

项目分组

本组项目都由本人完成

姓名：齐霄桦

学号：202100460110

班级：网络空间安全 2021 级 1 班

Github 账户地址 <https://github.com/QixiaoH>

小组：Group3

配置及环境

所有 project 的运行环境均相同，如下图所示：

设备规格

复制

^

设备名称	Thin_White_Duke
处理器	13th Gen Intel(R) Core(TM) i9-13900HX 2.20 GHz
机带 RAM	16.0 GB (15.7 GB 可用)
设备 ID	14734478-59C5-4D10-AE19-2DD9F952B2B9
产品 ID	00342-30965-33420-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	2023/5/13
操作系统版本	22621.1992
体验	Windows Feature Experience Pack 1000.22644.1000.0

[Microsoft 服务协议](#)
[Microsoft 软件许可条款](#)

Project1

实验原理

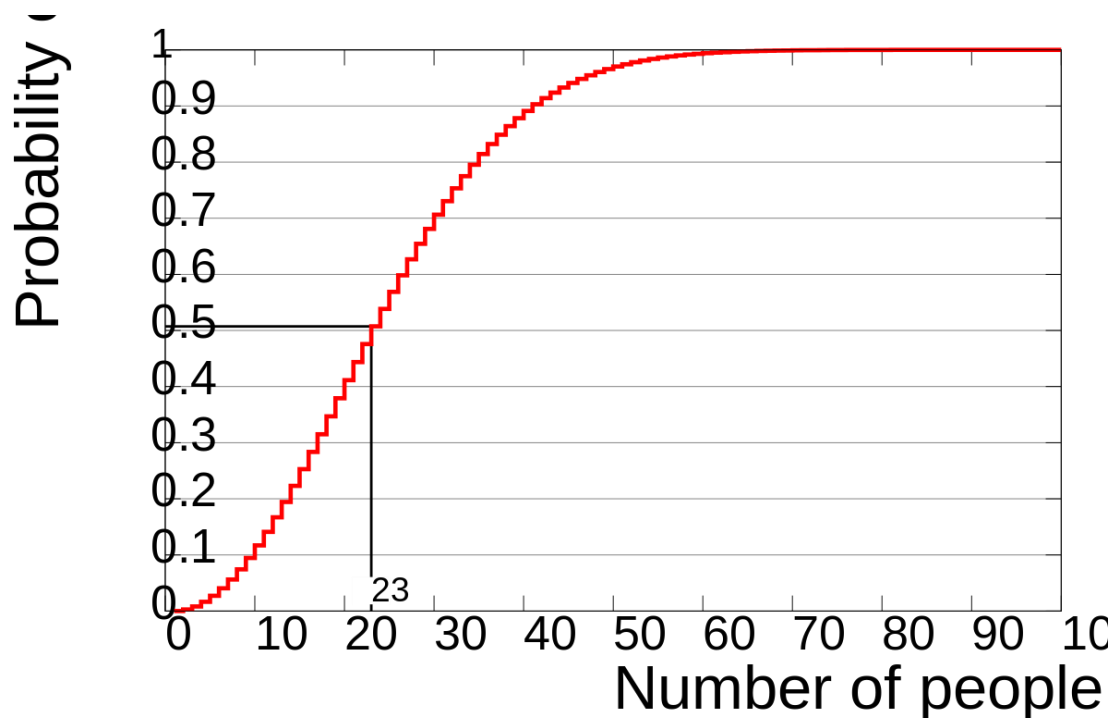
在哈希函数的上下文中，生日攻击利用生日悖论来找到具有相同哈希值的两个不同消息。哈希函数的输出空间通常很大，假设为 2^m 个可能的哈希值。当我们随机地选择一个消息，并将其哈希成一个特定的哈希值，我们可以认为这个过程相当于随机地选择一个元素并将其放入一个长度为 2^m 的容器中。

现在，假设我们选择了第二个消息，并希望找到与之前选择的具有相同哈希值的概率。在选择第一个消息时，哈希值是随机的，因此在 2^m 个可能的哈希值中，与第一个消息的哈希值相同的概率是 $1/2^m$ 。然后，当我们选择第二个消息时，我们希望其哈希值与第一个消息的哈希值相同。在 2^m 个可能的哈希值中，与第一个消息哈希值相同的概率仍然是 $1/2^m$ 。

现在我们可以用概率论的角度来看这个问题。我们需要计算第二个消息的哈希值与第一个消息的哈希值相同的概率，也就是两个消息的哈希值相同的概率。这可以用概率的补集来计算，即两个消息的哈希值不相同的概率。在第一个消息选择后，与之不相同的哈希值有 $(1 - 1/2^m)$ 的概率，因为只有一个特定的哈希值是相同的。然后，在选择第二个消息时，我们希望其哈希值与第一个消息的哈希值不同，所以也有 $(1 - 1/2^m)$ 的概率。因此，两个消息的哈希值不相同的概率是 $(1 - 1/2^m) * (1 - 1/2^m) = (1 - 1/2^m)^2$ 。

现在我们可以计算出两个消息的哈希值相同的概率，即它们产生碰撞的概率：

$$P(\text{碰撞}) = 1 - P(\text{不碰撞}) = 1 - (1 - 1/2^m)^2。$$



当选择的消息数量增加时，这个碰撞的概率会显著增加。例如，当我们选择约 $2^{(m/2)}$ 个消息时，碰撞的概率接近 50%。因此，利用生日攻击，我们可以在相对较短的时间内找到哈希函数的碰撞，而不需要遍历整个消息空间，这对于哈希函数的攻击非常有效。

代码分析

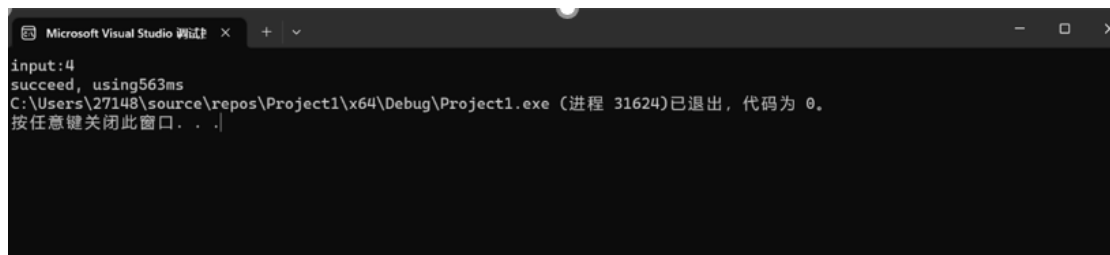
生日攻击的本质是穷举，所以破解速度应该较慢。我们只对结果的前 n 个比特进行破解，即随机选取两个不同的值作为 input，只要它们前 n 比特的值完全相同，就可以算作碰撞成功。并记录发生碰撞所需要的时间。如下面代码所示：

```
1. cout << "input:";
2.     int n;
3.     cin >> n;
4.     clock_t start, end;
5.     start = clock();
6.     int num = pow(2, n);
7.     start = clock();
8.     int M1 = rand() % (num + 1);
9.     int M2 = rand() % (num + 1);
10.
11.     while (SM3(2_16(10_2(M1))).subword(0, n / 4) != SM3(2_16(10_2
        (M2))).subword(0, n / 4))
```

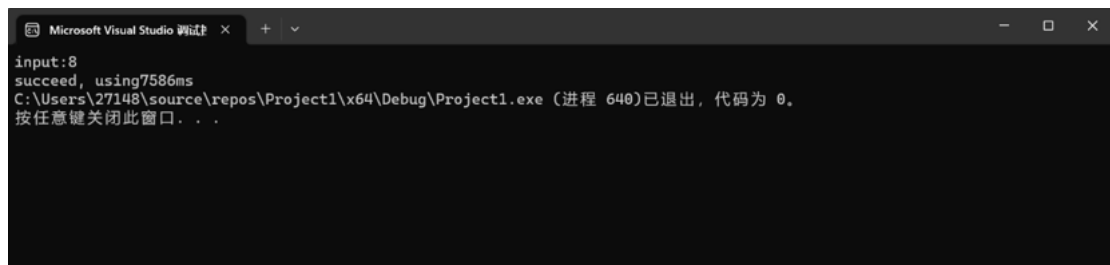
```
12.    {
13.        M1 = rand() % (num + 1);
14.        M2 = rand() % (num + 1);
15.    }
16.    end = clock();
17.    cout << "succeed, using " << (float)(end - start) * 1000 / CL
    O_PER_SEC << " ms";
18.    return 0;
```

实验结果

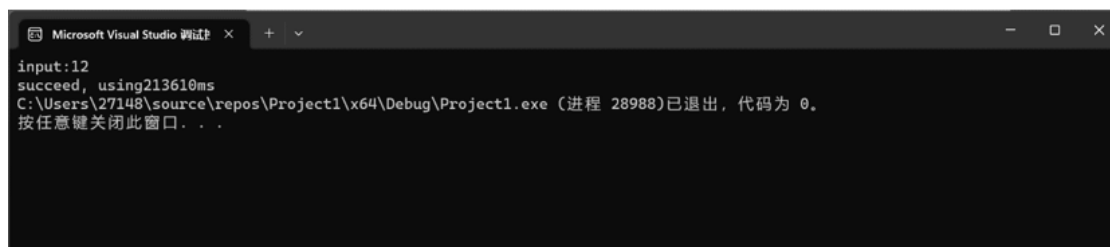
我们预期随着 n 的增大，发生碰撞所需要的时间应该成指数级增长。结果如下图所示：



```
Microsoft Visual Studio 调试
input:4
succeed, using 563ms
C:\Users\27148\source\repos\Project1\x64\Debug\Project1.exe (进程 31624)已退出，代码为 0。
按任意键关闭此窗口。 . . .
```



```
Microsoft Visual Studio 调试
input:8
succeed, using 7586ms
C:\Users\27148\source\repos\Project1\x64\Debug\Project1.exe (进程 640)已退出，代码为 0。
按任意键关闭此窗口。 . . .
```



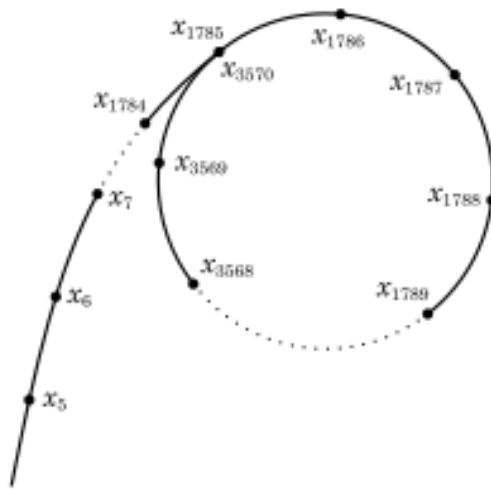
```
Microsoft Visual Studio 调试
input:12
succeed, using 213610ms
C:\Users\27148\source\repos\Project1\x64\Debug\Project1.exe (进程 28988)已退出，代码为 0。
按任意键关闭此窗口。 . . .
```

发现 $N=4$ 或 8 的时候碰撞速度很快，但是当 $N=12$ 的时候，需要花费大约 3.5 分钟才找到一组碰撞，并且随着 N 的增大，碰撞的时间成指数级增长。当 $N=16$ 时未发现碰撞。与我们的预期大致相同。

Project2

实验原理

Rho method 实验原理如下图：



实现 SM3 碰撞的步骤如下：

1. 选择一个适当的消息空间大小，例如，如果要找到前 n 比特的碰撞，那么选择一个大小为 2^n 的消息空间。
2. 在消息空间中随机选择两个消息 $M1$ 和 $M2$ 。
3. 计算 SM3 哈希值，即 $SM3(M1)$ 和 $SM3(M2)$ 。
4. 检查它们的哈希值的前 $n/4$ 比特是否相同，如果相同，则找到了一组前 n 比特的碰撞。

如果前 $n/4$ 比特不相同，则回到步骤 2，重新选择两个不同的消息，并再次计算哈希值，直到找到碰撞为止。

由于哈希函数的性质，即使在较小的消息空间中，找到碰撞也可能需要大量的计算和时间。

代码分析

我们利用 Rho 方法来实现对 SM3 进行 N=16 的碰撞，主要步骤为：

- 1.首先定义了 SM3 哈希函数中所需的一些辅助函数，如左移操作、FF 函数、GG 函数、P0 函数、P1 函数和 T 函数等。
- 2.然后定义了填充函数 padding，用于将输入的十六进制消息进行填充，使得消息满足 SM3 算法的要求。
- 3.接着定义了 block 函数，用于将填充后的消息切分成 128 位的块。
- 4.实现了 message_extension 函数，用于进行消息扩展，生成消息扩展字 W 和 W1。
- 5.然后定义了 message_compress 函数，用于对消息进行压缩，更新哈希状态 V。
- 6.最后，实现了 SM3 函数，该函数用于计算 SM3 哈希值。

在 Rho 攻击部分，定义了 rho_attack 函数，该函数执行了 Rho 方法的攻击步骤。

在主程序中，使用随机生成的 64 位整数作为输入，并执行 Rho 攻击，尝试找到哈希碰撞。

Rho 攻击的原理是在很小的随机输入空间中，通过哈希函数的输出结果找到循环节。当两个不同的输入产生相同的哈希结果时，即找到了碰撞。在这个代码实现中，我们生成随机的 64 位整数作为输入，然后执行 Rho 攻击尝试找到两个输入，使得它们产生相同的哈希结果。如果找到了碰撞，即 Rho 攻击成功，输出"Rho 攻击成功!"，并打印运行时间；如果未找到碰撞，则继续生成随机输入，直到找到碰撞为止。这里我们使用了一个小的随机输入空间（64 位整数），因此 Rho 攻击很快就能找到碰撞。

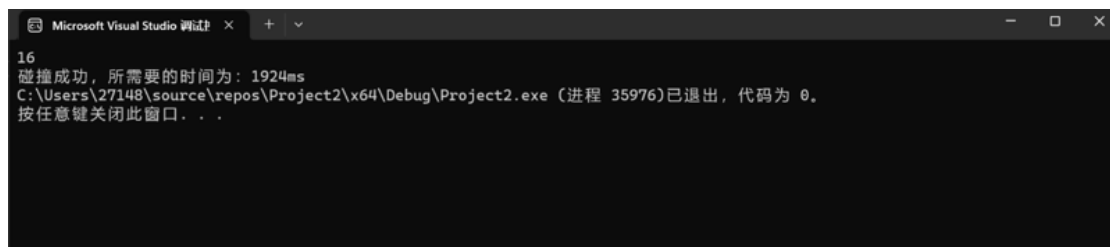
```
1. void rho_attack() {  
2.     vector<string> random_value; // 存储已生成的随机值  
3.     while (true) {  
4.         uint64_t r = rand() % (1ull << 64); // 生成 64 位的随机数  
5.         string m = padding(to_string(r)); // 对随机数进行填充
```

```

6.         vector<string> M = block(m); // 将填充后的消息切分成 128 位
           块
7.         vector<uint32_t> Mn = SM3(M); // 对消息进行哈希运算
8.         string tmp = "";
9.         for (int k = 0; k < IV_SIZE; k++) {
10.            char buffer[9];
11.            snprintf(buffer, sizeof(buffer), "%08x", Mn[k]);
12.            tmp += buffer; // 将哈希结果转换为十六进制字符串
13.        }
14.        string t = tmp.substr(0, 2); // 提取哈希结果的前两个字节
15.        if (find(random_value.begin(), random_value.end(), t) !=
            random_value.end()) {
16.            cout << "Rho 攻击成功!" << endl; // 攻击成功
17.            break;
18.        }
19.        else {
20.            random_value.push_back(t);
21.        }
22.    }
23.}

```

实验结果



可以发现，使用 Rho method 的速度远远大于生日攻击。与生日攻击的 8 比特在同一个数量级。

Project3

实验原理

攻击的目标是在已知哈希值 $H(M)$ 的情况下, 构造一个新的消息 M' , 使得哈希值 $H(M')$ 可以在不知道 M 的情况下计算出来。

SM3 哈希算法使用了 Merkle-Damgård 结构, 将消息划分为固定大小的块, 然后通过迭代的压缩函数来处理每个块, 最终得到哈希值。长度扩展攻击利用了 Merkle-Damgård 结构的特性, 即在已知 $H(M)$ 的情况下, 可以在不知道 M 的情况下扩展消息, 并计算出新消息的哈希值。

具体步骤如下:

已知哈希值 $H(M)$ 和消息 M 。

对消息 M 进行填充, 使得填充后的消息 M' 的长度满足 Merkle-Damgård 结构的要求。

在填充后的消息 M' 中添加需要扩展的内容, 例如附加数据 A , 形成 $M'' = M' + A$ 。

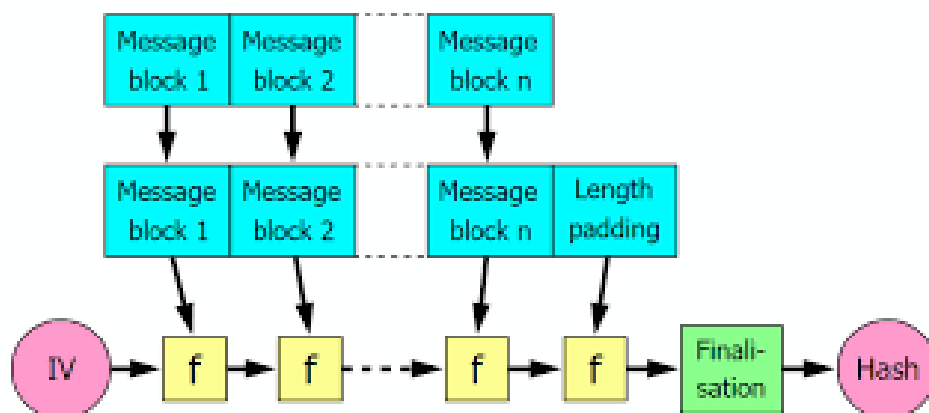
利用已知的 $H(M)$ 和 M' , 通过迭代压缩函数计算出 $H(M')$, 即哈希值 $H(M')$ 。注意, 这里的计算过程并不需要知道消息 M 的内容。

计算 $H(A)$, 即附加数据 A 的哈希值。

利用 $H(A)$ 和 $H(M')$, 通过迭代压缩函数计算出 $H(M'')$, 即哈希值 $H(M'')$ 。这里, 通过将附加数据 A 加入到填充后的消息 M' 中, 并计算 $H(M'')$, 实现了对消息 M'' 的哈希值的计算。

哈希值 $H(M'')$ 即是在不知道消息 M 的情况下, 计算出的新消息 M'' 的哈希值。

代码分析：



长度扩展攻击具有如下优点：

快速攻击速度： 长度扩展攻击是一种快速攻击方法，不需要通过穷举或其他复杂计算来找到有效的碰撞，可以在较短的时间内找到具有相同哈希值的两个不同消息。

避免碰撞计算： 在传统的碰撞攻击中，需要对两个不同的消息进行大量的哈希计算，以找到相同的哈希值，而长度扩展攻击避免了这种计算，直接构造了具有相同哈希值的消息。

因此在攻击的效率方面会比碰撞攻击更高。

具体消息扩展函数如下图所示：

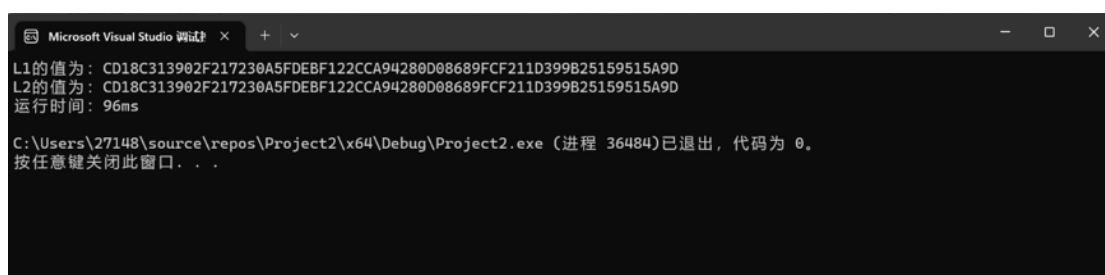
```
1. // 消息扩展
2. vector<uint32_t> message_extension(const vector<string>& M, int n
   ) {
3.     vector<uint32_t> W;
4.     vector<uint32_t> W1;
5.     for (int j = 0; j < 16; j++) {
6.         uint32_t x;
7.         sscanf(M[n].substr(32 * j, 32).c_str(), "%x", &x);
8.         W.push_back(x); // 将块中的十六进制值转换为整数并存储到W中
9.     }
10.    for (int j = 16; j < 68; j++) {
11.        W.push_back(P1(W[j - 16] ^ W[j - 9] ^ leftshift(W[j - 3],
            15)) ^ leftshift(W[j - 13], 7) ^ W[j - 6]); // 进行消息扩展
```

```

12.     }
13.     for (int j = 0; j < 64; j++) {
14.         W1.push_back(W[j] ^ W[j + 4]); // 生成W1
15.     }
16.     return W1;
17. }

```

运行结果：



运行时间仅仅需要 96ms,比仅适用 Rho 和生日攻击算法快得多。

Project4

实验原理：

主要通过一下方法进行加速：

1.使用位运算代替乘法：在 SM3 的实现中，我们使用位运算来进行循环左移操作，而不是使用乘法。这样做可以加快计算速度，特别是对于 32 位的整数类型。

```

1.  for (size_t i = 0; i < paddedMessage.size(); i += 64) {
2.      for (int j = 0; j < 16; j++) {
3.          W[j] = (static_cast<uint32_t>(paddedMessage[i + j * 4
4.              + 0])) << 24) |
5.              (static_cast<uint32_t>(paddedMessage[i + j * 4
6.              + 1])) << 16) |
7.              (static_cast<uint32_t>(paddedMessage[i + j * 4
8.              + 2])) << 8) |
9.              (static_cast<uint32_t>(paddedMessage[i + j * 4
10.             + 3])) << 0);

```

```

6.             (static_cast<uint32_t>(paddedMessage[i + j * 4
    + 3])));
7.         }

```

2.使用 std::bitset 代替字符串操作：在消息填充过程中，我们使用了 std::bitset 来生成 64 位的消息长度，并将其转换为字符串。这样可以避免字符串操作，提高了填充过程的效率。

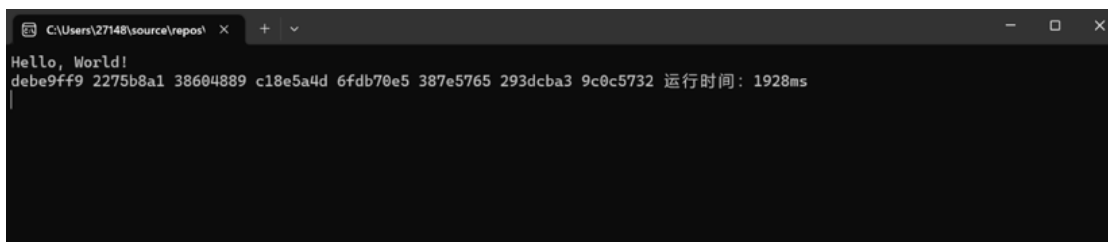
```

1. std::string SM3(const std::string& message) {
2.     uint32_t A = IV[0], B = IV[1], C = IV[2], D = IV[3], E = IV[4
    ], F = IV[5], G = IV[6], H = IV[7];
3.     std::vector<uint32_t> W(68), W1(64);
4.
5.     // Message padding
6.     std::string paddedMessage = message;
7.     paddedMessage += '\x80';
8.     while ((paddedMessage.size() * 8) % 512 != 448)
9.         paddedMessage += '\x00';
10.
11.    uint64_t messageLength = message.size() * 8;
12.    paddedMessage += std::bitset<64>(messageLength).to_string();

```

实验结果

未加速的运行结果：

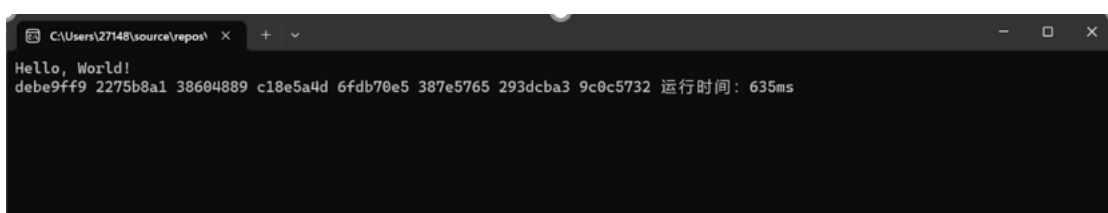


```

C:\Users\27148\source\repos\ X + v
Hello, World!
debe9ff9 2275b8a1 38604889 c18e5a4d 6fdb70e5 387e5765 293dcba3 9c0c5732 运行时间: 1928ms

```

进行加速后的运行结果：



```

C:\Users\27148\source\repos\ X + v
Hello, World!
debe9ff9 2275b8a1 38604889 c18e5a4d 6fdb70e5 387e5765 293dcba3 9c0c5732 运行时间: 635ms

```

加速前	1928ms
加速后	635ms

加速了约为 3 倍。

Project5

实验原理

Merkle Tree 是一种数据结构，她的生成步骤主要包括：

数据块：将待存储或验证的数据划分为较小的固定大小的数据块（通常是二进制数据）。

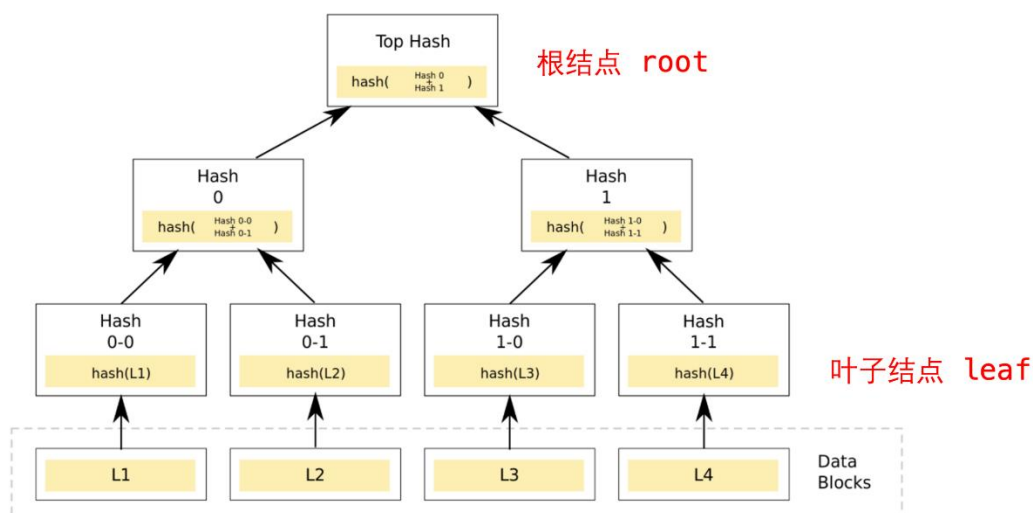
哈希计算：对每个数据块进行哈希计算。哈希函数（如 SHA-256）将数据块转换为固定长度的哈希值，通常是一个定长的字符串。

构建叶节点：每个数据块对应一个叶节点，其中叶节点的值对应数据块的哈希值。

合并节点：将相邻的叶节点成对地组合，并对每一对节点的值进行哈希计算，得到新的哈希值。这样，每个合并节点都有两个子节点。

重复步骤 4：不断重复上述步骤，直到最后只剩下一个根节点，即默克尔树的根。

最终形成的如下图的数据结构：



代码分析

在哈希函数的选择上，我们选取 SHA-256 作为哈希函数，并引入代码库：

```

1. #include <iostream>
2. #include <vector>
3. #include <string>
4. #include <openssl/sha.h> // For SHA-256 hash function
5.
6. // Function to compute SHA-256 hash of a string
7. std::string computeSHA256Hash(const std::string& data) {
8.     unsigned char hash[SHA256_DIGEST_LENGTH];
9.     SHA256_CTX sha256;
10.    SHA256_Init(&sha256);
11.    SHA256_Update(&sha256, data.c_str(), data.size());
12.    SHA256_Final(hash, &sha256);
13.
14.    std::string hashStr;
15.    for (int i = 0; i < SHA256_DIGEST_LENGTH; ++i) {
16.        hashStr += hash[i];
17.    }
18.    return hashStr;
19.}

```

接下来我们定义 Merkle Tree 的节点类型：

```

1. struct MerkleNode {
2.     std::string hash;

```

```

3.     MerkleNode* left;
4.     MerkleNode* right;
5.
6.     MerkleNode(const std::string& data) : hash(data), left(nullptr), right(nullptr) {}
7. };

```

然后，我们首先创建一个叶节点的列表 `leaves`，并为每个数据块创建一个叶节点，并计算每个叶节点的哈希值。接下来，我们在循环中将叶节点两两组合，计算它们的父节点的哈希值，并创建新的父节点。我们不断重复这个过程，直到最后只剩下一个根节点，即默克尔树的根。

```

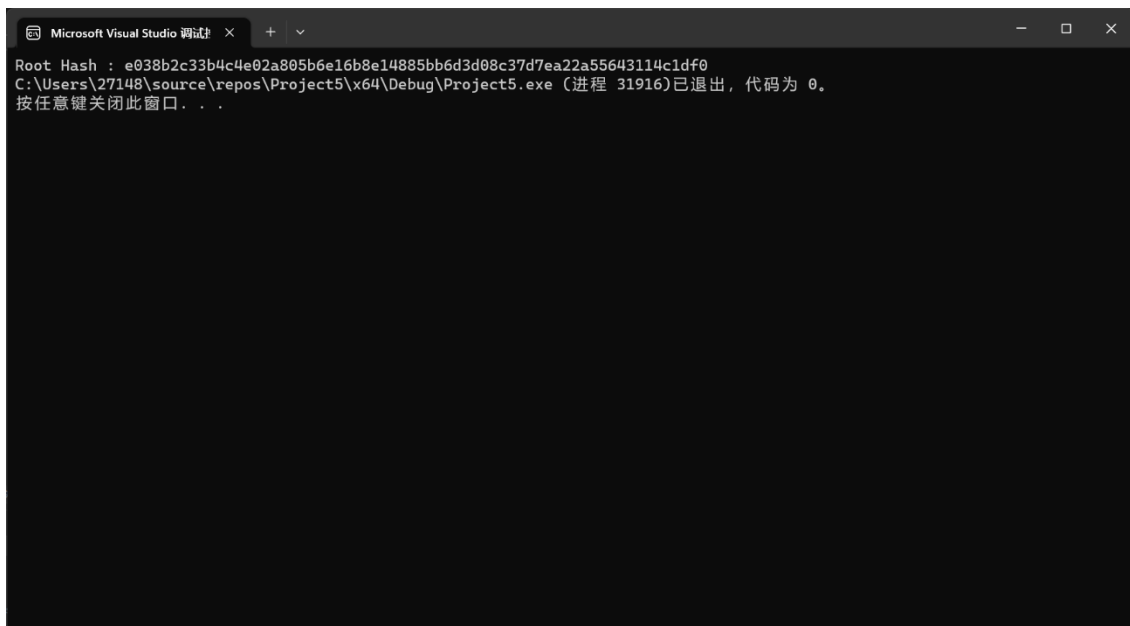
1. #include <vector>
2.
3. MerkleNode* buildMerkleTree(const std::vector<std::string>& data)
4. {
5.
6.     // Create leaf nodes for each data element and compute their hashes
7.     for (const std::string& str : data) {
8.         leaves.push_back(new MerkleNode(computeSHA256Hash(str)));
9.     }
10.
11.    if (leaves.empty()) {
12.        return nullptr;
13.    }
14.
15.    // Construct the tree by repeatedly combining pairs of nodes
16.    while (leaves.size() > 1) {
17.        std::vector<MerkleNode*> parents;
18.        for (size_t i = 0; i < leaves.size(); i += 2) {
19.            MerkleNode* leftChild = leaves[i];
20.            MerkleNode* rightChild = (i + 1 < leaves.size()) ? leaves[i + 1] : nullptr;
21.            std::string combinedData = leftChild->hash + (rightChild ? rightChild->hash : "");
22.            MerkleNode* parent = new MerkleNode(computeSHA256Hash(combinedData));
23.            parent->left = leftChild;
24.            parent->right = rightChild;
25.            parents.push_back(parent);
26.        }

```

```
27.     leaves = parents;
28. }
29.
30.     return leaves[0];
31. }
```

实验结果

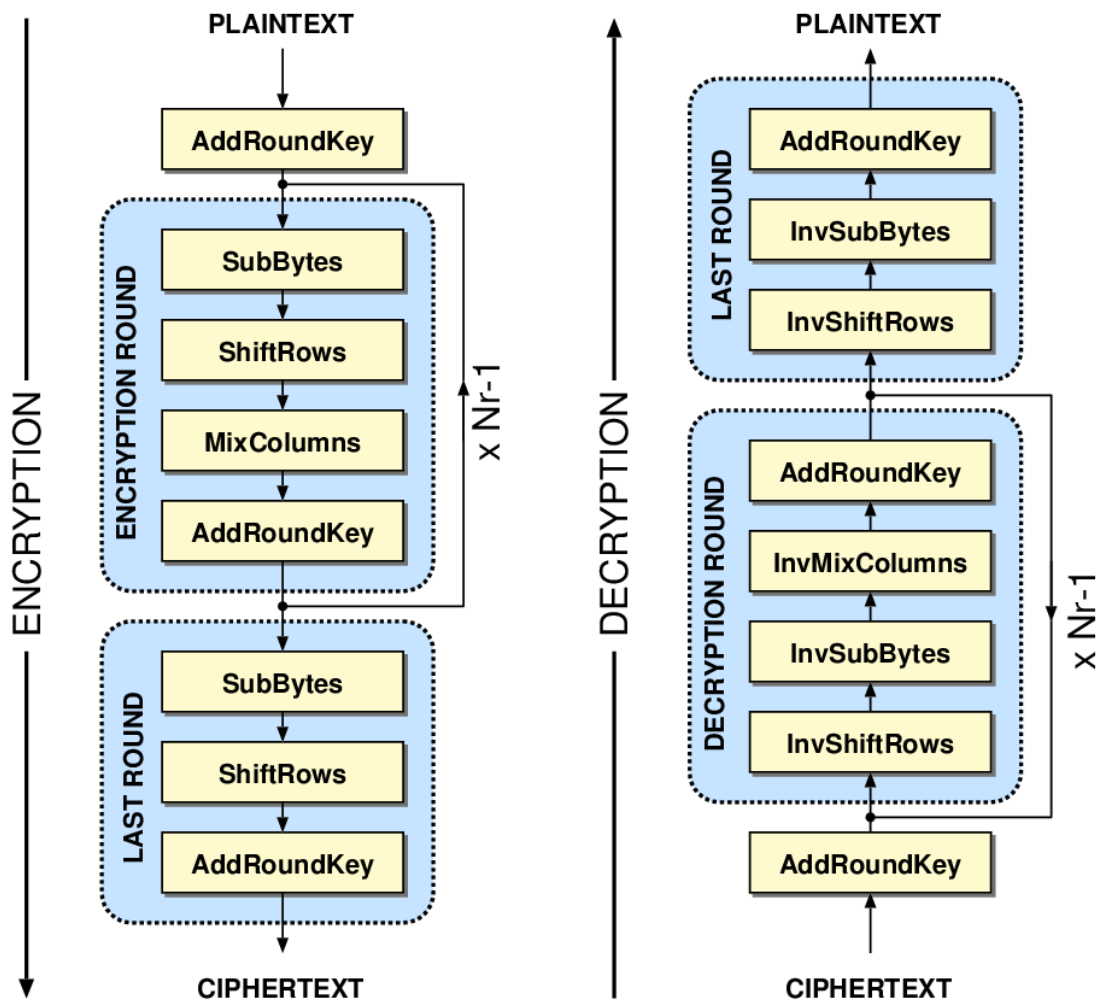
我建立了 1000 个节点，并输出了根节点。



```
Microsoft Visual Studio 调试  X + -
Root Hash : e038b2c33b4c4e02a805b6e16b8e14885bb6d3d08c37d7ea22a55643114c1df0
C:\Users\27148\source\repos\Project5\x64\Debug\Project5.exe (进程 31916) 已退出，代码为 0。
按任意键关闭此窗口。 . . .
```

Project9

实验原理及安全性分析



Key Expansion: 在 AES 算法中，首先需要对输入的密钥进行扩展，以生成一系列轮密钥，用于后续的加密和解密过程。

Initial Round: AES 的加密和解密都始于初始轮（Initial Round）。在初始轮中，输入数据（明文或密文）与第一个轮密钥进行 XOR 运算。

Rounds: AES 的加密和解密都包含多轮（10 轮、12 轮或 14 轮，取决于密钥长度）。在每一轮中，有四个基本步骤：SubBytes、ShiftRows、MixColumns 和 AddRoundKey。

SubBytes: 将每个字节替换为 S 盒中对应的值，S 盒是一个预定义的 256 字节的替换表。这个步骤引入了非线性性，增加了 AES 的安全性。

ShiftRows: 对状态矩阵中的行进行循环位移，使得每一列的字节都发生位移，这提供了一定的扩散效果。

MixColumns: 对状态矩阵中的每一列进行线性变换，通过乘以一个特定的矩阵来混淆数据，这增强了 AES 的复杂性。

AddRoundKey: 将状态矩阵的每个字节与当前轮的轮密钥进行 XOR 运算。

Final Round: 在最后一轮中，不执行 MixColumns 步骤，只进行 SubBytes、ShiftRows 和 AddRoundKey。

AES 加密方案中唯一的非线性部件是 S 盒，是抵抗线性分析的唯一部件。

代码分析

AES 的轮函数有 4 个：分别是与轮密钥异或，S 盒，行变换，列混淆，下面实现这四个函数：

```
1. void SubBytes(unsigned char* state)
2. {
3.     for (int i = 0; i < 16; i++)
4.     {
5.         state[i] = S_box[state[i]];
6.     }
7. void InvSubBytes(unsigned char* state)
8. {
9.     for (int i = 0; i < 16; i++)
10.    {
11.        state[i] = Contrary_S_box[state[i]];
12.    }
13. void ShiftRows(unsigned char* state)
14. {
15.     unsigned char tmp[16];
16.     tmp[0] = state[0];
17.     tmp[1] = state[5];
18.     tmp[2] = state[10];
19.     tmp[3] = state[15];
```

```
20. tmp[4] = state[4];
21. tmp[5] = state[9];
22. tmp[6] = state[14];
23. tmp[7] = state[3];
24. tmp[8] = state[8];
25. tmp[9] = state[13];
26. tmp[10] = state[2];
27. tmp[11] = state[7];
28. tmp[12] = state[12];
29. tmp[13] = state[1];
30. tmp[14] = state[6];
31. tmp[15] = state[11];
32. for (int i = 0; i < 16; i++)
33. {
34.     state[i] = tmp[i];
35. }
```

实验结果

The screenshot shows the Microsoft Visual Studio interface. The title bar at the top reads "Microsoft Visual Studio 调试". Below it, there are three tabs: one active tab labeled "调试" and two inactive tabs labeled "+" and "-". The main area displays the output of a program running in the debugger. The text shown is:
`明文为: 00000000000000000000702100460110`
`密钥为: 000000000000000000000202100460110`
`d756442b888397f567a00605d4e7bc14`
`花费时间为0.01s`
`解密成功!`
`C:\Users\27148\source\repos\Project5\x64\Debug\Project5.exe (进程 22848)已退出, 代码为 0.`
`按任意键关闭此窗口...`

最终明文 202100460110 在加密后又经过解密, 实现解密成功, 实现时间为 2.5s。

与密码库中的函数进行比较，结果为：

密码库的 AES	0.16s
自己编写的 AES	2.5s

实验原理

SM4 是中国的商用密码方案, 其需要经过 32 轮的加密, 分组大小密钥长度均为 128bit。
SM4 的加密和解密函数相同, 所以在实现的时候不需要额外设计解密函数。其加解密步骤主要为:

输入: 128 位明文块 M 和 128 位密钥 K 。

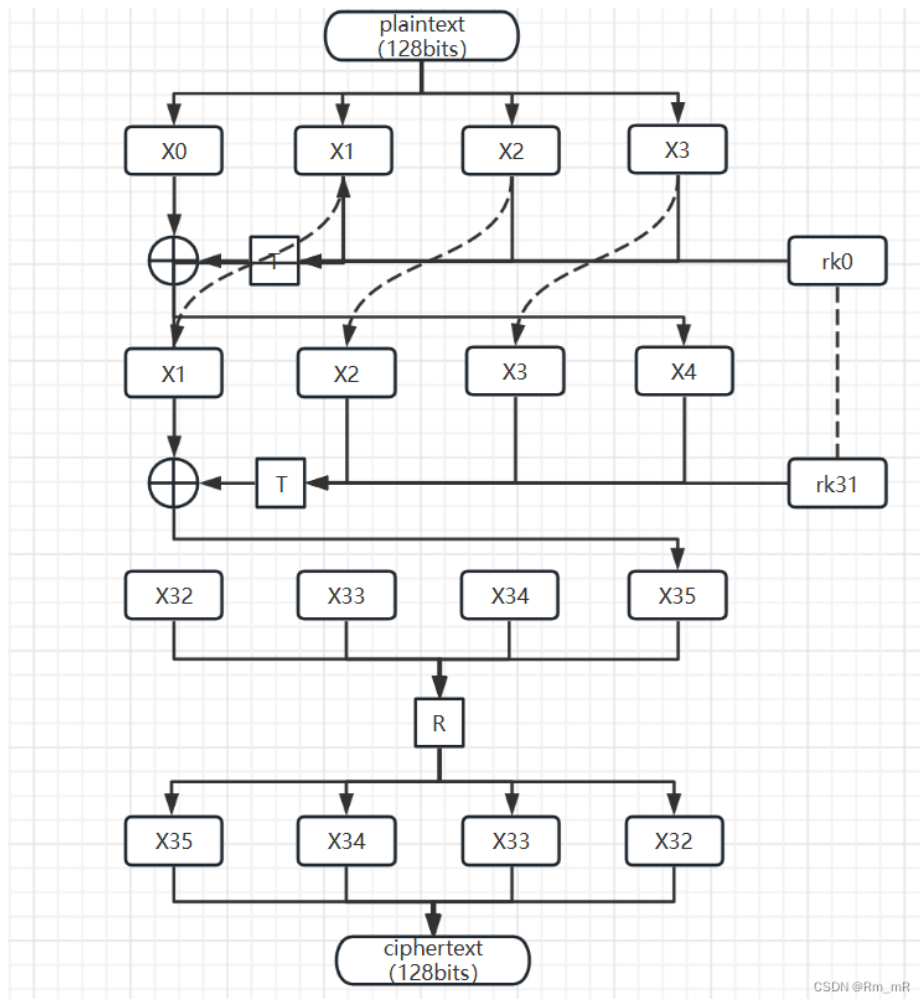
密钥扩展: 通过密钥扩展算法, 将 128 位密钥扩展为 32 轮子密钥 W_0, W_1, \dots, W_{31} 。

初始轮: 将明文块与第一个子密钥 W_0 进行异或运算。

32 轮加密: 重复 32 轮的加密过程, 每一轮包括以下四个步骤:

- 1. SubBytes:** 将明文块中的每个字节使用 S-box 进行替换。
- 2. ShiftRows:** 将明文块中的行进行循环位移。
- 3. MixColumns:** 对明文块中的列进行混淆操作。
- 4. AddRoundKey:** 将明文块与当前轮的子密钥进行异或运算。

输出: 在最后一轮加密后, 得到加密后的密文块。



SM4 解密步骤：

SM4 的解密过程与加密过程基本相同，但是需要使用子密钥的逆序，并且 SubBytes 和 ShiftRows 步骤使用的是逆变换。解密的最后一轮不包括 MixColumns 步骤。

由此可见，SM4 的安全性依赖于：Substitution Box (S-box)：SM4 使用一个预定义的 256 字节的 S-box 替换表，将输入的 8 位字节映射为 8 位字节输出，增加了非线性性。

Linear Transformation: SM4 进行一系列线性变换，包括移位和异或操作，以增加算法的扩散性。

Key Expansion: 在加密过程开始之前，需要对输入的 128 位密钥进行扩展，生成 32 轮子密钥，用于后续的加密和解密过程。

代码分析

主要对轮函数进行分析，4 个轮函数分别为：SubBytes 2. ShiftRows 3. MixColumns 4. AddRoundKey。

1. SubBytes

```
1. // SubBytes 函数
2. void SubBytes(unsigned char state[4][4]) {
3.     for (int i = 0; i < 4; i++) {
4.         for (int j = 0; j < 4; j++) {
5.             unsigned char b = state[i][j];
6.             unsigned char row = b >> 4; // 取得高4 位作为行索引
7.             unsigned char col = b & 0xF; // 取得低4 位作为列索引
8.             state[i][j] = S_box[row][col]; // 使用S-box 查找替换值
9.         }
10.    }
11.}
```

2. ShiftRows

```
1. // ShiftRows 函数
2. void ShiftRows(unsigned char state[4][4]) {
3.     for (int i = 1; i < 4; i++) {
4.         int shiftAmount = i;
5.         for (int j = 0; j < shiftAmount; j++) {
6.             unsigned char tmp = state[i][0];
7.             for (int k = 0; k < 3; k++) {
8.                 state[i][k] = state[i][k + 1];
9.             }
10.            state[i][3] = tmp;
11.        }
12.    }
13.}
```

3. MixColumns

```

1. // MixColumns 函数
2. void MixColumns(unsigned char state[4][4]) {
3.     for (int i = 0; i < 4; i++) {
4.         unsigned char col[4];
5.         for (int j = 0; j < 4; j++) {
6.             col[j] = state[j][i];
7.         }
8.         state[0][i] = xtime(col[0] ^ col[1]) ^ col[1] ^ col[2] ^
col[3]; // 2 倍乘法运算
9.         state[1][i] = xtime(col[1] ^ col[2]) ^ col[0] ^ col[2] ^
col[3];
10.        state[2][i] = xtime(col[2] ^ col[3]) ^ col[0] ^ col[1] ^
col[3];
11.        state[3][i] = xtime(col[0] ^ col[2] ^ col[3]) ^ col[0] ^
col[1] ^ col[2];
12.    }
13.}

```

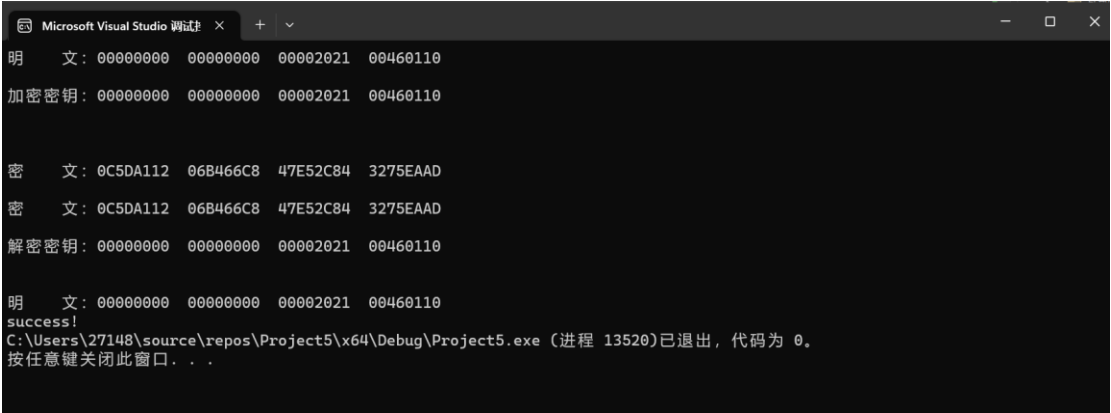
4.AddRoundKey

```

1. // AddRoundKey 函数
2. void AddRoundKey(unsigned char state[4][4], unsigned char roundKe
y[4][4]) {
3.     for (int i = 0; i < 4; i++) {
4.         for (int j = 0; j < 4; j++) {
5.             state[i][j] ^= roundKey[i][j]; // 对状态矩阵与轮密钥进
行异或
6.         }
7.     }
8. }

```

实验结果



```
Microsoft Visual Studio 调试  x + -
明 文: 00000000 00000000 00002021 00460110
加密密钥: 00000000 00000000 00002021 00460110

密 文: 0C5DA112 06B466C8 47E52C84 3275EAD
密 文: 0C5DA112 06B466C8 47E52C84 3275EAD
解密密钥: 00000000 00000000 00002021 00460110

明 文: 00000000 00000000 00002021 00460110
success!
C:\Users\27148\source\repos\Project5\x64\Debug\Project5.exe (进程 13520)已退出, 代码为 0。
按任意键关闭此窗口...
```

最终解密成功!

Project10

报告详见:

<https://github.com/QixiaoH/homework-group3/tree/main/Project10>

project 11

SM2 概述

M2 算法基于椭圆曲线密码学，它利用椭圆曲线上的点运算实现加密和签名操作。相比传统的 RSA 算法，SM2 在相同的安全强度下，具有更短的密钥长度，计算速度更快，适用于资源有限的场景，例如移动设备和物联网设备。

SM2 算法具有以下主要特点：

安全性:SM2 采用了一系列安全性措施, 确保其抵御常见的密码学攻击, 如大整数分解、离散对数等。

签名与加密:SM2 既可用于数字签名来验证消息的真实性和完整性, 也可用于数据加密, 确保数据在传输过程中的机密性。

签名速度:相比传统 RSA 算法, SM2 签名速度更快, 特别是在移动设备等资源受限的环境下表现更优秀。

密钥长度:在相同的安全级别下, SM2 所需的密钥长度明显短于 RSA 算法, 这有助于减少密钥管理的复杂性。

实验原理

SM2 加密和解密是非对称加密的过程, 使用公钥和私钥进行加密和解密操作。

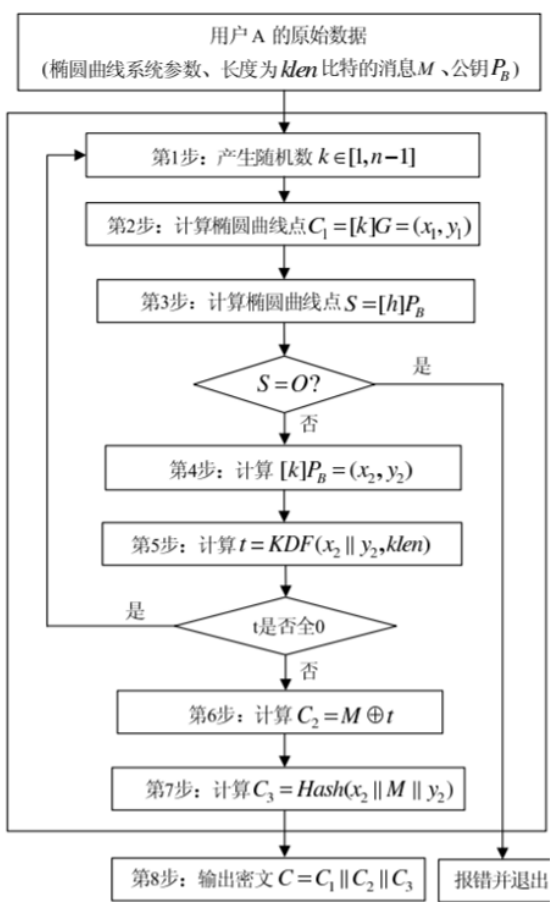


图1 加密算法流程

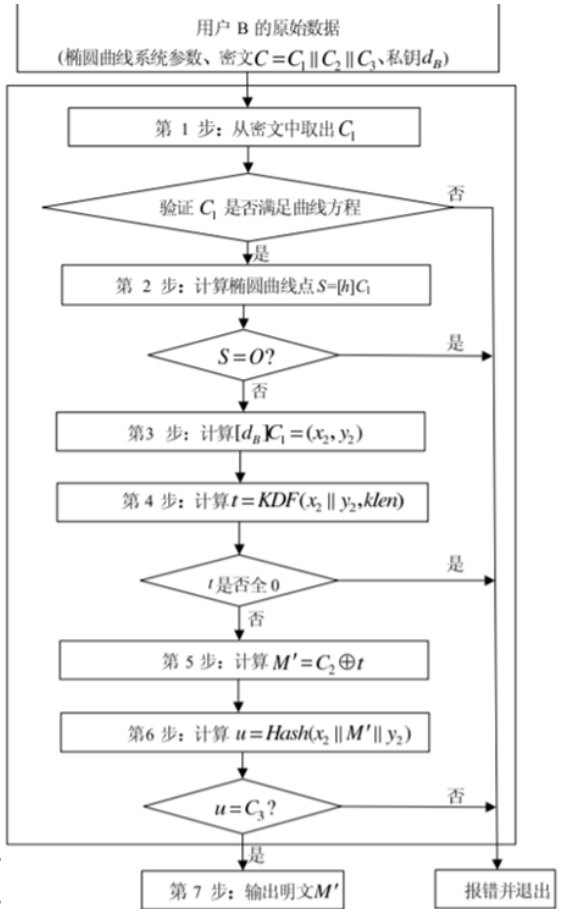


图2 解密算法流程

CSDN @千千

加密过程:

1. 从指定范围内 $[1, n-1]$ 用随机数发生器产生一个随机数 k 。
2. 使用椭圆曲线点运算计算点 $C1 = k * G$ ，其中 G 是椭圆曲线的基点，将 $C1$ 的数据类型转换为比特串。
3. 使用椭圆曲线点运算计算点 $S = h * PB$ ，其中 PB 是接收方的公钥， h 是一个固定的参数。若 S 是无穷远点，则报错并退出。
4. 使用椭圆曲线点运算计算点 $[k]PB = (x_2, y_2)$ ，将坐标 x_2 、 y_2 的数据类型转换为比特串。
5. 使用密钥派生函数（KDF）计算比特串 $t = \text{KDF}(x_2 \parallel y_2, \text{klen})$ ，若 t 为全 0 比特串，则返回 $A1$ 。
6. 对明文 M 进行异或操作，计算 $C2 = M \oplus t$ 。
7. 计算 $C3 = \text{Hash}(x_2 \parallel M \parallel y_2)$ 。
8. 输出密文 $C = C1 \parallel C2 \parallel C3$ 。

解密过程：

为了对密文 $C = C1 \parallel C2 \parallel C3$ 进行解密，作为解密者的用户 B 应实现以下运算步骤：

1. 从 C 中取出比特串 $C1$ ，将 $C1$ 的数据类型转换为椭圆曲线上的点，并验证 $C1$ 是否满足椭圆曲线方程，若不满足则报错并退出。
2. 使用椭圆曲线点运算计算点 $S = h * C1$ ，若 S 是无穷远点，则报错并退出。
3. 使用椭圆曲线点运算计算点 $[dB]C1 = (x_2, y_2)$ ，将坐标 x_2 、 y_2 的数据类型转换为比特串。
4. 使用密钥派生函数（KDF）计算比特串 $t = \text{KDF}(x_2 \parallel y_2, \text{klen})$ ，若 t 为全 0 比特串，则报错并退出。
5. 从 C 中取出比特串 $C2$ ，计算明文 $M' = C2 \oplus t$ 。
6. 计算 $u = \text{Hash}(x_2 \parallel M' \parallel y_2)$ ，从 C 中取出比特串 $C3$ ，若 $u \neq C3$ ，则报错并退出。

7. 输出明文 M'。

代码分析

我们基于 Openssl 代码库实现 SM2，下面展示一段核心代码，其功能主要包括：

确定密钥长度 (keysize)，即椭圆曲线群的阶 n 的字节长度。

确定身份 ID

如果提供了私钥 (sk) 和公钥 (pk)，并通过验证，那么不管 genkeypair 是否为 True，都会使用提供的公私钥对；否则，如果 genkeypair 为 True，则自动生成一个合格的公私钥对。

生成 SM2 实例时，会根据是否提供公私钥对以及 genkeypair 的值，决定是否自动生成公私钥对。

```
1. SM2(BIGNUM *p = SM2_p, BIGNUM *a = SM2_a, BIGNUM *b = SM2_b, BIGNUM *n = SM2_n,
2.      EC_POINT *G = nullptr, int h = 1, int ID = 0, BIGNUM *sk = nullptr, EC_POINT *pk = nullptr,
3.      bool genkeypair = true) {
4.
5.     if (!G) {
6.         G = EC_POINT_new(EC_GROUP_new_by_curve_name(NID_sm2))
7.         ;
8.         BIGNUM *xG = BN_new();
9.         BIGNUM *yG = BN_new();
10.        BN_hex2bn(&xG, SM2_Gx);
11.        BN_hex2bn(&yG, SM2_Gy);
12.        EC_POINT_set_affine_coordinates_GFp(EC_GROUP_new_by_curve_name(NID_sm2), G, xG, yG, nullptr);
13.        BN_free(xG);
14.        BN_free(yG);
```

实验结果

```
root@LAPTOP-LNFP4GC:/mnt/e/20181217/OpenSSL-Test/EVP_Encrypt# ./mysm2_encrypt
SM2 private key (in hex form):
1238D18D2CF5FE138C39F4CB1E8324D2B1CC39570797A95223090A070654C48B

x coordinate in SM2 public key (in hex form):
7F536C6D664A5026D9FC05A36CF83C4A05EB5AF5F9D2B3ADE7955A3739859E7

y coordinate in SM2 public key (in hex form):
D6D524F3C97FB4E436DAB47471E2DA6A22CFF4F2F871EAF9EF538181F11E12D

Message length: 16 bytes.
Message:
0x32 0x30 0x31 0x38 0x31 0x32 0x31 0x37 0x32 0x30 0x31 0x38 0x31 0x32 0x31 0x37

Ciphertext length: 123 bytes.
Ciphertext (ASN.1 encode):
0x30 0x79 0x2 0x20 0x43 0x15 0x98 0xc2 0x50 0xba 0x97 0x44 0x71 0x90 0x7d 0x5c 0x99 0xe8 0xc5 0x57 0xf3 0xf9 0x
14 0xd7 0xfb 0xba 0x26 0xd5 0x13 0x54 0x30 0xf2 0x3 0xb9 0xf5 0x8e 0x2 0x21 0x0 0x94 0x3f 0xb3 0x26 0x1a 0xa5
0x5 0x53 0x8c 0x6d 0x13 0xf9 0xf5 0xe1 0x98 0x56 0x27 0x15 0x5f 0x25 0xe8 0x7b 0x8f 0x79 0x4b 0x5 0x5e 0x3a 0x5
5 0xe0 0x4a 0x6c 0x4 0x20 0xc7 0x7 0x92 0x36 0x18 0x14 0x9e 0x7a 0x88 0xa 0xe7 0xf5 0x31 0xc4 0x19 0x3b 0x56 0
xe 0xa2 0x15 0x96 0x41 0x72 0xec 0xf9 0x11 0x93 0xfd 0xa4 0x56 0xb 0x13 0x4 0x10 0xda 0x8d 0xb1 0xec 0xb8 0xd0
0x41 0x6b 0xf0 0x38 0x9a 0x61 0xea 0x67 0x25 0x5c

Decrypted plaintext length: 16 bytes.
Decrypted plaintext:
0x32 0x30 0x31 0x38 0x31 0x32 0x31 0x37 0x32 0x30 0x31 0x38 0x31 0x32 0x31 0x37

Encrypt and decrypt data succeeded!
root@LAPTOP-LNFP4GC:/mnt/e/20181217/OpenSSL-Test/EVP_Encrypt# |
```

SM2 实现成功!

与密码库中的 SM2 函数的运行速度比较，结果如下图所示：

密码库的 SM2 函数	265ms
自己编写的 SM2 函数	1.6s

参考文献

参考文献：

【1】：<https://zhuanlan.zhihu.com/p/455030060>

【2】：<https://zhuanlan.zhihu.com/p/442141489>

【3】：<https://baike.baidu.com/item/SM2/15081831?fr=aladdin>

【4】：<https://zhuanlan.zhihu.com/p/347750862>

【5】：<https://gitee.com/basddsa/hggm>

【6】：<https://blog.csdn.net/chexlong/article/details/103293311>

【7】： https://blog.51cto.com/u_14793075/5307999

【8】： https://sca.gov.cn/sca/xwdt/2010-12/17/content_1002386.shtml.