SHA-256 加密算法的改进研究

杨昀昶[1]

（西安邮电大学经济与管理学院,陕西省西安市 710121）

通信作者联系方式：（Email:yangyunchang001@gmail.com 电话：029-68801085 手机：18292577417）
投稿日期：2018-6-24,
作者简介：
杨昀昶（1997-），男，本科，从事量化投资与市场微观结构理论的研究。
Email:yangyunchang@gmail.com

摘 要 随着区块链在金融各个领域的快速发展，而区块链算法的核心在于加密算法的优化。而当前，区块链使用的加密算法为 SHA-256 算法，而改进 SHA-256 散列算法，可以提高区块链的运算效率。现时，大量的研究致力于加密算法的硬件优化，少有利用密码学原理对算法进行软件优化。本文主要对 SHA-256 算法进行了分析，并对其进行改进，并通过实证分析，证明了改进的 SHA-256 算法拥有更好的安全性、抗碰撞性和重合率低的性质，并且带来了更好的公平性。

首先，本文增加了每个 512 位数据块的 Hash 运算的次数，可以提高算法的非线性扩散性；其次本文修正了 SHA-256 算法中的压缩函数，并修正了主循环模块的算法，同时给出了新的常数序列。

其次，本文通过分析，发现对 SHA-256 算法的改进，提高了算法的安全性，伪随机性等；

最后，本文对优化后的算法进行了实证研究，发现改进的 SHA-256 算法具有较强的雪崩效应。

关键词 SHA-256 算法 密码学 Hash 运算 安全性分析 雪崩效应

1 SHA-256 散列算法概述

1.1 安全散列算法 SHA

安全散列算法 SHA（Secure Hash Algorithm）[1]是由美国国家标准技术研究所发布的国家标准 FIPS PUB 180，这个标准也已经更新到 FIPS PUB 180-3。其中规定了 SHA-1、SHA-224、SHA-256、SHA-384 和 SHA-512 这几种单向散列算法。而前三者对长度不超过 2^64 二进制位的消息适用。后面两个对长度不超过 2^128 二进制位的消息适用。

散列其实就是对信息的提炼，而根据其加密性强弱，其可逆程度也会不同，加密性越强，不可逆性就越强。也就是说输入信息的任何变化，都会导致散列结果的巨大变化，这就是雪崩效应。

1.2 SHA-256 基本架构

SHA-256 散列算法操作可以简单地划分为三个步骤：

1） 预处理：对原消息进行补位、补长度等处理。

2） 消息调度程序：该程序可以从 16 位的输入消息块生成 64 位。

3） 压缩功能：在每一轮中都会使用一个消息依赖的实际散列操作函数。

1.3 SHA-256 算法目标

SHA-256 算法的目标是将一个长度不大于 2^64 位的消息（原消息）通过压缩映射为 256 位的二进制值（消息摘要）。

$$\mathrm{h}: a^{2\wedge 64} \xrightarrow{\Delta} b^{256}$$

其中，a 为原消息的输入，b 为消息摘要的输出，长度为 256 位。

2 SHA-256 算法的实现

2.1 填充消息

填充消息的目标是将原消息的长度扩展为 SHA256 块大小的倍数（512 位的倍数）。假设一个消息的长度为 L 位，那么，在消息后先添加一位“1”，再添加 k 位“0”，使得其比特长度与 448 mod 512 同余，填充的比特数范围为[1,512]：

$$\mathrm{L} + 1 + \mathrm{k} = 448 \bmod 512$$

2.2 解析填充的消息

在原始消息被填充后，它被解析为 N 个 512 位的块。每当输入第 i 个 512 位的数据块时，就会将这个数据块分成 16 个 32 位的字。再用 W 的计算公式将 32 位数据扩展为 64 位数据。

$$W_t = \begin{cases} M_1^{(i)} & 0 \le t \le 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

其中：

$$\sigma_0(x) = ROTR^7(x) + ROTR^{11}(x) + SHR^3(x)$$

$$\sigma_1(x) = ROTR^2(x) + ROTR^{19}(x) + SHR^{10}(x)$$

其中，+是比特异或，ROTR^n 是循环右移 n 位，SHR^n 是逻辑右移 n 位

2.3 设定初始哈希值

在散列计算开始之前，初始散列值被设定为前 8 个质数（2,3,5,7……）的小数部分的二进制形式的前 32 位。

$H_0^{(0)}$	$H_1^{(0)}$	$H_2^{(0)}$	$H_3^{(0)}$
6a09e667	bb67ae85	3c63f372	510e527f
$H_4^{(0)}$	$H_5^{(0)}$	$H_6^{(0)}$	$H_7^{(0)}$
a54ff53a	9b05688c	1f83d9ab	5be0cd19

初始化工作变量：

$$\mathrm{A} = H_0^{(0)}$$
$$\mathrm{B} = H_1^{(0)}$$
$$\mathrm{C} = H_2^{(0)}$$
$$\mathrm{D} = H_3^{(0)}$$
$$\mathrm{E} = H_4^{(0)}$$
$$\mathrm{F} = H_5^{(0)}$$
$$\mathrm{G} = H_6^{(0)}$$
$$\mathrm{H} = H_7^{(0)}$$

2.4 消息压缩功能

消息压缩函数执行实际散列运算，且是强迫 SHA-256 算法为单向性质的主要操作。通过每一轮计算两个临时变量 T1,T2，让前述八个工作变量 A,B,C,D,E,F,G,H 初始化，初始定义由上一小节的$H_0^{(0)}$~$H_7^{(0)}$给出。完成初始化后，通过 Wt 和一个特定的 32 位常量

Kt 以及中间变量和逻辑函数计算出每一轮的工作变量。

对于$0 \le t \le 63$，执行：

$$T1 = h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$$

$$T2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$$

$$h = g$$
$$g = f$$
$$f = e$$
$$e = d + T1$$
$$d = c$$
$$c = b$$
$$b = a$$
$$a = T1 + T2$$

其中，逻辑函数为

$$Ch(x,y,z) = (x\Delta y) + (\neg x\Delta z)$$

$$Maj(x,y,z) = (x\Delta y) + (x\Delta z) + (y\Delta z)$$

$$\sum_0^{\{256\}}(x) = ROTR^2(x) + ROTR^{13}(x) + ROTR^{22}(x)$$

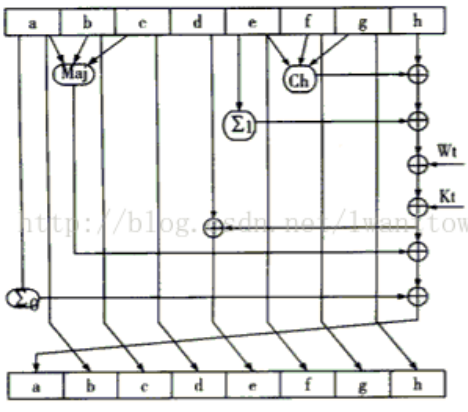
$$\sum_1^{\{256\}}(x) = ROTR^6(x) + ROTR^{11}(x) + ROTR^{25}(x)$$

其中，Δ是比特与，$\neg$是比特取反

Kt 常数为：（详见附录 1）

整体计算思路：

图 1-1 SHA-256 算法计算思路图



2.5 每个分组的中间散列值计算方法

$$H_0^{(i)} = a + H_0^{(i-1)}$$
$$H_1^{(i)} = b + H_1^{(i-1)}$$
$$H_2^{(i)} = c + H_2^{(i-1)}$$
$$H_3^{(i)} = d + H_3^{(i-1)}$$
$$H_4^{(i)} = e + H_4^{(i-1)}$$
$$H_5^{(i)} = f + H_5^{(i-1)}$$
$$H_6^{(i)} = g + H_6^{(i-1)}$$
$$H_7^{(i)} = h + H_7^{(i-1)}$$

将所有分组处理完后，最后输出 256 位的 Hash 值。

最终该消息摘要可以表示为：

$$SHA256(M) = H_0^N : H_1^N : \cdots\cdots : H_7^N$$

3 改进 SHA-256 算法的实现

3.1 算法改进思路

为了进一步加强区块链的安全性，提高加密算法的雪崩效应和抗碰撞性。本文在本章提出改进的方案。

在中本聪（2008）提出 bitcoin 的概念的时候，SHA-256 被认为是最安全的散列算法之一。但后来发现，尤其是王小云等人发现，研究人员开始怀疑算法的难度不够，安全性不足，可能存在漏洞。

因此，本文将从如下一些方面提出关于 SHA-256 算法的改进思路：1）增加 Hash 运算的轮次；2）改进 SHA-256 算法的压缩函数；3）改进原有的常数 Kt。

3.2 改进算法实现

解析填充消息阶段：

对于数据块$M^{(i)}$,作出如下变换：

在原始消息被填充后，它被解析为 N 个 512 位的块。每当输入第 i 个 512 位的数据块时，就会将这个数据块分成 16 个 32 位的字。再用 W 的计算公式将 32 位数据扩展为 80 位数据。

$$W_t = \begin{cases} M_1^{(i)} & 0 \le t \le 15 \\ \sigma_1(W_{t-5}) + W_{t-8} + \sigma_0(W_{t-16}) + W_{t-9} & 16 \le t \le 79 \end{cases}$$

其中：

$$\sigma_0(x) = ROTR^8(x) + ROTR^{16}(x) + SHR^5(x)$$

$$\sigma_1(x) = ROTR^{12}(x) + ROTR^{20}(x) + SHR^8(x)$$

消息压缩功能阶段：

对于$0 \le t \le 80$，执行：

$$T1 = h + \sum_{1}^{\{256\}}(e) + Ch(e,f,g) + K_t^{*\{256\}} + W_t$$

$$T2 = \sum_{0}^{\{256\}}(a) + Maj(a,b,c)$$

$$h = g + T1$$

$$g = f + T2$$

$$f = e + T1 + T2$$

$$e = d + T1$$

$$d = c + T2$$

$$c = b + T1$$

$$b = a + T2$$

$$a = T1 + T2$$

其中，逻辑函数为

$$Ch(x,y,z) = (x\Delta y) + (\neg x\Delta z) + (y\Delta\neg z)$$

$$Maj(x,y,z) = (x\Delta y) + (x\Delta z) + (y\Delta z)$$

$$\sum_{0}^{\{256\}}(x) = ROTR^{8}(x) + ROTR^{16}(x) + ROTR^{21}(x)$$

$$\sum_{1}^{\{256\}}(x) = ROTR^{4}(x) + ROTR^{12}(x) + ROTR^{28}(x)$$

其中，Δ是比特与，¬是比特取反

新的 Kt 常数为：（详见附录 2）

附录 2 中常数 Kt 前 64 个与附录 1 相同，后 16 个为新的常数。附录 2 只写出新增的常数。

3.3 改进的 SHA-256 算法实证仿真测试

本文使用 Python 3.6.4（64 位）平台，基于 iPython 程序，调用本文的源代码（见附录 3），输入相应的字符，并进行仿真测试。本文使用的计算机是西安邮电大学计算金融实验室提供的四屏工作站，操作系统为 Windows 7。

1） 输入消息小于 448 位

加密过程中，选用的测试字符是‘123’。

输入消息：123

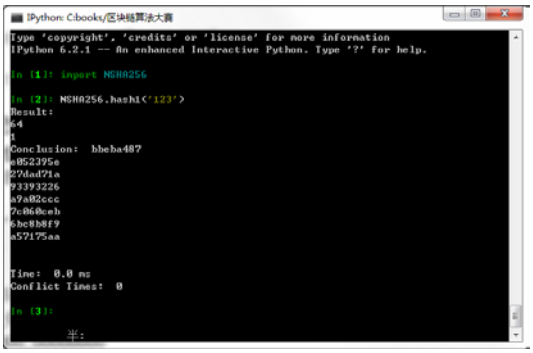
摘要输出：

bbeba47_e052395e_27dad71a_93393226_a9a02ccc_7c060ceb_6bc8b8f9_a57175aa

时间：0 秒

冲突次数：0

图 3-1 运行结果展示



2） 输入消息不小于 448 位

输入消息：

‘sascbjsbdncjkhdcihdskdk……’

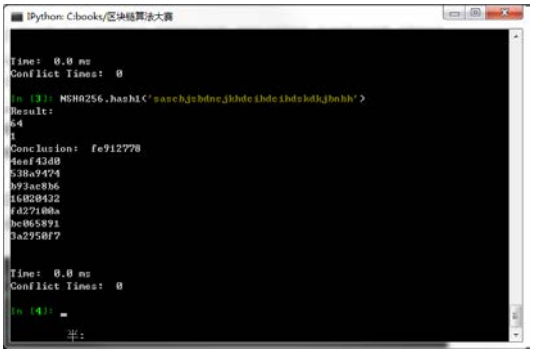
摘要输出：

fe912778_4eef43d0_538a9474_b93ac8b6_16020432_fd27100a_bc065891_3a2950f7

时间：0 秒

冲突次数：0

图 3-2 运行结果展示



4 改进 SHA-256 算法的安全性分析

4.1 冲突次数的稳固性分析

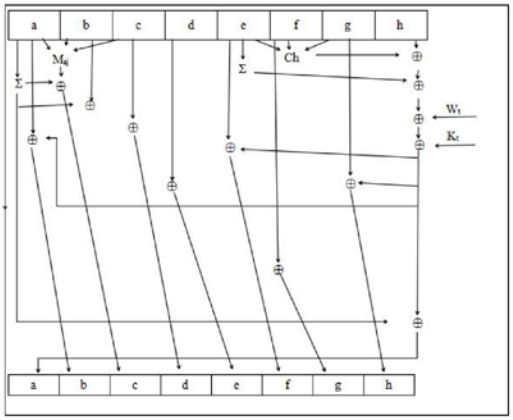
根据密码学知识，如果找到 CV 函数一个碰撞问题的解，则可以找到 Hash 函数的一个第二原像问题的解，即每个明文加密过程中的冲突次数等于此明文在 CV 中对于碰撞问题解的个数。

通过对改进算法的分析，新的算法对于明文预处理的过程，对于消息列表的计算使用更加复杂的非线性计算，所以，本文依旧认为明文到消息摘要的映射过程是严格的

单射。

本文对于八个工作变量的迭代运算过程如下图所示：

图 4-1 改进 SHA256 算法的操作过程



可以看出，所有的工作变量均为 8 位十六进制字符，即共 256 位二进制字符。显而易见，哈希函数是一个 256 位二进制字符到 256 位的二进制字符的映射。

迭代中所用到的逻辑函数均为高次非线性函数，因此哈希算法是一个双射，即多个字符映射到同一个字符的情况可以避免。因此认为，哈希函数的碰撞问题无解，所以，本算法在任何加密计算中冲突次数都均为零。

4.2 碰撞问题的安全性分析

SHA-256 算法安全性的核心在于抗碰撞能力，即黑客找出两个消息 M1 和 M2，且$M_1 \neq M_2$，使得 H（M1）=H（M2）。即密码学中，加密函数的碰撞问题的解。因此，反过来看，本文通过在现有条件下，能不能成功找到这个函数的一对碰撞，即能分析出该算法有没有抗对碰能力。目前已有的对 SHA-256 算法进行攻击的方法主要是生日攻击和差分攻击。

1） 生日攻击：这一方法是利用 Hash 值的长度，即由于 Hash 值的长度不足，而无法认为这一加密算法是可以对原消息到消息摘要是一一对应的映射。因此，抵抗生日攻击的最有效方法就是 Hash 值足够长。

2） 差分攻击：这一方法的基本原理是利用明文的输入差值对输出差值的影响，运用差分的高概率的继承或者消除还产生相同的输出。

改进的算法有 512 位哈希值，因此可以认为算法可以抵抗生日攻击。另外，根据相关算法计算得出，改进的算法迭代碰撞的复杂度约为 302，因此可以认为抵抗大多数差分攻击。

综上，可以认为改进的 SHA-256 算法有足够的安全性。

5 雪崩效应分析

所谓雪崩效应是指改进的算法，在明文有略微改变时，其最后得到的消息摘要与没有改变前的消息摘要有较大差异。因此，本文通过对比三组相似的原消息的消息摘要，来分析算法的雪崩效应。

1） abc 和 abc1

abc 对应的结果：

12bd012f_f8ddbc81_ffc88e55_726cee9e_092679a3_144f198a_cc8e837b_f050aa1e

abc1 对应的结果：

c26a2d7b_83aa522e_c48fa485_4efb3ccd_36e2d278_54bafe80_ac9bd4df_2a4e6eab

两者相似度为 3.2912%

2） 12345 和 12346

12345 的运行结果：

0d77f367_08a5341b2_98ec4b00_be799d2_f283783e_14607892_a7525aa4_1e0d1996

123456 的运行结果：

fcc36db1_d4a053fb_1ca96866_2ab22f29_d798a86d_5a86dc71_abfedea2_a1fc1605

两者相似度为 4.6788%

3） 101010 和 010101

101010 的运行结果：

bc4f88af_7376ff6e_7a2be51c_6ac0df77_d7340091_5e21f063_dd55ad71_8af14aff

010101 的运行结果：

e94b5fbf_ad65ccdb_5680c240_b1c2
8d35_beccaace_3aa86925_f122d219
_54552eae

两者相似度为 9.7281%

综上，经过微小改变，最后重合率不足 10%，因此可以认为改进的 SHA-256 算法拥有较强的雪崩效应。

6 结论

本文对 SHA-256 算法进行改进，使用了更多复杂度较高的函数和算法，从而使 SHA-256 算法的安全性有显著提高，同时，该算法还存在较为显著的雪崩效应。

参考文献

[1] 秦川红.智能变电站通信网络实时性与安全性研究[M].大连：大连理工大学，2013.

附录 1 常数 Kt

428a2f98	71374491	b5c0fbcf	e9b5dba5	3956c25b	59f111f1	923f82a4	ab1c5ed5
d807aa98	12835b01	243185be	550c7dc3	72be5d74	80deb1fe	9bdc06a7	c19bf174
e49b69c1	efbe4786	0fc19dc6	240ca1cc	2de92c6f	4a7484aa	5cb0a9dc	76f988da
983e5152	a831c66d	b00327c8	bf597fc7	c6e00bf3	d5a79147	06ca6351	14292967
27b70a85	2e1b2138	4d2c6dfc	53380d13	650a7354	766a0abb	81c2c92e	92722c85
a2bfe8a1	a81a664b	c24b8b70	c76c51a3	d192e819	d6990624	f40e3585	106aa070
19a4c116	1e376c08	2748774c	34b0bcb5	391c0cb3	4ed8aa4a	5b9cca4f	682e6ff3
748f82ee	78a5636f	84c87814	8cc70208	90befffa	a4506ceb	bef9a3f7	c67178f2

附录 2 新常数K_t^*

328a3f95	61375461	a5c0fbef	f9b5dca7
2953c25e	59f211f3	623f82a6	ad1c5ed4
d507ab93	12634b05	223175bf	750c7ac5
76ba5d34	84dec1fd	7bac06a4	c17bf162

附录 3 改进算法实现代码

NSHA256.py：

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jun 24 20:11:03 2018

This is a advanced 'SHA256' algorithm.

Call Interface:
        import NSHA256
        NSHA256.hash1(str)        'str' is string type

@author: Lewis Yeung

@Github：yangyunchang001@gmail.com
"""

import struct
import time

def out_hex(list1):    #加密算法
        for i in list1:
```

```
                print ("%08x" % i)
        print ("\n")

def rotate_left(a, k):
        k = k % 32
        return ((a << k) & 0xFFFFFFFF) | ((a & 0xFFFFFFFF) >> (32 - k))
def rotate_right(a, k):
        k = k % 32
        return (((a >> k) & 0xFFFFFFFF) | ((a & 0xFFFFFFFF) << (32 - k))) & 0xFFFFFFFF

def rotate_shift(a,k):
    k = k%32
    return ((a >> k) & 0xFFFFFFFF);

def P_0(X):
        return (rotate_right(X, 8)) ^ (rotate_right(X, 16)) ^ (rotate_shift(X,5))

def P_1(X):
        return (rotate_right(X, 12)) ^ (rotate_right(X, 20)) ^ (rotate_shift(X,8))

IV="0x6A09E667 0xBB67AE85 0x3C6EF372 0xA54FF53A 0x510E527F 0x9B05688C 0x1F83D9AB
0x5BE0CD19"

IV=IV.replace("0x","")
IV=int(IV.replace(" ",""),16)

K = """0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4,
0xab1c5ed5,
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe,
0x9bdc06a7, 0xc19bf174,
        0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa,
0x5cb0a9dc, 0x76f988da,
        0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
0x06ca6351, 0x14292967,
        0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb,
0x81c2c92e, 0x92722c85,
        0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624,
0xf40e3585, 0x106aa070,
        0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
0x5b9cca4f, 0x682e6ff3,
        0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb,
0xbef9a3f7, 0xc67178f2,
        0x328a3f95, 0x61375461, 0xa5c0fbef, 0xf9b5dca7, 0x2953c25e, 0x59f211f3,
0x623f82a6, 0xad1c5ed4,
```

```
        0xd507ab93, 0x12634b05, 0x223175bf, 0x750c7ac5, 0x76ba5d34, 0x84dec1fd,
0x7bac06a4, 0xc17bf162 """

K=K.replace("\n","")
K=K.replace(", ","")
K=K.replace(" ","")
K=K.replace("0x","")
K=int(K,16)

a = []
for i in range(0, 8):
        a.append(0)
        a[i] = (IV >> ((7 - i) * 32)) & 0xFFFFFFFF
IV = a

k = []
for i in range(0,80):
    k.append(0)
    k[i]= (K >> ((79-i)*32)) & 0xFFFFFFFF
K = k

count = 0

def CF(V_i, B_i, count):
        W = []
        for j in range(0, 16):
                W.append(0)
                unpack_list = struct.unpack(">I", B_i[j*4:(j+1)*4])
                W[j] = unpack_list[0]
                count +=1
        for j in range(16, 80):
                W.append(0)
                count +=1
                s0 = P_0(W[j-16])
                s1 = P_1(W[j-5])
                W[j] = (W[j-8] + s0 + W[j-9] + s1) & 0xFFFFFFFF
        W_1 = []


        A, B, C, D, E, F, G, H = V_i
        """
        print "00",
        out_hex([A, B, C, D, E, F, G, H])
        """
```

```
"""
S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
ch := (e and f) xor ((not e) and g)
temp1 := h + S1 + ch + k[i] + w[i]
S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
maj := (a and b) xor (a and c) xor (b and c)
temp2 := S0 + maj

h := g
g := f
f := e
e := d + temp1
d := c
c := b
b := a
a := temp1 + temp2
T1 = H + LSigma_1(E) + Conditional(E, F, G) + K[i] + W[i];
T2 = LSigma_0(A) + Majority(A, B, C);
"""
for j in range(0, 79):
        SS1 = rotate_right(E,4) ^ rotate_right(E,12) ^ rotate_right(E,28)
        SS0 = rotate_right(A,8) ^ rotate_right(A,16) ^ rotate_right(A,21)
        ch    = (E & F) ^ ((~E) & G) ^ (F & (~G))
        temp1 = (H + SS1 + ch + K[j] + W[j]) & 0xFFFFFFFF
        maj = (A & B) ^ (A & C) ^ ( B & C)
        temp2 = (SS0 + maj) & 0xFFFFFFFF
        count -=1

        H = G + temp1
        G = F + temp2
        F = E + temp1 + temp2
        E = (D + temp2)
        D = C    + temp2
        C = B    + temp1
        B = A    + temp2
        A = (temp1 + temp2)



        A = A & 0xFFFFFFFF
        B = B & 0xFFFFFFFF
        C = C & 0xFFFFFFFF
        D = D & 0xFFFFFFFF
        E = E & 0xFFFFFFFF
        F = F & 0xFFFFFFFF
```

```
                G = G & 0xFFFFFFFF
                H = H & 0xFFFFFFFF
                """
                str1 = "%02d" % j
                if str1[0] == "0":
                        str1 = ' ' + str1[1:]
                print str1,
                out_hex([A, B, C, D, E, F, G, H])
                """

        V_i_1 = []
        V_i_1.append((A + V_i[0]) & 0xFFFFFFFF)
        V_i_1.append((B + V_i[1]) & 0xFFFFFFFF)
        V_i_1.append((C + V_i[2]) & 0xFFFFFFFF)
        V_i_1.append((D + V_i[3]) & 0xFFFFFFFF)
        V_i_1.append((E + V_i[4]) & 0xFFFFFFFF)
        V_i_1.append((F + V_i[5]) & 0xFFFFFFFF)
        V_i_1.append((G + V_i[6]) & 0xFFFFFFFF)
        V_i_1.append((H + V_i[7]) & 0xFFFFFFFF)

        return V_i_1

def hash_msg(msg):      #迭代计算
        len1 = len(msg)
        reserve1 = len1 % 64
        msg1 = msg.encode() + struct.pack("B",128)
        reserve1 = reserve1 + 1
        for i in range(reserve1, 56):
                msg1 = msg1 + struct.pack("B",0)

        bit_length = (len1) * 8
        bit_length_string = struct.pack(">Q", bit_length)
        msg1 = msg1 + bit_length_string

        print (len(msg1) )
        group_count = int(len(msg1) / 64 )
        print(group_count)

        m_1 = B = []
        for i in range(0, group_count):
                B.append(0)
                B[i] = msg1[i*64:(i+1)*64]

        V = []
```

```
        V.append(0)
        V[0] = IV
        for i in range(0, group_count):
                V.append(0)
                V[i+1] = CF(V[i], B[i], count)
        print ("Conclusion: ", end=" ")
        out_hex(V[i+1])

def hash1(str):                 #调用接口
        start = time.time()
        end = time.time()
        print ("Result: ")
        hash_msg(str)
        print ("Time: ", end - start , 'ms')
        print ("Conflict Times: ", count)
```