SM3

一、算法原理

SM3 是中华人民共和国政府采用的一种密码散列函数标准，由国家密码管理局于 2010 年 12 月 17 日发布。相关标准为“GM/T 0004-2012 《SM3 密码杂凑算法》”。

在商用密码体系中，SM3 主要用于数字签名及验证、消息认证码生成及验证、随机数生成等，其算法公开。据国家密码管理局表示，其安全性及效率与 SHA-256 相当。SM3 采用 Merkle-Damgard 结构。消息分组长度为 512 位，HASH 值长度为 256 位。整个算法过程可以分成四个步骤：消息填充、消息扩展、迭代压缩、输出结果。

1.消息填充

SM3 的消息扩展步骤是以 512 位的数据分组作为输入的。因此需要把原始数据长度填充至 512 位的倍数。数据填充规则如下：

1、先填充一个“1”，后面加上 k 个“0”。其中 k 是满足(n+1+k) mod 512 = 448 的最小正整数。

2、在尾部加 64 位的数据长度。

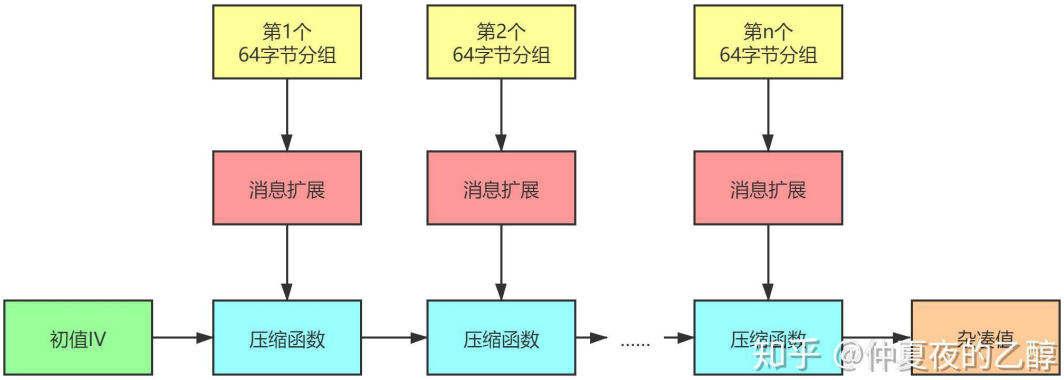
2.消息扩展

SM3 的迭代压缩步骤没有直接使用数据分组进行运算，而是使用这个步骤产生的 132 个消息字。（一个消息字的长度为 32 位/4 个字节/8 个 16j 进制数字）概括来说，先将一个 512 位数据分组划分为 16 个消息字，并且作为生成的 132 个消息字的前 16 个。再用这 16 个消息字递推生成剩余的 116 个消息字。

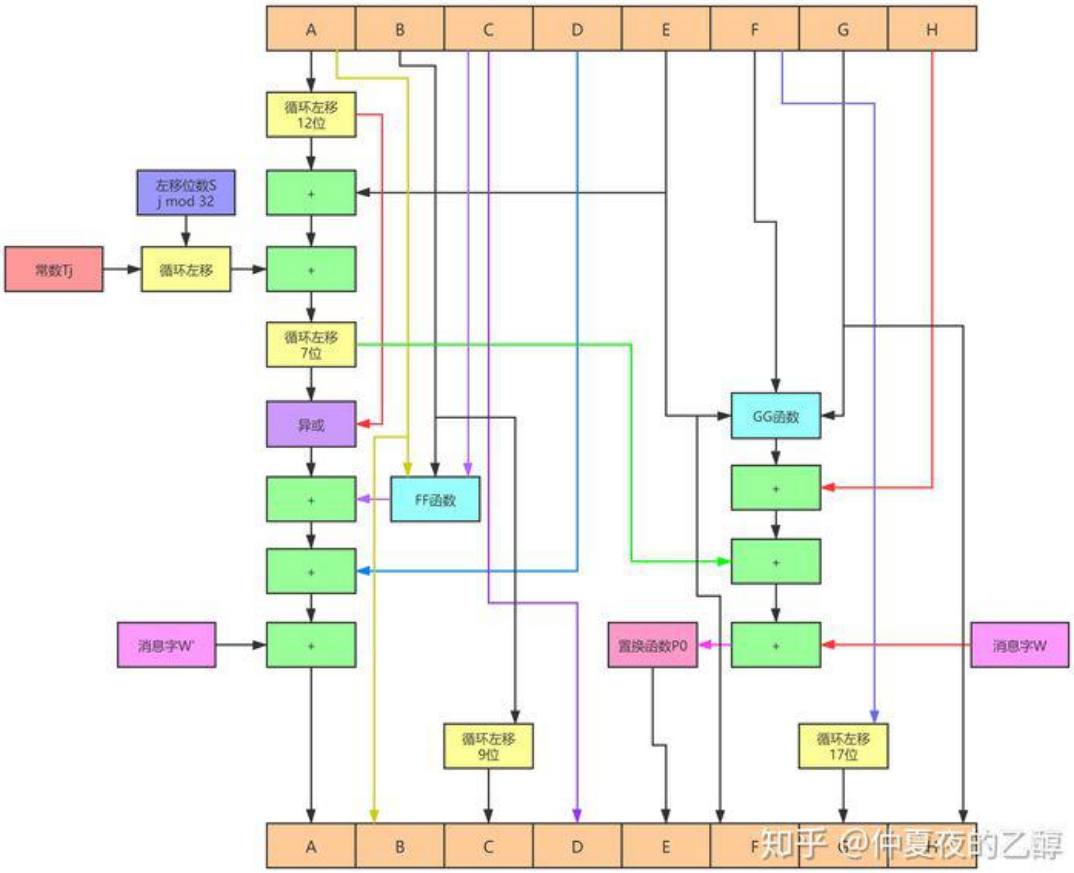
在最终得到的 132 个消息字中，前 68 个消息字构成数列 {W j {W_j}Wj}，后 64 个消息字构成数列 {W j ‘ {W_j^`}Wj'}，其中下标 j 从 0 开始计数。

3.迭代压缩

SM3 使用消息扩展得到的消息字进行运算。这个迭代过程可以用这幅图表示：



初值 IV 被放在 A、B、C、D、E、F、G、H 八个 32 位变量中，整个算法中最核心、也最复杂的地方就在于压缩函数。压缩函数将这八个变量进行 64 轮相同的计算，一轮的计算过程如下图所示：



最后，再将计算完成的 A、B、C、D、E、F、G、H 和原来的 A、B、C、D、E、F、G、H 分别进行异或，就是压缩函数的输出。这个输出再作为下一次调用压缩函数时的初值。依次类推，直到用完最后一组 132 个消息字为止。

4.输出结果

将得到的 A、B、C、D、E、F、G、H 八个变量拼接输出，就是 SM3 算法的输出。

二、生日攻击

1.生日攻击原理

生日攻击是一种密码学攻击手段，所利用的是概率论中生日问题的数学原理。

假设有一个函数 f，它的输出范围是 N，那么我们的攻击就是找到两个不同的 x，y，让 f(x)=f(y)，即 x 和 y 发生了碰撞。

根据概率论的公式，要达到 50%的几率发生碰撞，那么需要尝试的次数大约是:$\sqrt{N}$。

2.SM3 生日攻击实现

首先根据 SM3 算法原理实现 SM3，然后因为自身笔记本电脑的算力有限，因此不能对于所有的 SM3 输出进行碰撞，具体实现中是对于 SM3 所有输出比特的前 32 比特进行碰撞，即生日碰撞的尝试次数约为 2^16。在每次计算出 Hash 值后，使用 C++的 map 库存储这类一对一的数据，第一个可以称为关键字(key)，每个关键字只能在 map 中出现一次；第二个可能称为该关键字的值(value)。为了能够快速查找对比 HASH 值（find()），将 HASH 值作为 KEY，而消息作为 VALUE。

部分实现如下所示:

```
map<string, string>::iterator iter;
        iter = mapsm3.find(result.substr(0, outputlen/4));
        if (iter != mapsm3.end())
        {
            label = 1;
            cout << "Find!" << endl;
            cout << "两个发生碰撞的消息分别为：" << endl;
            cout << iter->second << endl;
            cout << data << endl;
```

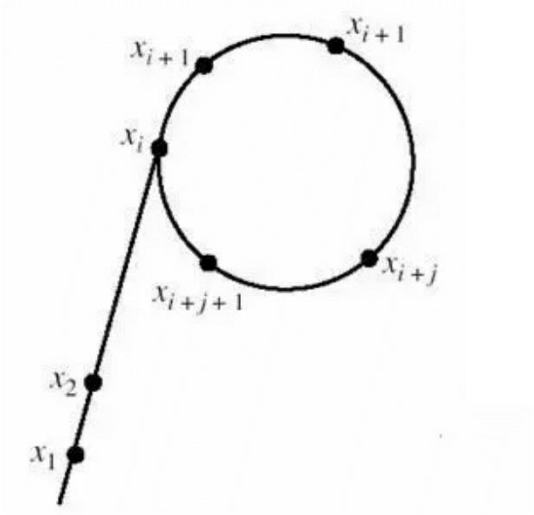
```
            cout << endl;
            cout << "前"<<outputlen<<"位哈希值为:" << endl;
            cout << result.substr(0, outputlen / 4) << endl;
            break;
        }
        else
        {
            mapsm3.insert(map<string, string>::value_type(result.substr(0, outputlen
/ 8), data));
        }
```

运行结果：

```
F:\课程\大三下\创新创业实践\SM3生日攻击\x64\Debug\SM3生日攻击.exe
考虑HASH输出碰撞的位数:8
Find!
两个发生碰撞的消息分别为：
IIMHiLNHCELiGBmQVSKbtinTksiMedgpyFfXhrTVAxJTEnGTLSBNOIXAfifvzMKLMTQFVbbpfdkmDAOdXke
JuwgAoyCxHFbKcSzDgMSdhgtFdbOqvZKnsZLHtZnSbsRvMBnYQezRVouHtCcsiZGbAcHmVCOOBAYyvMKTzqdzZeVURMLaOUktHjk

前8位哈希值为:
E0
请按任意键继续. . .
```

三、SM3 之 Rho method 攻击

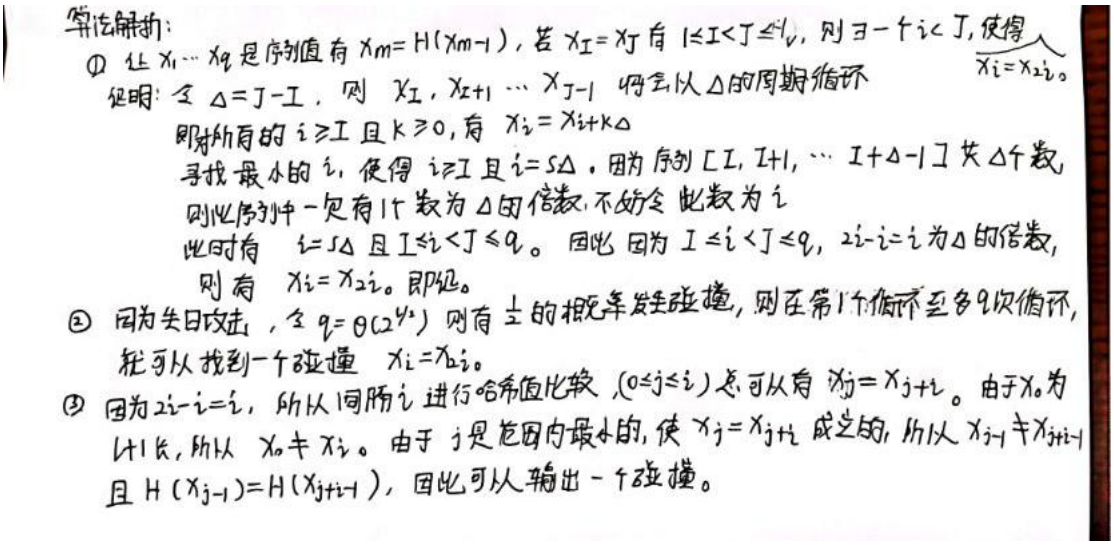
1. Rho method 原理



由上述生日算法可知假设有一个函数 f，它的输出范围是 N，那么我们的攻击就是找到两个不同的 x，y，让 f(x)=f(y)，即 x 和 y 发生了碰撞。

根据概率论的公式，要达到 50%的几率发生碰撞，那么需要尝试的次数大约是:$\sqrt{N}$。

以下是我手写相关理解推导后的扫描文件:



2.SM3 之 Rho 方法攻击实现

根据上述方法思路，在 SM3 实现的基础是尝试攻击，找到一对碰撞。在此方法中的空间复杂度相对于原始生日攻击的指数级来说，大大地进行了降低，只需要常数级的空间复杂度，大大节约了内存，不再需要记录每一对（消息-HASH）。

主要部分代码如下:

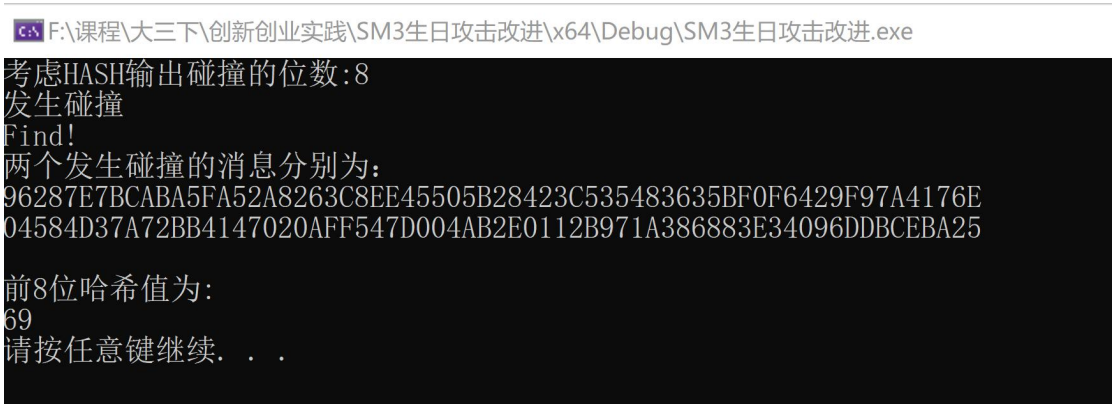
```
while (1)
    {
        n--;
        if (n == 0)
        {
            k = rand() % 500 + 20;
            data = "";
            for (int i = 1; i <= k; i++)
            {
                int x, s;
```

```
            s = rand() % 2;
            if (s == 1)
                x = rand() % ('Z' - 'A' + 1) + 'A';
            else
                x = rand() % ('z' - 'a' + 1) + 'a';
            data += (char)x;
        }
        res = "";
        for (int i = 0; i < data.size(); i++) {
            res += ten_sixteen((int)data[i]);
        }
        data = res;
        data1 = data;
        data2 = data;
        seed = 0;
        n = nnum;
    }
    seed++;

    paddingValue = padding(data1);
    data1 = iteration(paddingValue);

    paddingValue = padding(data2);
    data2 = iteration(paddingValue);
    paddingValue = padding(data2);
    data2 = iteration(paddingValue);

    if (data1.substr(0, outputlen / 4) == data2.substr(0, outputlen / 4))
    {
        break;
```

```
        }
    }
    cout << "发生碰撞" << endl;
    data2 = data1;
    data1 = data;
    string a = data1;
    string b = data2;
    label = 0;
    for (int j = 0; j < seed; j++)
    {
        paddingValue = padding(data1);
        data1 = iteration(paddingValue);

        paddingValue = padding(data2);
        data2 = iteration(paddingValue);



        if (data1.substr(0, outputlen / 4) == data2.substr(0, outputlen / 4))
        {
            label = 1;
            cout << "Find!" << endl;
            cout << "两个发生碰撞的消息分别为：" << endl;
            cout << a << endl;
            cout << b << endl;
            cout << endl;
            cout << "前" << outputlen << "位哈希值为:" << endl;
            cout << data1.substr(0, outputlen / 4) << endl;
            break;
        }
        else
```

```
        {
            a = data1;
            b = data2;
        }
    }
    if (label == 0)
    {
        cout << "没有找到碰撞" << endl;
    }
```

运行结果：



四、SM3 之长度扩展攻击

1.长度扩展攻击原理

长度扩展攻击是针对 MD 结构的散列函数作为攻击对象的，典型的 MD 结构散列函数有 MD4、MD5、RIPEMD-160、SHA-0、SHA-1、SHA-256、SHA-512、WHIRLPOOL 等。

在长度扩展攻击中，敌手已知一个消息扩展后的长度值和它的 HASH 值和以及 HASH 函数的具体结构，但是不知该消息。

敌手可以任意生成一段字符串（即为 M*），将此字符串按照 HASH 的填充机制填充到一定长度，注意要将长度为总长度，即原消息扩展后的长度+新字符串的长度。

令原 HASH 为 IV, 和扩展的新的消息分块利用 HASH 的迭代压缩函数进行求解新的 HASH（因为 HASH 函数的结构已知，Kerckhoffs 原则指出：一个密码系统的安全性不是取决于算法的保密性）

至此, 敌手得到一个新的消息 (原消息扩展后的字符串 || 新的字符串 M*) 的 HASH 值，即为上述迭代压缩函数的输出值。

2.SM3 长度扩展攻击实现

在 SM3 实现的基础上，加入上述长度扩展攻击原理，进行了实现。

```
/*在长度扩展攻击中，可以已知一个HASH值和该消息扩展后的长度值，但是不知该消息*/
    /*即在此程序中，攻击者已知result、midlen，但是不知data*/
    /*以下进行攻击*/

    k = rand() % 56+1;                          //随机生成一个分块长度
    string datan = "";
    for (int i = 1; i <= k; i++)
    {
        int x, s;                               //x表示这个字符的ascii码 ，s表示这个字符的大小写
        s = rand() % 2;
        if (s == 1)
            x = rand() % ('Z' - 'A' + 1) + 'A';
        else
            x = rand() % ('z' - 'a' + 1) + 'a';
        datan += (char)x;//将x转换为字符输出
    }
    res = "";
    for (int i = 0; i < datan.size(); i++) {
        res += ten_sixteen((int)datan[i]);
    }
    datan = res;//十六进制的一个随机消息分块（此为攻击者随机生成的一串消息）

    /*利用已知的HASH值和原消息扩展后的长度值，来伪造一个新消息的HASH*/
    //扩展新的消息分块，注意长度为总长度，即原消息扩展后的长度+新消息分块的长度
    int res_length = res.size() * 4;
    res += "8";
    while (res.size() % 128 != 112) {
        res += "0";
    }
    string res_len = ten_sixteen(res_length+ midlen);
```

```
	while (res_len.size() != 16) {
		res_len = "0" + res_len;
	}
	res += res_len;

	//令原HASH为IV，和扩展的新的消息分块进行求解新的HASH（因为HASH函数的结构已知）
	string paddingValuen = res;
	string V = result;
	string B = paddingValuen;
	string extensionB = extension(B);
	string compressB = compress(extensionB, V);
	V = XOR(V, compressB);//V 为伪造出的 HASH 值
```

在真实环境的敌手中，敌手并不知道原来的消息。在本程序中，为了检验伪造结果的正确性，对于要伪造的新消息（原消息扩展后的字符串 || 敌手所选的新的字符串）使用原始的加密流程进行了加密，和伪造的结果进行了比对，进一步说明了长度扩展攻击结果的正确性。

```
//为了检验是否伪造成功，进行如下步骤，正式攻击中，由于不知原来的消息，不能进行以下步骤，直接输出攻击结果
	string newmassage = paddingValue + datan;//伪造的总的新消息
	string a = padding(newmassage);
	string truehash = iteration(a);//算法求出此消息的HASH

	cout<<"伪造的完整消息是："<< newmassage << endl;
	if (truehash == V)//进行结果比较
	{
		cout << "伪造成功！" << endl;
		cout<< "Hash为：" << V << endl;
	}
```

该攻击实现了对于任意一串消息，在得知它的标签后，都可以基于原消息成功伪造新的消息和标签。