南開大學
Nankai University

网络空间安全学院

密码学实验报告

实验五：MD5 实验报告

姓名：魏伯繁

学号：2011395

专业：信息安全

2022 年 12 月 28 日

目录

1 实验目的与实验要求

1.1 实验目的

通过手动编写 MD5，了解哈希函数的实现过程，理解哈希函数的特征以及其实现机理，为后续密码学的学习奠定基础。

1.2 实验要求

1、自己编写完整的 MD5 实现代码，并提交程序和程序流程图。2、对编好的 MD5 算法，测试其雪崩效应，要求给出文本改变前和改变后的 Hash 值，并计算出改变的位数。写出 8 次测试的结果，并计算出平均改变的位数。

2 实验理论

2.1 哈希函数

Hash 函数是将任意长的数字串转换成一个较短的定长输出数字串的函数，输出的结果称为 Hash 值。

Hash 函数具有如下特点：

(1) 快速性：对于任意一个输入值 x，由 Hash 函数，计算 Hash 值 y，即是非常容易的。

(2) 单向性：对于任意一个输出值 y，希望反向推出输入值 x，使得，是非常困难的。

(3) 无碰撞性：包括强无碰撞性和弱无碰撞性，一个好的 Hash 函数应该满足强无碰撞性，即找到两个不同的数字串 x 和 y，满足，在计算上是不可能的。

Hash 函数可用于数字签名、消息的完整性检验。消息的来源认证检测等。现在常用的 Hash 算法有 MD5、SHA - 1 等。下面从 MD5 入手来介绍 Hash 算法的实现机制。

2.2 MD5 哈希算法

MD 系列单向散列函数是由 Ron Rivest 设计的，MD5 算法对任意长度的输入值处理后产生 128 位的 Hash 值。MD5 算法的实现步骤如下:

在 MD5 算法中，首先需要对信息进行填充，使其字节长度与 448 模 512 同余，即信息的字节长度扩展至，n 为一个正整数。填充的方法如下：在信息的后面填充第一位为 1，其余各位均为 0，直到满足上面的条件时才停止用 0 对信息的填充。然后，再在这个结果后面附加一个以 64 位二进制表示的填充前信息长度。经过这两步的处理，现在的信息字节长度为，即长度恰好是 512 的整数倍，这样做的目的是为了满足后面处理中对信息长度的要求。

MD5 中有 A、B、C、D，4 个 32 位被称为链接变量的整数参数，它们的初始值分别为：

$$A0 = 0x01234567,\ B0 = 0x89abcdef,C0=0xfedcba98,D0=0x76543210$$

当设置好这 4 个链接变量后，就开始进入算法的 4 轮循环运算。循环的次数是信息中 512 位信息分组数目，也就是其关于 512 的倍数。

首先将上面 4 个链接变量复制到变量 A、B、C、D 中，以备后面进行处理。

然后进入主循环，主循环有 4 轮，每轮循环都很相似。第一轮进行 16 次操作，每次操作对 A、B、C、D 中的 3 个做一次非线性函数运算，然后将所得结果加上第四个变量，文本的一个子分组（32 位）

和一个常数。再将所得结果向左循环移 S 位，并加上 A、B、C、D 其中之一。最后用该结果取代 A、B、C、D 其中之一。

$$F(B,C,D) = (B \wedge C) \vee (\overline{B} \wedge D)$$

$$G(B,C,D) = (B \wedge D) \vee (C \wedge \overline{D})$$

$$H(B,C,D) = B \oplus C \oplus D$$

$$I(B,C,D) = C \oplus (B \vee \overline{D})$$

图 2.1: 逻辑函数表达式

MD5 轮主要操作为:

$$a = b + ((a + f(b,c,d) + M + t \ll s))$$

对应于四轮操作，f 分别取 F，G，H，I；对每一轮的 16 次运算，M 分别取 M1，M2，…，M16。对于 4 轮共 64 次运算，t 为给定的一些常数，另外一个常数是的整数部分，其中 i = 1，2，…，64。在 MD5 中，i 的单位是弧度，由此构成了 32 位的随机数源，它消除了输入数据中任何规律性的特征，这些随机数源的来源其实是对 sin 函数做运算，我们在计算的过程中不需要实际计算，可以根据课本上提供的常数表进行录入即可，运算时直接查表取值，增加运算效率。

对于 4 轮 64 次操作的具体运算，。所有这些操作完成之后，将 A，B，C，D 分别加上 A0，B0，C0，D0。然后用下一分组数据继续进行运算，最后得到一组 A，B，C，D。把这组数据级联起来，即得到 128 比特的 Hash 结果。

每一步的运算步骤就是将 b、c、d 应用于对应轮数所对应的逻辑函数，然后将计算结果与 a 相加、与 M 相加，M 的取值来源于 512 比特的明文分组，再与对应的 t 相加然后移位。对应的 M、t、s 都可以查表得到。

3 实验流程

在介绍完 MD5 的整体步骤后，我们用一组流程图来更加清晰、明确的认识 MD5 的具体实验流程。

首先，我们先来查看 MD5 的宏观流程图，在 MD5 算法中，其核心步骤为 64 步的轮函数，在之前则需要做数据填充、长度填充等内容。

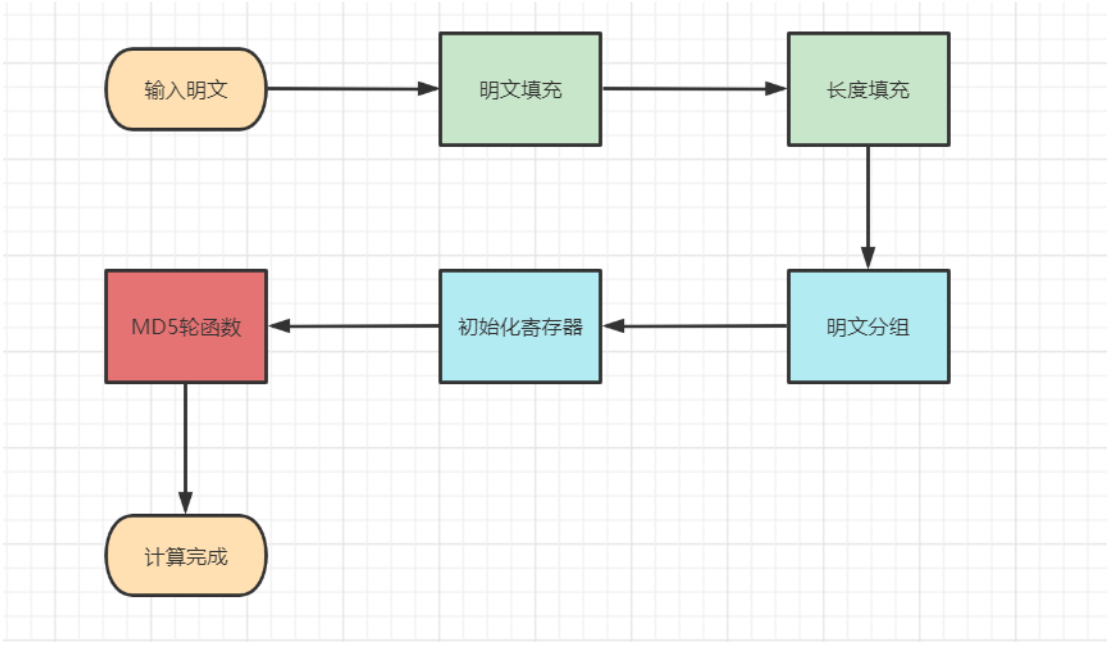

图 3.2: MD5 宏观流程图

接下来，我们深入 MD5 的轮函数，其 64 个轮函数具体又可以分为四轮，每一轮中都具有相似的步骤特征：例如使用相同的逻辑函数，在特定范围内取 T 的值等，当执行完成四步轮函数后，还需要将寄存器的值和原始寄存器的值相加来得到最后对明文分组一步的处理结果。

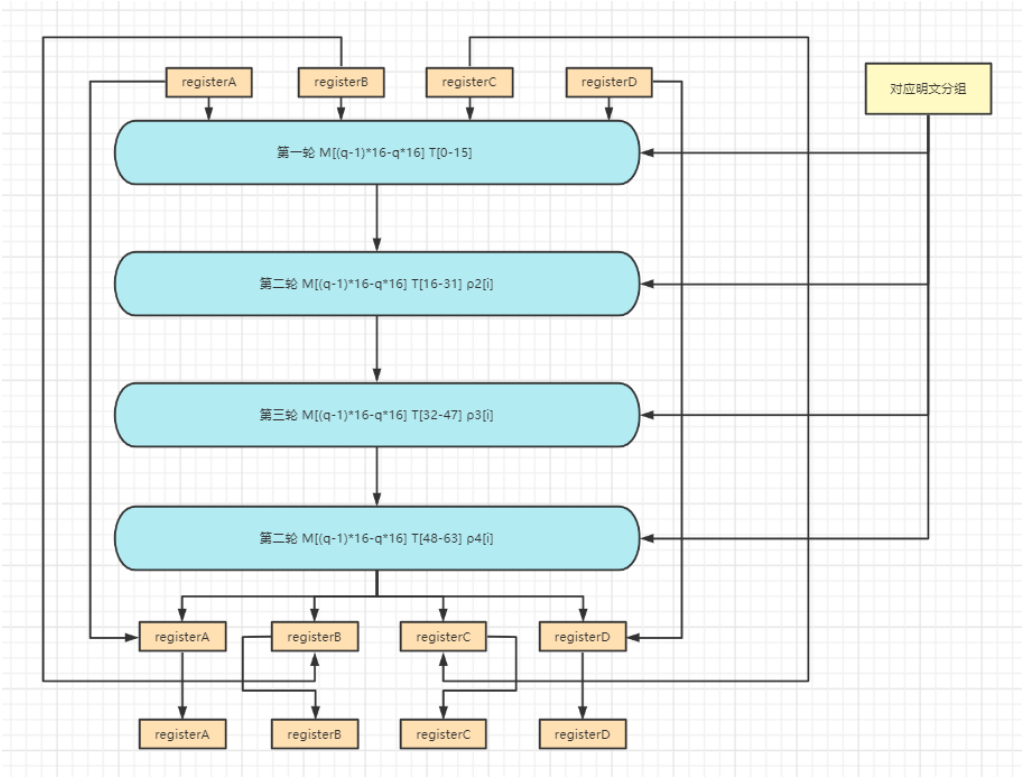

图 3.3: MD5 四轮处理流程图

最后，我们来查看每一步 MD5 都是如何处理的，具体的表达式在第三部分已经介绍过了，在这

里就不再详细说明每一步的具体含义。需要注意的是，由于对于明文的分组以及寄存器的位数都是 32 位，所以步骤中的加法都是模 232 次方加法，也就是只保留 32 位，以便于后续计算

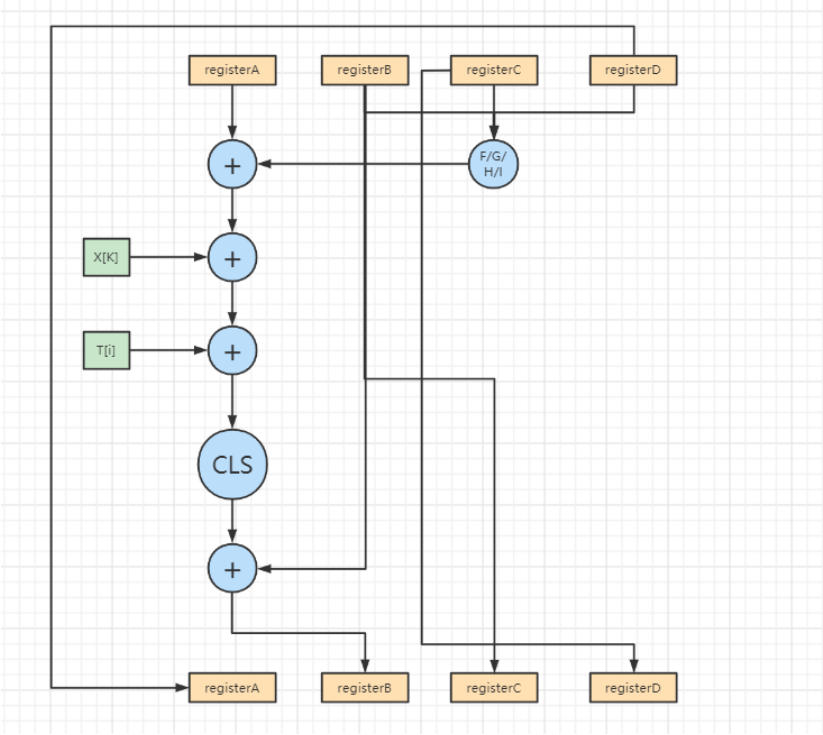

图 3.4: MD5 逐步流程图

4 代码实现

MD5 主要属于流程相对复杂，但是其具体操作的编写难度不大，所以在本节中将分布介绍 MD5 的核心部分代码。

为了正确对明文进行相应处理，我定义了一个 myMD5 类用来存储输入的文件，由于在存储时是小端序存储的，所以还需要配备一个将输入文件转换为小端序的函数 getSmalltext

```
1
2  class myMD5 {
3  public:
4      vector<string>text;
5      vector<string>smallText;
6      myMD5(vector<char>&vc);
7      myMD5(const myMD5& m);
8      int getGroupSize()const { return text.size() / 16; }
9      int getWordSize() const { return text.size(); }
10     string getGroup(int i);
11     string getWord(int i);
12     friend ostream& operator<<(ostream& out, const myMD5& m);
13     void printSmallText();
```

```
14      void getSmallText();
15  };
16
```

接下来是关于 MD5 哈希函数的整体宏观函数实现，具体包括根据输入的明文分组并填充后的需要加密的字符串进行 512bit 的分组并进行轮处理，调用轮处理函数

```
1
2       string HashMD5(myMD5& m) {
3       InitialRegister();
4       m.getSmallText();
5       m.printSmallText();
6       for (int k = 0; k < m.getGroupSize(); k++) {
7           string ra = registerA;
8           string rb = registerB;
9           string rc = registerC;
10          string rd = registerD;
11          for (int j = 1; j <= 4; j++) {
12              Myround(k, j, m);
13          }
14          registerA = stringADD(registerA, ra);
15          registerB = stringADD(registerB, rb);
16          registerC = stringADD(registerC, rc);
17          registerD = stringADD(registerD, rd);
18      }
19      string ret;
20      ret.append(smallEndRegister(registerA));
21      ret.append(smallEndRegister(registerB));
22      ret.append(smallEndRegister(registerC));
23      ret.append(smallEndRegister(registerD));
24      return ret;
25      }
26
```

接下来就是轮函数的具体实现，轮函数需要做的事情很简单，就是调用 step 函数，也就是步骤函数，并传递正确的轮数作为参数。

```
1
2       void Myround(int q, int round, myMD5& m) {
3           for (int i = 1; i <= 16; i++) {
4               step(q, round, i, m);
5           }
```

```
    }

```

在步骤函数中，会根据不同的轮数参数以及不同的明文分组参数来调用不同的函数内参数来完成一步的 MD5 函数运算

```

    void step(int q, int round, int i,myMD5 &m) {
     //把 bcd 寄存器和逻辑函数的取值计算出来
     string ADDbcd;
     if (round == 1) {
         ADDbcd = F(registerB, registerC, registerD);
     }
     if (round == 2) {
         ADDbcd = G(registerB, registerC, registerD);
     }
     if (round == 3) {
         ADDbcd = H(registerB, registerC, registerD);
     }
     if (round == 4) {
         ADDbcd = I(registerB, registerC, registerD);
     }
     //A 和逻辑函数的取值
     string Add1;
     Add1 = stringADD(registerA, ADDbcd);
     //然后把 X[k] 取出来
     int k;
     if (round == 1) { k = i-1; }
     if (round == 2) { k = round2(i-1); }
     if (round == 3) { k = round3(i-1); }
     if (round == 4) { k = round4(i-1); }
     //完成对于 Xk 的加法
     string Xk = m.getWord(q*16+k);
     string Add2 = stringADD(Xk, Add1);
     //完成对于 Ti 的加法
     string Ti = T[16 * (round - 1)+i];
     Ti = Hex2Bin(Ti);
     string Add3 = stringADD(Ti, Add2);
     //完成对于 ADD3 的移位
     string moveS = CLS(Add3, clss[round][i]);
     string newA = stringADD(moveS, registerB);
     //完成寄存器的交换
```

```
37        string ra = registerA;
38        string rb = registerB;
39        string rc = registerC;
40        string rd = registerD;
41        registerA = rd;
42        registerB = newA;
43        registerC = rb;
44        registerD = rc;
45    }
```

5 实验结果

5.1 计算结果检测

首先我们来对计算的结果进行检测，我们在控制台输入想要被计算 MD5 哈希值的文本序列，然后程序会自动返回在转换后被填充后的内容。

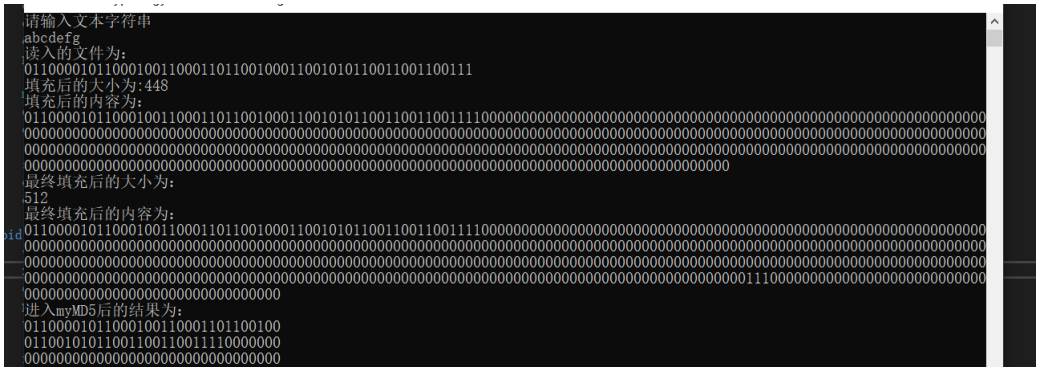

图 5.5: 输入带转换序列

经过 64 轮计算后，程序会输入最后的计算结果，这里我们对最后的寄存器中保存的小端序内容进行了转换展示，这样比较起来也较为清晰。

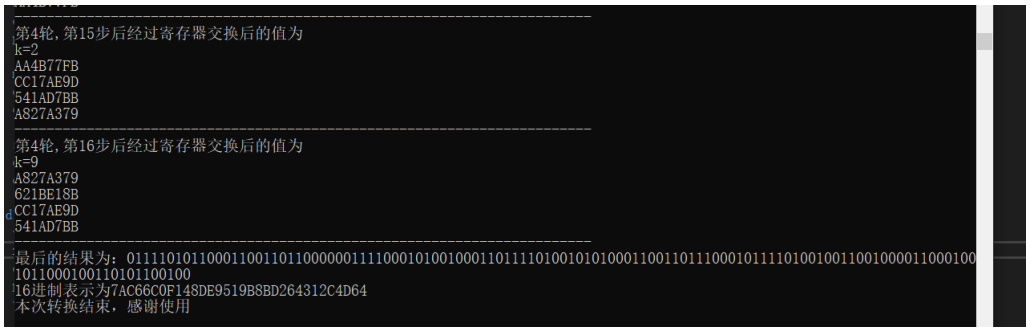

图 5.6: 展示计算结果

我们将转换后的结果和目前市面上存在的 MD5 转换程序相比较发现转换结果一致，编写 MD5 程序的工作顺利完成。

图 5.7: 与正确结果相比较

5.2 雪崩效应检测

哈希函数一个很重要的特性就是他的无规律性，通常一个比特的明文改变就会另计算结果的每一位有一般的概率改变，这是哈希函数一个非常重要的特性。

本次我们实现哈希函数的雪崩效应检测的思路是，构造一个 64 比特均为 0 的明文，随后再进行对逐比特进行更改，最后查看结果的 128 比特的变化情况。

具体的实验代码如下

雪崩效应检测函数

```
double avalanchetest() {
        cout << "下面进行雪崩效应检测" << endl;
        vector<char>vc;
        for (int i = 0; i < 64; i++) {
          vc.push_back('0');
        }
        vector<char>vc2(vc);
        fill(vc);
        fillLength(vc, vc2);
        myMD5 m(vc);
        string base = HashMD5(m);
        double finali=0;
        for (int i = 0; i < 64; i++) {
          vc.clear();
          for (int j = 0; j < 64; j++) {
            if (j == i) { vc.push_back('1'); }
            else { vc.push_back('0'); }
          }
          vector<char>vc3(vc);
          fill(vc);
          fillLength(vc, vc3);
          myMD5 mm(vc);
          string ss = HashMD5(mm);
          finali += stringDiff(base, ss);
        }
        return finali / 64.0;
}
```

最后的测试结果如下：

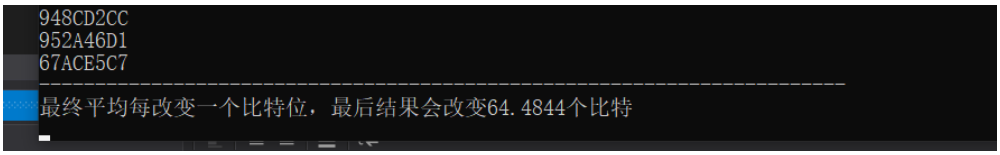

图 5.8: 雪崩效应检测

6 总结

通过本次实验，我充分理解了哈希函数的重要性质以及其特点，通过动手编写程序进一步掌握了哈希函数的具体加密流程，使我受益匪浅。