot d_A) \bmod n$ ，我们可以得到  $d_A = (s + r)^{-1} \cdot (k - s) \bmod n$ 。根据此式编写计算代码，即可计算出  $k$  来。

②: reusing k.

此问题针对两次不同的签名，但是两次签名选取了相同的  $k$  进行计算，此时我们可以根据签名结果计算出  $d_A$  来。我们已知  $s_1(1 + d_A) = (k - r_1 \cdot d_A) \bmod n$  以及  $s_2(1 + d_A) = (k - r_2 \cdot d_A) \bmod n$ ，由这两个式子便可得到  $d_A = \frac{s_2 - s_1}{s_1 - s_2 + r_1 - r_2} \bmod n$

③: reusing k by different users

此问题针对的情况是两个加密者使用了相同的  $k$ ，此时两个用户可以分别得到对方的密钥出来。由于  $(r, s)$  的签名过程分别是  $r = (\text{Hash}(Z \| M) + x) \bmod n$  与  $s =$

$((1 + d_B)^{-1} \cdot (k - r \cdot d)) \bmod n$ , 一般情况下签名结果是公开的, 而对于二人来说,  $k$  也相当于公开的, 因此二人可以使用公式  $d = \frac{k-s}{s+r} \bmod n$ , 来根据对方的签名结果来得到对方的公钥。

④: same d and k with ECDSA

根据 ECDSA 的签名过程, 我们能得到等式  $d \cdot r_1 = ks_1 - e_1 \bmod n$ , 同样的根据 SM 2 的签名过程, 我们有等式  $d \cdot (s_2 + r_2) = k - s_2 \bmod n$ , 二者结合进行计算后我们可以得到  $d = \frac{s_1 s_2 - e_1}{(r_1 - s_1 s_2 - s_1 r_2) \bmod n} \circ$

## 11.2 实现方式

```

1
2 #1,leaking k
3 def LeakingK():
4     r,s = Sm2Sign(m, ID, d)
5     da = gmpy2.invert(s + r,n)
6     da = da * (k - s)%n
7     return da
8
9 #2,reusing k
10 def ResusingK():
11     r1,s1 = Sm2Sign(Ma, ID, d)
12     r2,s2 = Sm2Sign(Mb, ID, d)
13     temp = s1-s2+r1-r2
14     da = gmpy2.invert(temp,n)
15     da = da * (s2-s1)%n
16
17 #3,reusing k by different users
18 def Compute(R, S):
19     R = int(R, 16)
20     S = int(S, 16)
21     D = k - S

```

```

22     D = D * gmpy2.invert(S + R, n) % n
23     return D
24     ## Alice的消息
25     M_1 = "abc"
26     ## Bob的消息
27     M_2 = "xyz"
28     ## Alice的ID
29     ALICE_ID = "ALICE"
30     ## Bob的ID
31     BOB_ID = "BOB"
32     ## Alice的私钥
33     d_alice = random.randint(1, n)
34     ## Bob的私钥
35     d_bob = random.randint(1, n)
36     r_alice, s_alice = sign(M_1, ALICE_ID, d_alice)
37     r_bob, s_bob = sign(M_2, BOB_ID, d_bob)
38     ## Alice计算Bob的私钥
39     d2 = Compute(r_bob, s_bob)
40     ## Bob计算Alice的私钥
41     d1 = Compute(r_alice, s_alice)
42
43     #4,same d and k with ECDSA
44     def SameDKECDSA():
45         r1,s1 = Sm2Sign(m, ID, d)
46         r2,s2 = ECDSA(m, ID, d)
47         temp = r1-s1*s2-s1*r1
48         da = gmpy2.invert(temp,n)
49         da = da*(s1*s2-e1)%n

```

### 11.3 运行结果

```

----- Alice与Bob的私钥 -----
Alice的私钥: 9510997258574332496346815017004658177586133076190438977351986250922891913726
Bob的私钥: 53071795680833194303560837337858074436280746529349170574469191384620245636353
----- Alice与Bob的签名 -----
Alice的签名: r: 9c71b5984ff66458897668aff87198905ee8395b733d3dd6d4ebf49e5e50fa10 s: b08fc6777692a70a88386007ae3a83dcdf39db14643a11a4fd943f2
Bob的签名: r: 5e3d00aa68e8eccc47ec38bfaf295dd3f335177aa34bc72ccb357a36958cd0ff s: c0d2c3538ac1bb8bfa71c43bb10824eb2a59068b288d665dbf8812e04
----- 计算得到二者的私钥 -----
Alice计算的Bob的私钥: 53071795680833194303560837337858074436280746529349170574469191384620245636353
Bob计算的Alice的私钥: 9510997258574332496346815017004658177586133076190438977351986250922891913726

Process finished with exit code 0

```

图 17: project9 结果

## 12 Project13: Implement the above ECMH scheme

### 12.1 思路分析

实现 *ECMH* 方案，如得到  $Hash(a)$ ，需要先将  $a$  经 *sm3* 哈希，然后作为  $x$  代入到  $y^2 = x^3 + ax + b$  的方程中求  $y$ 。这里需要使用二次剩余求出对应的  $y$ 。将初始点对应为  $(0,0)$ ，初始点与该点相加。即得到  $Hash(a)$ 。（椭圆曲线的点的加法）

若得到  $Hash(a, b)$ 。只需在  $Hash(a) + Hash(b)$ 。先将  $b$  经 *SM3* 哈希，然后作为  $x$  代入到  $y^2 = x^3 + ax + b$  的方程中求  $y$ 。然后与  $Hash(a)$  相加。

计算  $Hash(a, b, c) - Hash(c)$  的过程与相加类似，只不过椭圆曲线上的加法改成减法。

### 12.2 实现方式

```

1      # ECMH
2      def ECMH(data):
3          Infinty = Add(0, 0, 0, 0)
4          for item in data:
5              item = sm3(item) #都是字符串类型
6              item = int(item, 16)

```

```

7         item1 = (pow(item, 3) + a * item + b) % p
8         item_y = QR(item1, int(p))
9         Infinty = Add(Infinty[0], Infinty[1], item, item_y)
10    return Infinty
11
12    def ECMH_ADD(data1, data2):
13        data2 = sm3(data2[0])
14        data2 = int(data2, 16)
15        data2_x = (pow(data2, 3) + a * data2 + b) % p
16        data2_y = QR(data2_x, p)
17        result = Add(data1[0], data1[1], data2, data2_y)
18        return result
19
20    def ECMH_REMOVE(data1, data2):
21        data2 = sm3(data2[0])
22        data2 = int(data2, 16)
23        data2_x = (pow(data2, 3) + a * data2 + b) % p
24        data2_y = QR(data2_x, p)
25        result = Add(data1[0], data1[1], data2, p - data2_y)
26        return result
27
28    #示例
29    str1 = ['ab46546464']
30    str2 = ['ab46546464', 'ab46546464']
31    str3 = ['123456ac757645ef5465', 'a5459645646acd354563d']
32    str4 = ['a5459645646acd354563d', '123456ac757645ef5465']
33    str5 = ['123456ac757645ef5465', 'a5459645646acd354563d', '
34        ab46546464']
35
36    strx = ['ab46546464', '3265752a23434c']

```

```
36     stry = ['3265752a23434c', 'ab46546464']
37     result1 = ECMH(str1)
38     result2 = ECMH(str2)
39     result3 = ECMH(str3)
40     result4 = ECMH(str4)
41     result5 = ECMH(str5)
42     print("第一个字符串集: ", str1, '\n')
43     print("hash: ", result1, '\n')
44
45     print("第二个字符串集", str2, '\n')
46     print("hash: ", result2, '\n')
47
48     print("第三个字符串集", str3, '\n')
49     print("hash: ", result3, '\n')
50
51     print("第四个字符串集", str4, '\n')
52     print("hash: ", result4, '\n')
53
54     print("第五个字符串集", str5, '\n')
55     print("hash: ", result5, '\n')
56
57     if result1 != result2:
58         print("由第一个和第二个字符串集的结果可知, Hash{a}不等于Hash
59             {a, a}\n")
60
61     if result3 == result4:
62         print("由第三个和第四个字符串集的结果可知, Hash{a, b}等于
63             Hash{b, a}\n") #这个有问题
```

```

64     result6 = ECMH_ADD(result3, str1)
65     if result6 == result5:
66         print("Hash{a, b} + Hash{c}: ", result6, '\n')
67         print("Hash{a, b, c} = Hash{a, b} + Hash{c}: ", result5, '
        \n')
68         print("由前两步得出Hash{a, b, c} = Hash{a, b} + Hash{c}\n"
        )
69
70     result7 = ECMH_REMOVE(result5, str1)
71     if result7 == result3:
72         print("Hash{a, b, c} - Hash{c}: ", result7, '\n')
73         print("Hash{a, b} = Hash{a, b, c} - Hash{c}: ", result3, '
        \n')
74         print("由前两步得出Hash{a, b} = Hash{a, b, c} - Hash{c}\n"
        )

```

### 12.3 运行结果

实验结果如下。

```

Hash{a, b} + Hash{c}: [mpz(50011904358568899279028813338687177356313390772665418748836803736118202317106), mpz(10374600
6025012999008673214721056715634160611411067154025080934133493042839046)]

Hash{a, b, c} = Hash{a, b} + Hash{c}: [mpz(5001190435856889927902881333868717735631339077266541874883680373611820231710
6), mpz(103746006025012999008673214721056715634160611411067154025080934133493042839046)]

由前两步得出Hash{a, b, c} = Hash{a, b} + Hash{c}

Hash{a, b, c} - Hash{c}: [mpz(968676388168333436460634648576543442660218412910851208554995896191982295876), mpz(10906
9016995869833460958312559263049883113420405649545513694535970105725369404)]

Hash{a, b} = Hash{a, b, c} - Hash{c}: [mpz(96867638816833343646063464857654344266021841291085120855499589619198229587
6), mpz(109069016995869833460958312559263049883113420405649545513694535970105725369404)]

由前两步得出Hash{a, b} = Hash{a, b, c} - Hash{c}

请按任意键继续. . . |

```

图 18: ECMH 实验结果

可以看出：



$$\text{Hash}(a) \neq \text{Hash}(a,a)$$

$$\text{Hash}(a, b) = \text{Hash}(b, a)$$

```

Hash{a, b} + Hash{c}: [mpz(50011904358568899279028813338687177356313390772665418748836803736118202317106), mpz(10374600
6025012999008673214721056715634160611411067154025080934133493042839046)]

Hash{a, b, c} = Hash{a, b} + Hash{c}: [mpz(5001190435856889927902881333868717735631339077266541874883680373611820231710
6), mpz(103746006025012999008673214721056715634160611411067154025080934133493042839046)]

由前两步得出Hash{a, b, c} = Hash{a, b} + Hash{c}

Hash{a, b, c} - Hash{c}: [mpz(96867638816833343436460634648576543442660218412910851208554995896191982295876), mpz(10906
9016995869833460958312559263049883113420405649545513694535970105725369404)]

Hash{a, b} = Hash{a, b, c} - Hash{c}: [mpz(9686763881683334343646063464857654344266021841291085120855499589619198229587
6), mpz(109069016995869833460958312559263049883113420405649545513694535970105725369404)]

由前两步得出Hash{a, b} = Hash{a, b, c} - Hash{c}

请按任意键继续. . . |

```

图 19: ECMH 实验结果

可以看出：

$$\text{Hash}(a,b) + \text{Hash}(c) = \text{Hash}(a, b, c)$$

$$\text{Hash}(a, b, c) - \text{Hash}(c) = \text{Hash}(a,b)$$

## 13 Project14: Implement a PGP scheme with SM2

### 13.1 思路分析

#### 13.1.1 PGP 加密

第一步、sm2 密钥协商得到对称密钥。

第二步、sm2 加密对称密钥。

第三步、对称加密算法加密明文，密钥是协商后的对称密钥。

流程图如下。

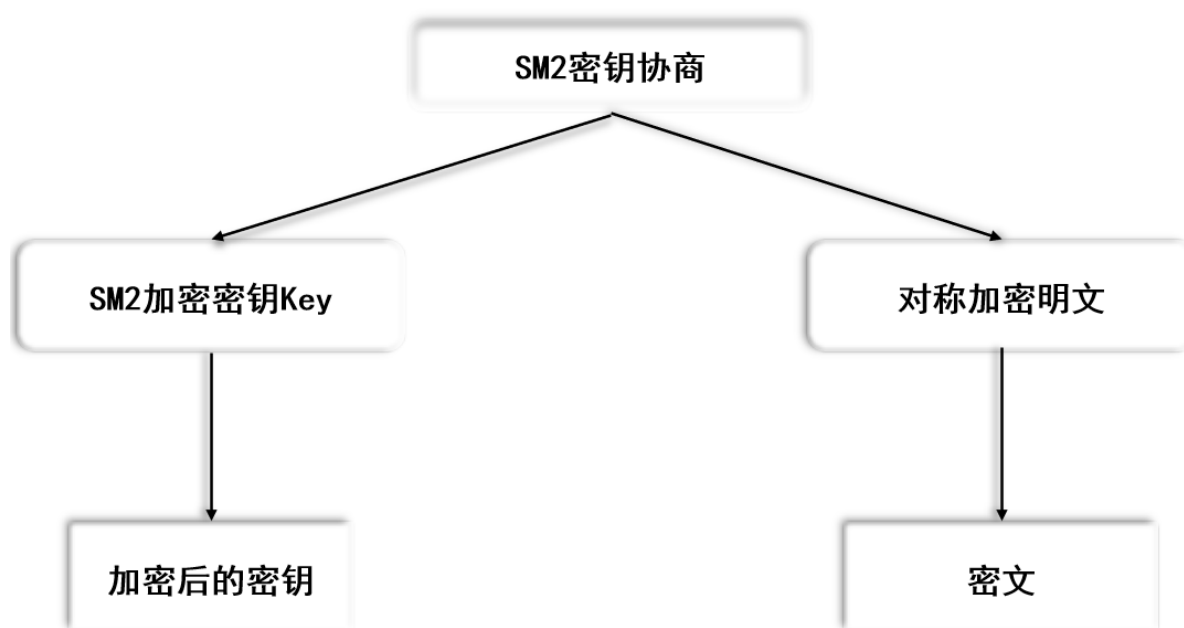


图 20: PGP 加密算法

在实现 PGP 加密过程中，首先利用 sm2 实现密钥协商，得到对称密钥 Key。其次，利用 sm2 加密函数对 Key 进行加密，加密公钥是  $P_k$ 。最后，对明文数据使用 AES 对称加密，密钥是 Key。

### 13.1.2 PGP 解密

第一步、sm2 解密对称密钥。

第二步、对称加密算法解密密文，密钥是 sm2 解密后的对称密钥。

流程图如下。

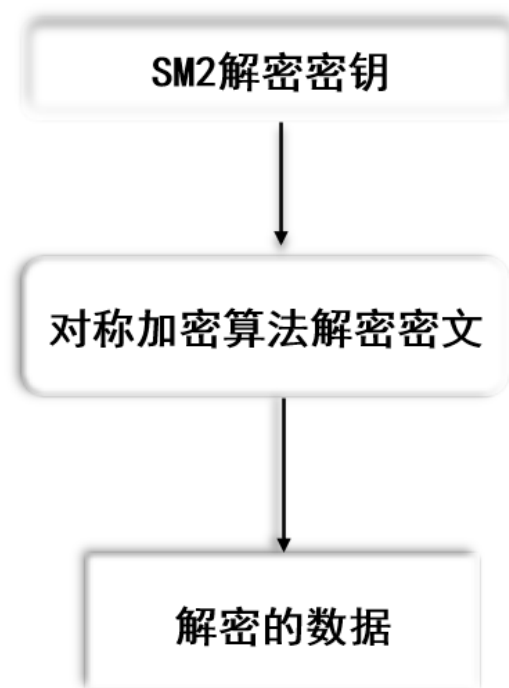


图 21: PGP 解密算法

在实现 PGP 解密过程中，首先利用私钥  $S_k$ ，使用 sm2 解密算法，得到对称密钥 Key。其次，对密文数据使用 AES 算法，密钥是 Key。最后，得到解密后的数据。

## 13.2 运行结果

```
d:\vscodefile\pythoncode - V: X + v
A协商的密钥: e8f4fc4578964e9e91799b8269676013
B协商的密钥: e8f4fc4578964e9e91799b8269676013
请按任意键继续. . . |
```

图 22: SM2 密钥协商正确性验证

PGP 方案加密函数与解密函数正确性验证。

```
明文是: THANKS
密文是: gAAAAABkxRCyYexZIGT7mPrOWUrFFQPJAmMqQ-JMbPCeubkZ-nXqhaIm2aY709m0w9TLNjI6I4q3hd5ysU519P942G7L6TZzJQ==
协商的密钥是: 99d54e422a438fadf9d4e14e97ff465b
加密后的密钥是: cff769e9efbf64dc8d793b799a21d6262bccca36cb43da0115cc6b6da50001e940ca7afb8ae493478933c6be8fe6d1e2f0884b
564640a1e2dfe976f38389e4b 0a5a52005755525053510c510c050356575c540c5d0106545a0f03050154025b 11d2ebe636101776e99f2e706d36c
84c735372f530283b500f6574ead9afa84d
解密后的数据是: THANKS
```

图 23: PGP 加解密算法验证

PGP 加密函数与解密函数时间效率测量。

```
明文是: THANKS
密文是: gAAAAABkxRCyYexZIGT7mPrOWUrFFQPJAmMqQ-JMbPCeubkZ-nXqhaIm2aY709m0w9TLNjI6I4q3hd5ysU519P942G7L6TZzJQ==
协商的密钥是: 99d54e422a438fadf9d4e14e97ff465b
加密后的密钥是: cff769e9efbf64dc8d793b799a21d6262bccca36cb43da0115cc6b6da50001e940ca7afb8ae493478933c6be8fe6d1e2f0884b
564640a1e2dfe976f38389e4b 0a5a52005755525053510c510c050356575c540c5d0106545a0f03050154025b 11d2ebe636101776e99f2e706d36c
84c735372f530283b500f6574ead9afa84d
解密后的数据是: THANKS
测量1000次运行时间为: 14.950654029846191
测量1次运行时间为: 0.014950654029846192
请按任意键继续. . . |
```

图 24: PGP 加解密算法效率

如上图所示，一次加密与解密耗时为：0.014950654029846192s。

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

### 14.1 思路分析

实现签名算法，使用 python 的 TCP 通信。通信双方为 A 和 B。其中，A 向 B 发送  $P_1$ ， $Q_1$  和  $e$ ，B 向 A 发送  $r, s_2, s_3$ 。

在代码 SM2\_2P\_SIGN\_A.py 中，实现 A 的功能。第一步，A 随机生成  $d_1$ ，然后计算  $P_1$ ，将  $P_1$  发给 B。第二步，A 在收到 B 的确认后计算  $Q_1$  和  $e$ ，并将其发给 B。第三步，A 在收到 B 发送的  $r, s_2, s_3$  后依次回复确认，计算出  $s$ 。

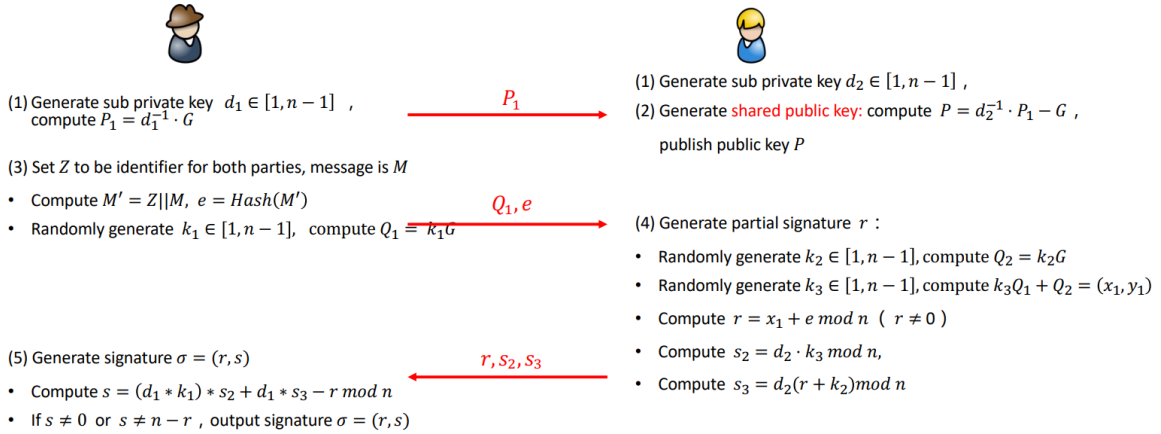
在代码 SM2\_2P\_SIGN\_B.py 中，实现 B 的功能。第一步，B 随机生成  $d_2$ 。第二步，收到 A 发送的  $P_1$  后回复确认，然后计算  $P$ 。第三步，收到 A 发送的  $Q_1$  和  $e$ ，依次回复确认，计算出  $r, s_2, s_3$ ，并将其发送给 A。

注意，由于通信质量较差，在最初的实现中，数据是连续发送的，如 A 将  $Q_1$  和  $e$  直接发给 B。但是这样容易出现数据丢失的问题，或者数据合并到一起发送，容易出错。所以改为了数据发送一个，对方收到后就发送一个回复，以确保数据传输准确。但是这样降低了效率。

流程图如下所示。

- Public key:  $P = [(d_1 d_2)^{-1} - 1]G$
- Private key:  $d = (d_1 d_2)^{-1} - 1$

- Signature
  - $(k_1 k_3 + k_2)G = (x_1, y_1)$
  - $r = (x_1 + e) \bmod n$
  - $s = (1 + d)^{-1} \cdot ((k_1 k_3 + k_2) - r \cdot d) \bmod n$



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

图 25: SM2\_2P\_SIGN 流程图

## 14.2 运行结果

实验结果如下。

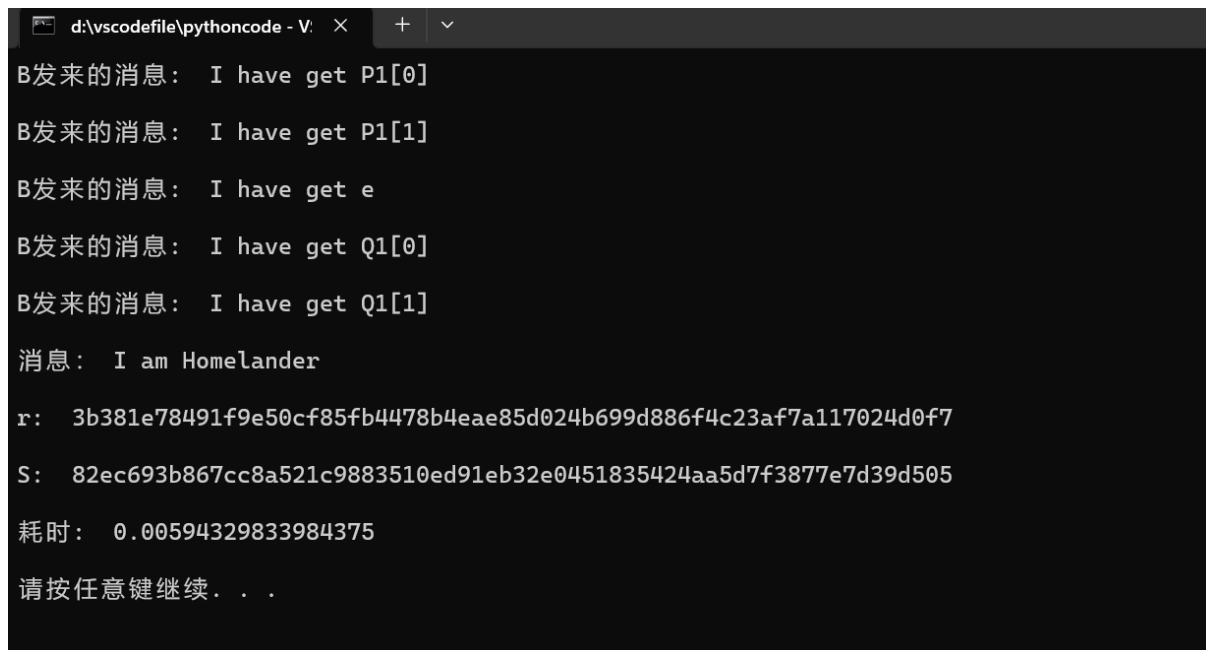
B 的结果展示。

```

d:\vscodefile\pythoncode - V:  ×  +  ∨
Listening on port: 50007
Connected by ('127.0.0.1', 52790)
A发来的消息: I get r
A发来的消息: I get s2
请按任意键继续. . .
  
```

图 26: B 通信实验结果

A 的结果展示。



```

d:\vscodefile\pythoncode - V:  X  +  v
B发来的消息: I have get P1[0]
B发来的消息: I have get P1[1]
B发来的消息: I have get e
B发来的消息: I have get Q1[0]
B发来的消息: I have get Q1[1]
消息: I am Homelander
r: 3b381e78491f9e50cf85fb4478b4eae85d024b699d886f4c23af7a117024d0f7
S: 82ec693b867cc8a521c9883510ed91eb32e0451835424aa5d7f3877e7d39d505
耗时: 0.00594329833984375
请按任意键继续. . .

```

图 27: A 通信实验结果

如上图所示，一次签名耗时为：0.00594329833984375s。

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

### 15.1 思路分析

实现签名算法，使用 python 的 TCP 通信。通信双方为 A 和 B。其中，A 向 B 发送  $T_1$ ，B 向 A 发送  $T_2$ 。

在代码 SM2\_2P\_DECRYPT\_A.py 中，实现 A 的功能。在该代码里实现加密，生成  $d_1$  和  $d_2$  以及公钥 P，使用 SM2 加密明文得到密文  $C_1$ 、 $C_2$ 、 $C_3$ 。并将  $d_2$  发送给 B。然后开始实现解密函数。

第一步，A 得到  $d_1$ ，然后计算  $T_1$ ，将  $T_1$  发给 B。第二步，A 收到 B 发送的  $T_2$ ，解密得到数据，具体方法和实验原理中的一致。

在代码 SM2\_2P\_DECRYPT\_B.py 中，实现 B 的功能。第一步，B 收到  $d_2$ 。第二步，收到 A 发送的  $T_1$  后回复确认，然后计算  $T_2$ ，并将其发送给 A。流程图如下所示。

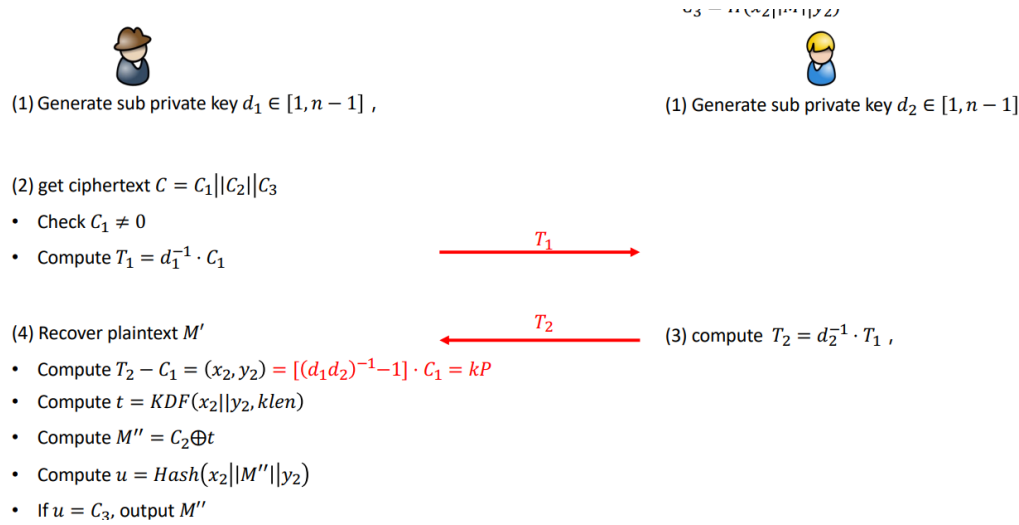


图 28: SM2\_2P\_DECRYPT 流程图

## 15.2 运行结果

实验结果如下。

B 的结果展示。

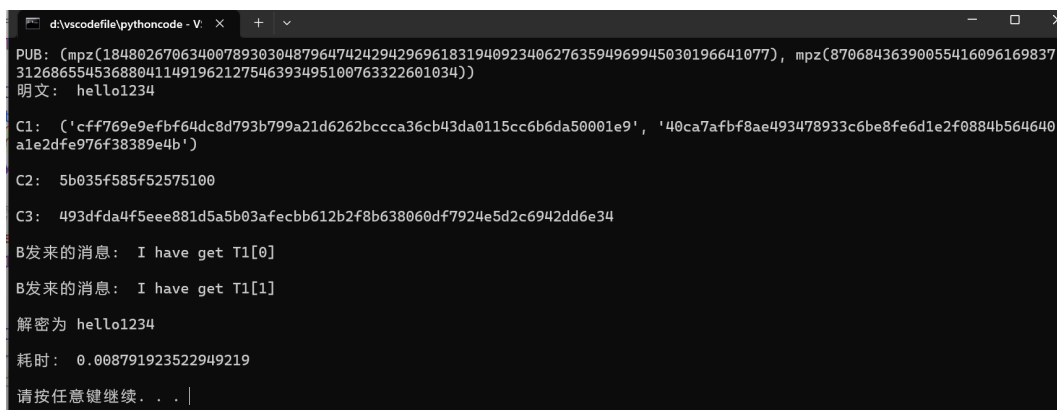
```
d:\vscodefile\pythoncode - V:  X  +  -  X
Listening on port: 50007
Connected by ('127.0.0.1', 57405)
T2: (mpz(110408819978378844851437120600532061584542499012021641882564633351328852934094), mpz(5075990188315473205176192
9132588875164579129257660389039855395527812219785494))

A发来的消息: I get T2[0]
A发来的消息: I get T2[0]
请按任意键继续. . . |
```

图 29: B 通信实验结果

A 的结果展示。





```
d:\vscode\pythoncode - V. x + v
PUB: (mpz(18480267063400789303048796474242942969618319409234062763594969945030196641077), mpz(87068436390055416096169837
312686554536880411491962127546393495100763322601034))
明文: hello1234
C1: ('cfff769e9efbf64dc8d793b799a21d6262bccca36cb43da0115cc6b6da50001e9', '40ca7afbfb8ae493478933c6be8fe6d1e2f0884b564640
a1e2dfe976f38389e4b')
C2: 5b035f585f52575100
C3: 493dfda4f5eee881d5a5b03afecbb612b2f8b638060df7924e5d2c6942dd6e34
B发来的消息: I have get T1[0]
B发来的消息: I have get T1[1]
解密为 hello1234
耗时: 0.008791923522949219
请按任意键继续. . .
```

图 30: A 通信实验结果

如上图所示，一次签名耗时为：0.008791923522949219s。

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

### 16.1 问题分析

Google:

Google 记住密码的插件实现是谷歌账号自带的功能，可以再不同的设备和平台上快速登录网站。其有以下特点：

1. 谷歌记住密码的插件使用 AES-128 加密算法来加密用户的密码。但加密密钥是由谷歌自身的服务器生成和管理的，在黑客攻击云端时，用户的密码就有泄露的风险。
2. 谷歌记住密码的插件将用户的账号密码保存在谷歌的云端服务器上。我们可以设置一个同步密码来保护这些密码。云端存储便于用户方便跨平台同步和使用，但是需要用户登录谷歌账号才能使用。
3. 在用户访问已保存密码的网站时，谷歌记住密码的插件会自动填充账号和密码。我们可以在浏览器的设置中选择是否开启这个功能，或者选择仅保存某几个网站的密码。

Firefox:

火狐是通过浏览器自带的密码管理器来实现记住密码的功能的，仅支持在本地使用。

1. 当用户在网页上输入账号和密码时，火狐会将用户选择保存的账号和密码存储在本地的密码管理文件中。

2. 火狐也有着自动填充的功能，当用户再次访问同一个网站时，火狐会自动从密码管理器中读取对应的账号和密码，并填充到网页的输入框中。用户也可以在浏览器的设置中查看和管理自己保存的密码，或者删除不需要的密码。

3. 火狐使用 AES-256 加密算法来加密用户的密码。如果用户没有设置主密码，那么加密密钥就是从用户的电脑上获取的，这意味着如果有人能够访问用户的电脑，就有可能破解用户的密码。

Firefox 和 Google 的记住密码插件的实现区别有以下几点：

1. 加密算法：Firefox 使用的是 AES-256 加密算法加密用户的密码，而 Google 则使用 AES-128 加密算法来加密。

2. 密钥存储：若用户没有设置主密码，Firefox 会选择从本地选取加密密钥。而 Google 的密钥是由谷歌云端生成的。

3. 是否自带：Firefox 的记住密码插件是浏览器自带的，而 Google 则是由谷歌账号提供记住密码服务。

Firefox 的优点是不依赖于第三方服务，更加隐私和自主，缺点是安全性取决于用户自己设置主密码和保护电脑。Google 的优点是方便跨平台同步和使用，缺点是安全性取决于 Google 自身和外部因素。用户可以根据自己的需求和偏好选择适合自己的记住密码插件。

## 17 Project22: research report on MPT

### 17.1 MPT 的优点

1. 完全确定性：具有相同键值对数据的 Merkle Patricia Tree 一定是完全相同的，包括每个节点的哈希值。

2. 高效性：插入、查找和删除操作的时间复杂度都是  $O(\log(n))$ ，其中  $n$  是键值对的数量。

3. 简单性：相比于其他基于比较的数据结构，如红黑树，Merkle Patricia Tree 更容易理解和实现。

4. 可验证性：可以在不知道所有数据的情况下验证某个键值对是否存在于 Merkle

Patricia Tree 中。

## 17.2 MPT 在以太坊中的用途

1. 存储账户状态：用户的账户的信息都被存储在一个被称作状态树的 Merkle Patricia Tree 中。状态树的根哈希值包含在区块头中。其实不止状态树的根哈希包含在区块头中，其他树的根哈希也被包含在了这个区域。

2. 存储交易回执：用户间每次进行一个交易执行后都会产生一个回执，回执中包含执行状态、日志等信息。Merkle Patricia Tree 存储了所有交易回执，这个 MPT 被称为回执树。

3. 存储交易数据：每个区块中的所有交易都被存储在了一个交易树种，于上述两种不同的是交易树是一个简单的 Merkle Tree。在区块链中 MPT 可以实现以下功能：

4. 存储键值对数据：MPT 可以存储包括但不限于账户的地址、余额、代码的键值对数据，并且这些数据是任意长的。它们构成了区块链的世界状态，也被称作每个节点需要维护的全局数据。

5. 生成根哈希：通过自底向上的方式，MPT 可以计算出每个节点的哈希值，最终得到一个根哈希，用来表示整棵树的状态。得到的根哈希可以作为区块链中每个区块的标识，也可以用来同步不同节点之间的数据。

6. 提供默克尔证明：默克尔证明可以用来在不知道所有数据的情况下，验证某个键值对是否存在于树中。该功能常见于轻节点中，因为它们不需要存储所有的数据，只需要存储一些必要的哈希值，就可以通过默克尔证明来验证数据的正确性。例如在区块链中进行验证交易时，可以在交易树中利用 Merkle 证明来在短时间内实现数据完整和正确的验证。