密码学实验报告 实验九

2019 年 5 月 27 日

1 SHA-1 密码学 Hash 函数

1.1 算法原理

安全 Hash 算法 SHA（Secure Hash Algorithm）是由 NIST 于 1993 年发布的标准，SHA-1 是基于 MD4 的算法，与其结构十分相似，SHA-1 算法的输入是长度小于2^{64}的任意消息M，输出 160 位的 Hash 值。首先会将输入填充至长度为 512 比特的倍数，然后对 512 比特的分组Y_i进行处理。SHA-1 算法的中间结果和最终结果都保存在160位的缓冲区里，用5个32位的寄存器表示，记为A, B, C, D, E。

SHA-1 属于迭代 Hash 函数，其压缩函数为：

$$H_{SHA}: \{0,1\}^{160+512} \rightarrow \{0,1\}^{160}$$

SHA-1 循环以 512 比特作为分组，重复应用压缩函数H_{SHA}，直到最后一个分组，然后输出消息的 Hash 值，循环次数等于消息M中 512 比特分组的数目L。其处理过程可以归纳为：

$$CV_0 = IV$$

$$CV_i = SUM_{32}(CV_{i-1}, ABCDE_i)$$

$$MD = CV_{L-1}$$

其中IV为初始向量，$ABCDE_i$是处理第i个消息分组时最后一轮的输出，L是消息分组的个数，SUM_{32}是对输入每个字的模2^{32}相加，MD是最终的 Hash 值。

SHA-1 对每个分组使用四轮循环处理，四轮处理过程的算法结构相同，每一轮要对缓冲区$ABCDE$进行 20 次迭代，每次迭代的运算形式为：

$$A, B, C, D, E \leftarrow (E + f(B, C, D) + L^5(A) + W_t + K_t), A, L^{30}(B), C, D$$

其中加法为模2^{32}相加。

每次迭代对各个分组的处理步骤如图 1 所示。

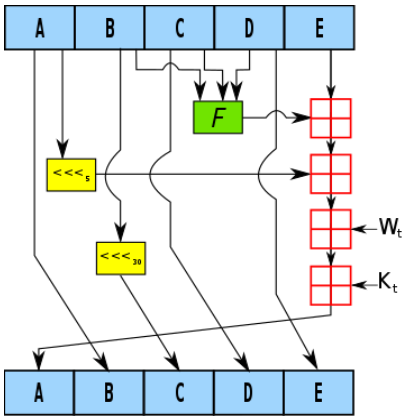

图 1 每轮迭代对分组进行的操作

其中K_i为加法常数，常数值如表 1 所列。

表 1 SHA-1 加法常数表

迭代步数	十六进制	十进制
$0 \le t \le 19$	5A827999	$\lfloor 2^{30} \times \sqrt{2} \rfloor$
$20 \le t \le 39$	6ED9EBA1	$\lfloor 2^{30} \times \sqrt{3} \rfloor$
$40 \le t \le 59$	8F1BBCDC	$\lfloor 2^{30} \times \sqrt{5} \rfloor$
$60 \le t \le 79$	CA62C1D6	$\lfloor 2^{30} \times \sqrt{10} \rfloor$

每一轮使用一个基本逻辑函数f，每个基本逻辑函数的输入是三个字，输出是一个字，它执行的是位逻辑运算，即输出的第n位是三个输入第n位的函数，基本逻辑函数f具体功能如表 2 所示。

表 2 SHA-1 基本逻辑函数功能

轮数	基本逻辑函数f	$f(B,C,D)$
1	$f_1(B,C,D)$	$(B \wedge C) \vee (\bar{B} \wedge D)$
2	$f_2(B,C,D)$	$B \oplus C \oplus D$
3	$f_3(B,C,D)$	$(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
4	$f_4(B,C,D)$	$B \oplus C \oplus D$

每一轮迭代需要提供一个 32 比特的输入字$W_t(0 \le t \le 79)$，它的前面 16 个字$W_0, W_1, \cdots, W_{15}$依次取自当前输入分组$Y_i(0 \le i \le L-1)$，其余字由以下方式扩展得到：

$$W_t = L^1(W_{t-16} \oplus W_{t-14} \oplus W_{t-8} \oplus W_{t-3}) \quad (16 \le t \le 79)$$

2017 年，Google 宣布了一个成功的 SHA-1 碰撞攻击，发布了两份内容不同但 Hash 值相同的 PDF 文件作为证明，目前 SHA-1 已被证明是一种不安全的算法。

1.2 算法实现的伪代码

SHA-1 算法 计算消息的 Hash 值

输入：消息M

输出：Hash 值H

function $SHA1(M)$

在消息M之后填充一个1和若干0，使其长度与 448 模 512 同余

用 64 位表示原始消息M的长度，并附加在消息M后

将消息M分为 512 比特的分组$Y_i(0 \le i \le L-1)$

$h_0 \leftarrow (67452301)_{16}$

$h_1 \leftarrow (\text{EFCDAB89})_{16}$

$h_2 \leftarrow (\text{98BADCFE})_{16}$

$h_3 \leftarrow (10325476)_{16}$

$h_4 \leftarrow (\text{C3D2E1F0})_{16}$

for $i \leftarrow 0 \ to \ L-1$ **do**

将Y_i分为 32 比特的字$w_j(0 \le j \le 15)$

for $j \leftarrow 16 \ to \ 79$ do

$w_j \leftarrow L^1(w_{j-3} \oplus w_{j-8} \oplus w_{j-14} \oplus w_{j-16})$

end for

$a \leftarrow h_0$

$b \leftarrow h_1$

$c \leftarrow h_2$

$d \leftarrow h_3$

$e \leftarrow h_4$

for $j \leftarrow 0 \ to \ 79$ **do**

if $0 \le j \le 19$ **then**

$f \leftarrow (b \text{ and } c) \text{ or } ((\text{not } b) \text{ and } d)$

$k \leftarrow (\text{5A827999})_{16}$

else if $20 \le j \le 39$ **then**

$f \leftarrow b \text{ xor } c \text{ xor } d$

$k \leftarrow (\text{6ED9EBA1})_{16}$

else if $40 \le j \le 59$ **then**

$f \leftarrow (b \text{ and } c) \text{ or } (b \text{ and } d) \text{ or } (c \text{ and } d)$

$k \leftarrow (\text{8F1BBCDC})_{16}$

else if $60 \le j \le 79$ **then**

$f \leftarrow b \text{ xor } c \text{ xor } d$

$k \leftarrow (\text{CA62C1D6})_{16}$

end if

$temp \leftarrow L^5(a) + f + e + k + w_j$

$e \leftarrow d$

$d \leftarrow c$
$c \leftarrow L^{30}(b)$
$b \leftarrow a$
$a \leftarrow temp$
end for
$h_0 \leftarrow h_0 + a$
$h_1 \leftarrow h_1 + b$
$h_2 \leftarrow h_2 + c$
$h_3 \leftarrow h_3 + d$
$h_4 \leftarrow h_4 + e$
end for
$H \leftarrow h_0||h_1||h_2||h_3||h_4$
return H
end function

1.3 测试样例及运行结果

我们尝试计算消息“The quick brown fox jumps over the lazy dog”的 Hash 值，字符串编码为 ASCII，SHA-1 函数的输出结果为

$$(2FD4E1C67A2D28FCED849EE1BB76E7391B93EB12)_{16}$$

再使用 Python 内置 hashlib 库中的标准 SHA-1 函数进行计算，得到结果为

$$(2FD4E1C67A2D28FCED849EE1BB76E7391B93EB12)_{16}$$

说明我们编写的 Hash 函数与标准函数输出相同，证明了算法正确性。

2 SHA-3 密码学 Hash 函数

2.1 算法原理

2.1.1 算法简介

SHA-3 为第三代安全散列算法，之前名为 Keccak 算法，SHA-3 在 2015 年 8 月 5 日由 NIST 通过 FIPS 202 正式发表。SHA-3 是一种基于替换的散列函数，故其硬件实现具有优势，由于 Keccak 算法采用了海绵结构，其输出长度可以扩展到任意位，这也是该算法被选为 SHA-3 标准的一个原因。

SHA-3 家族包括四种密码学 Hash 函数和两种可扩展输出函数，在本文主要介绍密码学 Hash 函数，分别为 SHA3-224、SHA3-256、SHA3-384 和 SHA3-512，分别对应 224 位、256 位、384 位和 512 位的输出长度，SHA-3 函数的输入消息

可以为任意长，输出的 Hash 值长度是固定的。

Keccak-p 置换是 SHA-3 的主体部分，构成海绵结构的部件，六种 SHA-3 家族的函数都可以看作是 Keccak-p[1600,24]置换，提供了 SHA-3 主要的安全性能。

给定一个消息M，四种 SHA-3 Hash 函数可以由 Keccak[c]函数定义，并且为消息M添加两比特的后缀以区分 SHA-3 可扩展输出函数族，输出长度d是固定的。

SHA3-224(M)=Keccak[448](M||01, 224);
SHA3-256(M)=Keccak[512](M||01, 256);
SHA3-384(M)=Keccak[768](M||01, 384);
SHA3-512(M)=Keccak[1024](M||01, 512);

其中容量c是输出长度d的两倍。

2.1.2 Keccak 函数

Keccak 是由 Keccak-p[b,12+2l]置换组成的海绵函数族，其填充规则被指定为 pad10*1，参数为位速率r和容量c，置换函数的输入位数为b，$b = 1600$。参数化的 Keccak[c]函数可被描述为：

Keccak[c]=Sponge[Keccak-p[1600,24], pad10*1, 1600-c]

其中 pad10*1 填充规则为：输入消息长度m，以及分组长度x，在消息M后填充一个 1，和若干个 0，最后再填充一个 1，使填充后的消息长度为分组长度x的整倍数的最小值，这种填充规则称为多重位速率填充。

需要注意的是，在 SHA-3 标准中对原始的 pad10*1 规则做了一些改动，在绝大多数情况下，消息M的长度是 8 的倍数，则填充的字节数q为

$$q = \frac{r - m \mod r}{8}$$

若$q = 1$，填充后的消息为$M||(86)_{16}$；

若$q \geq 2$，则填充后的消息为$M||(06)_{16}||(00)_{16}\cdots||(80)_{16}$，其中$(00)_{16}\cdots$代表有$q-2$个$(00)_{16}$字节。

2.1.3 海绵结构

海绵结构是一类具有任意输出长度的函数，其参数为一个固定输入长度的内部函数f、位速率r和填充规则pad，海绵结构被参数具体化后称为一个海绵函数，记作 Sponge[f, pad, r]，函数输入为一个比特串N和输出长度d，海绵函数对输入的处理分为两个阶段，分别称为“吸收”和“挤压”，如图 2 所示。

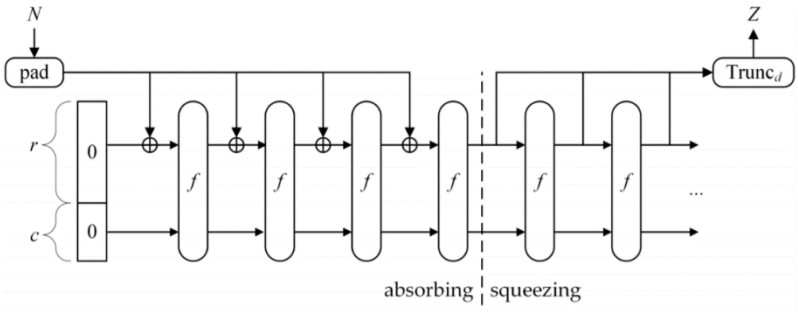

图 2 海绵函数的迭代结构

在“吸收”阶段，对于每轮迭代，通过填充若干个 0，将输入数据块的长度由r位扩展至b位，然后将扩展后的消息和s进行 XOR 异或运算得到b位的结果，并作为迭代函数f的输入，f函数的输出作为下一轮迭代中s的取值。如果需要的输出长度$d \leq b$，则直接返回s的前d位，否则进入“挤压”阶段。

在“挤压”阶段，s的前r位被保留作为输出分组Z_0，然后在每轮迭代中通过重复执行f函数来更新s的值，每次s的前r位被依次保留作为输出分组Z_i，并与前面的输出分组拼接起来，直到$j-1$次迭代后输出分组的长度满足$(j-1)\times r < d \leq j \times r$，这时将拼接后输出分组的前$d$位作为输出返回。

在 SHA-3 算法中，下游函数f是一个不可逆的置换函数，输入输出长度相同，每次对长度$b = r + c$位的状态变量s进行操作，状态变量初值为 0。位速率r是一个严格小于b的正整数，容量$c = b - r$，位速率反映了每轮迭代中处理的位数，r越大，海绵结构处理消息的速度就越快。填充规则是一个能够产生填充比特的函数，为了能够得到长度为r的消息分组，SHA-3 中被指定为 pad10*1 函数。

2.1.4 Keccak-p 置换函数

Keccak-p 函数的输入和输出均为长度$b = 1600$的状态变量s，首先将输入s重排为一个三维状态张量A，大小为$5 \times 5 \times 64$，其中每个 64 位单元被称为一个纵（lane）。用$A[x,y,z]$表示状态张量中单独的一位，而$A[x,y]$表示一个 64 位的纵。然后函数对张量执行 24 轮操作，每轮包括 5 个步骤，每个步骤通过置换或代替操作对状态张量进行更新，记作一个轮函数Rnd。

$$Rnd(A,i) = \iota\left(\chi\left(\pi\left(\rho\left(\theta(A)\right)\right)\right), RC_i\right) \quad (0 \leq i < 24)$$

其中RC_i为轮常数，通过使用不同的轮常数使每一轮操作都不相同。

每轮操作是五个映射的复合，算法实行中不需要查表、算术运算，只有对于纵的位运算和循环移位，表 3 总结了 5 个步骤的操作。

表 3 SHA-3 的 5 步映射

映射	类型	描述
θ	代替	每个字中的每一位的新值，取决于当前值、其前一列的每个字的相同位，后一列的每个字的邻接位
ρ	置换	对于每个字的内部使用循环移位进行置换操作，其中$A[0,0]$不变
π	置换	字之间进行5×5矩阵的置换，其中$A[0,0]$不变
χ	代替	每个字中的每一位的新值，取决于其当前值、相同行的下一个字的对应位的值、相同行的下下一个字的对应位的值
ι	代替	$A[0,0]$与轮常数进行异或运算

由b位的状态变量s对应的状态张量A定义为：

$$A[x,y,z]=s[64(5y+x)+z](0\le x,y<5,0\le z<64)$$

注意，在这里矩阵采用列优先的方式，即索引x表示列，索引y表示行。

若要从状态张量A恢复状态变量s时，按照列优先的次序将A的每个分量拼接起来即可。

(1) 对于θ步映射，我们定义为：

对所有的$0\le x<5,0\le z<64$，令

$$C[x,z]=A[x,0,z]\oplus A[x,1,z]\oplus A[x,2,z]\oplus A[x,3,z]\oplus A[x,4,z]$$

对所有的$0\le x<5,0\le z<64$，令

$$D[x,z]=C[(x-1)\bmod 5,z]\oplus C[(x+1)\bmod 5,(z-1)\bmod 64]$$

对所有的$0\le x<5,0\le y<5,0\le z<64$，令

$$A'[x,y,z]=A[x,y,z]\oplus D[x,z]$$

θ步映射提供了高强度的扩散效果，每一位的更新值都和原来 11 位的取值有关。

(2) 对于ρ步映射，我们定义为：

对所有的$0\le z<64$，令

$$A'[0,0,z]=A[0,0,z]$$

令$(x,y)=(1,0)$，对$t\leftarrow 0\ \ to\ \ 23$，令

$$A'[x,y,z]=A\left[x,y,\left(z-\frac{(t+1)(t+2)}{2}\right)\bmod 64\right]$$

$$(x,y)=(y,(2x+3y)\bmod 5)$$

ρ步映射的作用是提供每个纵内部的扩散，如果没有ρ步映射，纵内各个位之间的扩散将会非常缓慢。

(3) 对于π步映射，我们定义为：

对所有的$0 \le x < 5, 0 \le y < 5, 0 \le z < 64$，令

$$A'[x,y,z] = A[(x+3y) \bmod 5, x, z]$$

π步映射是张量中各个纵的置换，提供了纵之间的扩散效果。

(4) 对于χ步映射，我们定义为：

对所有的$0 \le x < 5, 0 \le y < 5, 0 \le z < 64$，令

$$A'[x,y,z] = A[x,y,z] \oplus \left((A[(x+1) \bmod 5, y, z] \oplus 1) \wedge A[(x+2) \bmod 5, y, z]\right)$$

其中∧表示逐位进行与逻辑。χ步映射提供了 SHA-3 轮函数唯一的非线性，如果没有χ步映射，整个轮函数将成为线性函数。

(5) 对于ι步映射，我们定义为：

对所有的$0 \le z < 64$，令

$$A'[0,0,z] = A[0,0,z] \oplus RC_i[z]$$

ι步映射通过将每轮互不相同的轮函数加入到变换中，打破了其他 4 个步骤的对称性，轮常数的汉明权重为 1~6，大部分位的取值为 0，在其他步中可以被扩散到状态张量中的其他位，提供了混淆效果。

ι步的各个轮常数如表 4 所示。

表 4 SHA-3 轮常数

RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
RC[6]	0x8000000080008081	RC[18]	0x000000000000800A
RC[7]	0x8000000000008009	RC[19]	0x800000008000000A
RC[8]	0x000000000000008A	RC[20]	0x8000000080008081
RC[9]	0x0000000000000088	RC[21]	0x8000000000008080
RC[10]	0x0000000080008009	RC[22]	0x0000000080000001
RC[11]	0x000000008000000A	RC[23]	0x8000000080008008

2.1.5 安全性

目前没有找到任何对 SHA-3 的有效攻击方法，对于输出长度为d的 SHA-3 函数，其抗碰撞攻击的安全长度为$d/2$，抗原像攻击和抗第二原像攻击的安全长度为d。

2.2 算法实现的伪代码

SHA-3 海绵结构 输出任意长的消息 Hash 值

输入： 消息M，输出长度d

输出： Hash 值H

function $SHA3(M, d)$
    $M \leftarrow M||0x06||0x00\cdots$
    $M \leftarrow M \oplus (0x00\cdots||0x80)$
    $s \leftarrow 0$
    for each M_i **in** M **do**
        $s \leftarrow s \oplus (M_i||0)$
        $s \leftarrow Keccak\text{-}p(s)$
    end for
    $z \leftarrow$ ""
    while $\text{len}(z) < d$ **do**
        $z \leftarrow z||s[0:r]$
        $s \leftarrow Keccak\text{-}p(s)$
    end while
    return $z[0:d]$
end function

Keccak-p 函数 对输入状态进行置换操作

输入： 状态变量s

输出： 状态变量s

function $Keccak\text{-}p(s)$
    $A[x, y, z] \leftarrow s[64(5y + x) + z]$
    for $i \leftarrow 0$ to 23 **do**
        // θ step
        for $x \leftarrow 0$ to 4 **do**
            $C[x] \leftarrow A[x, 0] \oplus A[x, 1] \oplus A[x, 2] \oplus A[x, 3] \oplus A[x, 4]$
        end for
        for $x \leftarrow 0$ to 4 **do**
            $D[x] \leftarrow C[x - 1] \oplus R^1(C[x + 1])$
        end for
        for $x \leftarrow 0$ to 4 **and** $y \leftarrow 0$ to 4 **do**
            $A[x, y] \leftarrow A[x, y] \oplus D[x]$
        end for
        // ρ and π steps
        for $x \leftarrow 0$ to 4 **and** $y \leftarrow 0$ to 4 **do**
            $B[y, 2x + 3y] \leftarrow R^{r[x,y]}(A[x, y])$
        end for
        // χ step
        for $x \leftarrow 0$ to 4 **and** $y \leftarrow 0$ to 4 **do**

```
            A[x,y] ← B[x,y] ⊕ ((not B[x+1,y]) and B[x+2,y])
        end for
        // ι step
        A[0,0] ← A[0,0] ⊕ RC[i]
    end for
    s ← []
    for y ← 0 to 4 do
        for x ← 0 to 4 do
            s ← s + A[x,y]
        end for
    end for
    return s
end function
```

2.3 测试样例与运行结果

我们尝试计算消息“The quick brown fox jumps over the lazy dog”的 Hash 值，字符串编码为 ASCII，SHA3-512 函数的输出结果为

$$\begin{pmatrix} 01DEDD5DE4EF14642445BA5F5B97C15E \\ 47B9AD931326E4B0727CD94CEFC44FFF \\ 23F07BF543139939B49128CAF436DC1B \\ DEE54FCB24023A08D9403F9B4BF0D450 \end{pmatrix}_{16}$$

再使用 Python 内置 hashlib 库中的标准 SHA3-512 函数进行计算，得到的结果为

$$\begin{pmatrix} 01DEDD5DE4EF14642445BA5F5B97C15E \\ 47B9AD931326E4B0727CD94CEFC44FFF \\ 23F07BF543139939B49128CAF436DC1B \\ DEE54FCB24023A08D9403F9B4BF0D450 \end{pmatrix}_{16}$$

说明我们编写的 Hash 函数与标准函数输出相同，证明了算法正确性。

3 HMAC 消息认证码

3.1 算法原理

HMAC（基于 Hash 的消息认证码）是密码学中消息认证码的一个分支，包括一个密码学 Hash 函数和一个指定的密钥，可以用于提供数据的完整性和真实性。任何密码学 Hash 函数例如 SHA-1 或 SHA-3，都可以用于计算 HMAC，对应的 MAC 算法被称为 HMAC-SHA1 或 HMAC-SHA3。HMAC 的强度取决于其

Hash 函数的强度以及密钥 Key 的长度和质量。

HMAC 使用了两次 Hash 计算，首先由原始密钥派生出两个子密钥——内部密钥和外部密钥。首先将内部密钥和消息传给 Hash 函数计算出内部 Hash 值，然后将外部密钥和内部 Hash 值再传给 Hash 函数计算出最终的 HMAC，这种设计方法是为了抵抗长度扩展攻击。

HMAC 并不会加密消息，因此消息必须与 HMAC 一起传输，对方利用密钥重新计算消息的 HMAC，若与接收的 HMAC 相一致，则说明消息是真实的。

HMAC 可以被简洁地描述为：

$$HMAC(K,m) = H\left((K' \oplus opad)||H\big((K' \oplus ipad)||m\big)\right)$$

其中

$$K' = \begin{cases} H(K) & \text{if } K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

H为密码学 Hash 函数

b为 Hash 函数分组长度

m为待认证的消息

K为原始密钥

K'是为使K的长度为b位而在K的右侧填充若干个0所得的结果

ipad 为字节 0x36 重复 b/8 次后的字节串

opad 为字节 0x5C 重复 b/8 次后的字节串

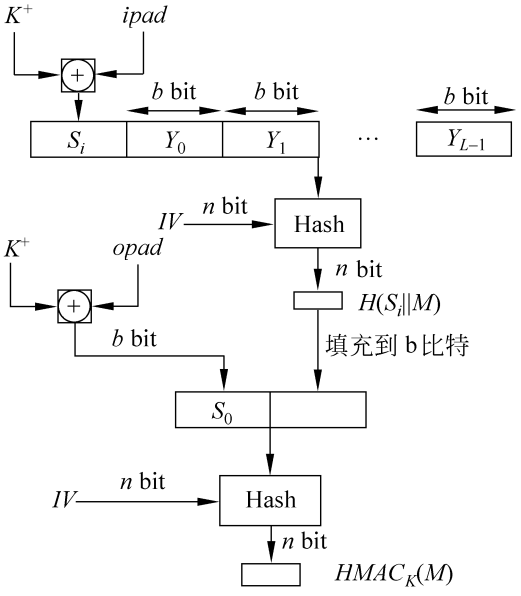

图 3 HMAC 的结构

3.2 算法实现的伪代码

HMAC 算法 产生基于 Hash 函数的消息认证码

输入： 密钥K，消息m
输出： 消息认证码$HMAC$
function $HMAC(K, m)$
    if $\text{len}(K) > b$ **then**
        $K \leftarrow H(K)$
    end if
    在K后填充若干个 0 得到b位的K'
    $S_i \leftarrow K' \oplus ipad$
    $inter \leftarrow H(S_i || M)$
    $S_o \leftarrow K' \oplus opad$
    $HMAC \leftarrow H(S_o || inter)$
    return $HMAC$
end function

3.3 测试样例及运行结果

令消息m为“The quick brown fox jumps over the lazy dog”，密钥为“key”，两者均采取 ASCII 编码，使用我们自己编写的 HMAC 算法，并用之前实现的 SHA-1 和 SHA3-512 作为内部 Hash 函数产生消息认证码，再与 Python 内置 hmac 库的标准函数输出作对比。

HMAC-SHA1 函数的输出结果为

$$(\text{DE7C9B85B8B78AA6BC8A7A36F70A90701C9DB4D9})_{16}$$

标准 HMAC-SHA1 函数的输出结果为

$$(\text{DE7C9B85B8B78AA6BC8A7A36F70A90701C9DB4D9})_{16}$$

HMAC-SHA3 函数的输出结果为

$$\begin{pmatrix} \text{237A35049C40B3EF5DDD960B3DC893D8} \\ \text{284953B9A4756611B1B61BFFCF53EDD9} \\ \text{79F93547DB714B06EF0A692062C609B7} \\ \text{0208AB8D4A280CEEE40ED8100F293063} \end{pmatrix}_{16}$$

标准 HMAC-SHA3 函数的输出结果为

$$\begin{pmatrix} \text{237A35049C40B3EF5DDD960B3DC893D8} \\ \text{284953B9A4756611B1B61BFFCF53EDD9} \\ \text{79F93547DB714B06EF0A692062C609B7} \\ \text{0208AB8D4A280CEEE40ED8100F293063} \end{pmatrix}_{16}$$

说明我们编写的 HMAC 函数与标准函数输出相同，证明了算法正确性。

4 生日攻击

4.1 算法原理

生日攻击是一种基于生日悖论的对 Hash 函数的一种碰撞攻击。这种攻击方式基于对固定排列数的随机攻击下发生碰撞的较高似然度。生日攻击可以在平均$\sqrt{2^n} = 2^{n/2}$次尝试内找到 Hash 函数的一个碰撞，而原像攻击的安全度为2^n。

生日问题可以描述如下：假设每个生日出现的概率相同，一年有 365 天，则使k个人中至少有两个人生日相同的概率大于0.5的最小k值是多少？答案是：若k个人中任意两人生日都不相同，则基本事件数为$\frac{365!}{(365-k)!}$，总的基本事件数为365^k，则该事件的概率为$\frac{365!}{(365-k)!365^k}$，存在至少两个人生日相同的概率显然为

$$P(365,k) = 1 - \frac{365!}{(365-k)!\,365^k}$$

当$k = 23$时，$P(365,23) = 0.5073$，即只需 23 人即可使至少有两个人生日相同的概率大于 0.5，当$k = 100$时，$P(365,100) = 0.9999997$，这比我们想象中的数字要大得多，故称为生日悖论。

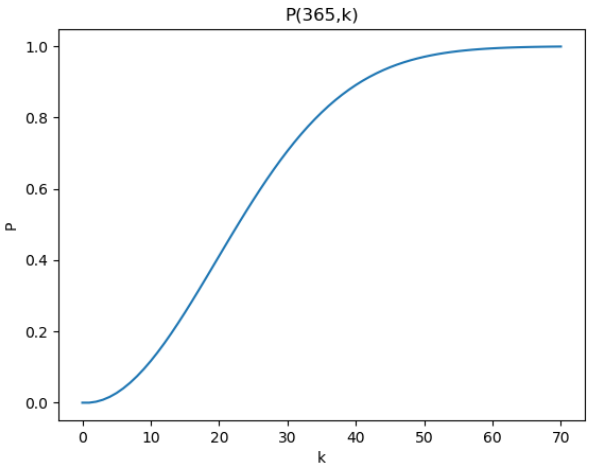

图 4 生日悖论

由生日悖论可以推广为下述问题：已知在一个1到n之间均匀分布的整数型随机变量，若该变量的k个取值中至少有两个取值相同的概率大于0.5，则k至少为多大？

与上述同理可得

$$P(n,k) = 1 - \frac{n!}{(n-k)!\,n^k}$$

利用不等式$1-x \le e^{-x}$可以得到

$$k = 1.18\sqrt{n} \approx \sqrt{n}$$

则将该结论用于 Hash 函数则可以得到，设 Hash 函数有2^m个可能的输出，即输出长度为m比特，如果 Hash 函数的k个随机输入中至少有两个产生相同输出的概率大于 0.5，则

$$k \approx \sqrt{2^m} = 2^{m/2}$$

寻找使 Hash 函数具有相同输出的两个任意输入的攻击方式被称为生日攻击。

4.2 算法实现的伪代码

生日攻击 寻找使 Hash 函数具有相同输出的两个任意输入

输入： Hash 函数H

输出： 碰撞值M_1, M_2

```
function birthdayAttack(H)
    dict ← {}
    while True do
        digest ← H(random_str)
        if digest in dict then
            return random_str, dict[digest]
        end if
        dict[digest] ← random_str
    end while
end function
```

4.3 测试样例及运行结果

为了使我们能够在有限的计算资源内得到 Hash 函数的碰撞结果，我使用了自定义的 SHA3-16 Hash 函数，该函数输出长度为 16 位，故存在$2^{16} = 65536$种不同的输出结果，根据上述生日攻击的结论，平均在$2^8 = 256$次尝试下即可得到一个碰撞，下面我们用实验验证该结论。

利用上述算法随机产生字符串，对 SHA3-16 函数进行 10 次生日攻击，结果如表 5 所示。

表 5 对 SHA3-16 函数的 10 次生日攻击结果

尝试次数	消息 A	消息 B
577	287.75799963230213	94.35128449083732
269	135.53423719925843	33.29584088644776
162	78.91598512833218	33.69807234254279

438	241.56637407217087	7.56615058482934
298	144.28068712931852	62.24889764108382
292	151.61558452108332	53.7634938851121
56	25.04504383193035	3.187031861577779
422	212.501025692241	153.6377513184485
210	101.27397710333751	38.450638779287665
591	294.0507092309667	1.5823209328350778

对上述数据绘制散点图，可视化后如图 5 所示。

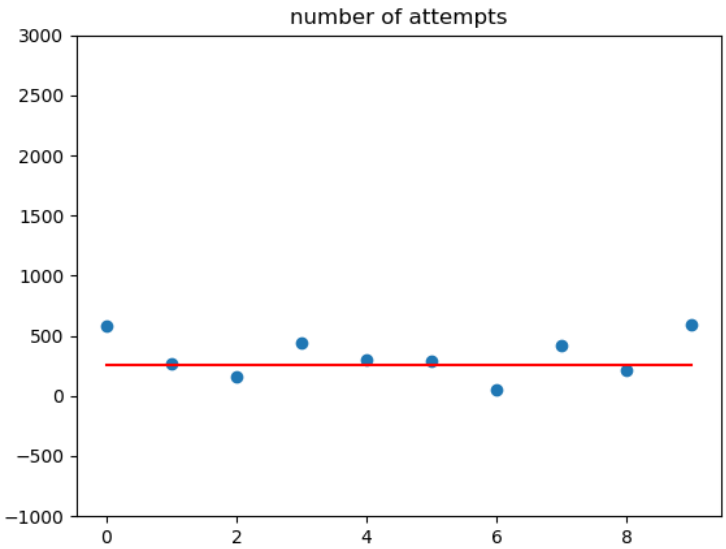

图 5 生日攻击尝试次数

可以看出，实验结果基本印证了生日攻击的结论，即平均在 256 次尝试下即可得到一组碰撞结果。

最后，为了验证碰撞是否成功，我们选取第一组碰撞消息计算其 Hash 值，结果如表 6 所示。

表 6 消息碰撞验证

消息字符串	287.75799963230213	94.35128449083732
Hash 值	0x7773	0x7773

两个消息的 Hash 值完全相同，证明碰撞结果是正确的。

5 消息变形算法

5.1 算法原理

Yuval 提出的基于生日悖论进行碰撞攻击的方法为：

(1) 发送方 A 准备对文本消息x进行签名，使用的方法是用 A 的私钥对m位的 Hash 码加密并将加密后的 Hash 码附于消息之后。

(2) 攻击者产生该合法消息x的$2^{m/2}$种变式x'，且每一种变式表达相同的意义，将这些消息以及对应的 Hash 值存储起来。

(3) 攻击者准备伪造一条消息y，并想获取 A 对y的签名。

(4) 攻击者再产生该伪造消息y的消息变式y'，每个变式y'与y表达相同的意义。对于每个y'，攻击者计算$H(y')$，并与任意的$H(x')$进行比对，重复这一过程直到碰撞出现。即这一过程直到找到一个y'与某个x'具有相同的 Hash 值。

(5) 攻击者将该合法消息x'提供给 A 签名，将该签名附于伪造消息的有效变式y'后并发送给预期的接收方。因为上述两个变式的 Hash 码相同，所以它们产生的签名也相同，因此攻击者即使不知道加密密钥也能攻击成功。

这样，如果使用 64 位的 Hash 码，那么所需代价的数量级仅为2^{32}。

产生多个具有相同意义的变式并不困难，一种简单的方法是在文件的词与词之间插入若干个“空格-空格-退格”字符对，然后在实例中用“空格-退格-空格”替代这些字符，从而产生各种变式。

我们设计了一种新的变形方法，通过向单词之间插入若干个“退格-空格”字符对的方式来对消息进行变形，为了使插入后的消息总长度尽量短，具体变形方法为：若原消息有n个空格字符，插入字符对的数量m从 1 开始不断增加，若插入m个字符对，则只需将m个字符对分散到n个单词之间的空格中即可。利用排列组合的知识可以得到，将m个相同字符对插入到n个不同的位置，共有C_{m+n-1}^{n-1}种方法。所以，对于有n个空格字符的消息，若要产生k个消息变形，只需找到满足以下式子的最小M即可。

$$1+\sum_{m=1}^{M} C_{m+n-1}^{n-1} \geq k$$

我们对上式进行简单的证明：首先考虑将m个相同字符对插入n个不同位置，且不存在没有字符对的位置，使用插板法，m个字符对有$m-1$个间隙，要分成n组，且不能有空组，只要在$m-1$个间隙中选取$n-1$个间隙，总的方法数为C_{m-1}^{n-1}；若可以产生空组，则先假设n个位置已经插好了 1 个字符对，也就是有$m+n$个字符对要插入n个不同的位置，方法数为C_{m+n-1}^{n-1}。令$m \leq M$，则产生的消息变形

为$1+\sum_{m=1}^{M} C_{m+n-1}^{n-1}$，其中 1 为原本的消息。

因为由k求M是较困难的问题，所以我们用画曲线图的方法对上式进行简单的观察，分别假设空格数量$n=5$和$n=10$，插入字符对最大数量M与消息变形数量k的关系如图 6 所示。

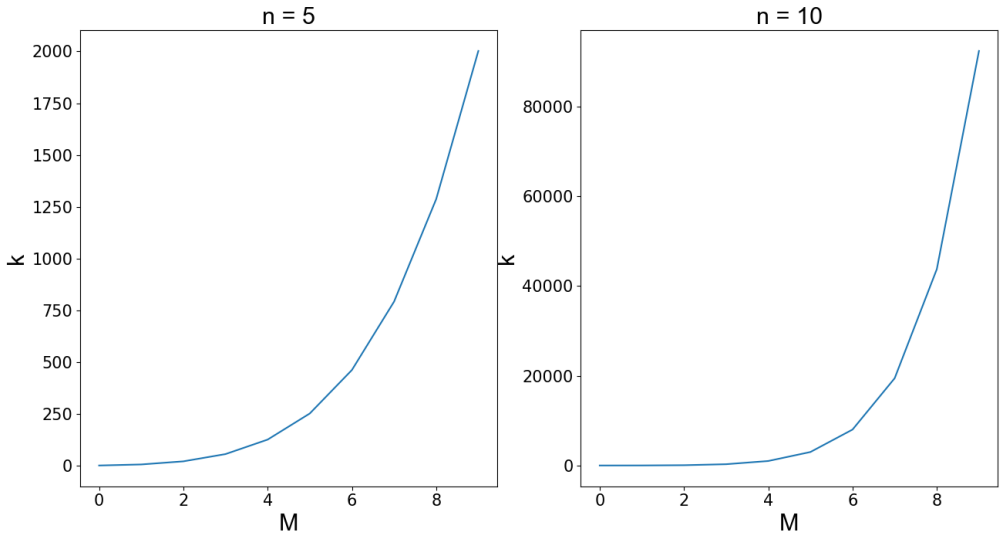

图 6 插入字符对数量与消息变形数量的关系

当字符串中有 10 个空格时，插入最多 9 个字符对，即可得到 92378 个消息变形，说明我们选择的消息变形方法具有很大的优势，用较少的字符对插入代价即可获得大量的消息变形。

5.2 算法实现的伪代码

消息变形算法 产生任意数量的消息变形

输入： 消息m，消息变形数d
输出： 变形后的消息数组l

```
function varyMsg(m, d)
    l ← {m}
    words ← split(m, " ")
    counter ← {(0, [0,0,⋯,0])}
    while len(l) < d do
        (k, count) ← pop(counter)
        for i ← k to len(words) − 2 do
            if len(l) = d then
                break
            end if
            count[i] ← count[i] + 1
            counter ← counter + {(i, count)}
            var ← words[0]
            for j ← 0 to len(words) − 2 do
```

```
                    var ← var + " " + "\b " * count[j] + words[j + 1]
                end for
                l ← l + {var}
                count[i] ← count[i] − 1
            end for
        end while
        return l
    end function
```

5.3 测试样例及运行结果

我们从短消息串“I like Python!”中产生 10 个消息变形，并使用 MD5 算法分别计算其 Hash 值来验证效果，结果如所示。

表 7 消息变形结果

消息字符串	MD5 值
I like Python!	82e3d15a68f615488565e1b44aaf9746
I \b like Python!	f1d8ce77c0c15e52c0f1963034518b57
I like \b Python!	b4b0044260537479d58a94ba40d98aca
I \b \b like Python!	a9f1286f30b3dd35c9c8d6a36ff22119
I \b like \b Python!	0d6fceccc38ca423428d4f58854217dc
I like \b \b Python!	452fb6e16b42a815c30a56693e996cf6
I \b \b \b like Python!	f32ab8f6008e7805581b430e68c42a24
I \b \b like \b Python!	eb99429b04cf90f1b98b5a7f849ec8c5
I \b like \b \b Python!	c9a7aa08b77eb558d5b21cc8d7082cab
I like \b \b \b Python!	b176ed7e2578e6a2c401f426e1be71cd

可以看到每个消息变形产生的 MD5 值都不相同，证明我们的算法可以被用作碰撞攻击中产生大量的消息变形。

6 总结

本次实验是实验内容较多的一次，分别实现了 SHA-1、SHA-3 密码学 Hash 函数和 HMAC 消息认证码，以及两个与 Hash 函数攻击问题相关的生日攻击和消息变形。通过本次实验，我对 Hash 函数及其攻击方法有了进一步的了解。

SHA-1 的实现较为简单，由于其设计时的结构也是考虑到方便软件实现，我们利用之前编写的 bitarray 类即可很方便地实现相应的操作，为了使我们设计的 Hash 函数与 Python 标准库的 Hash 函数操作相似，我们还实现了一个 Digest 类用于控制 Hash 值的显示，提供了字节串、十六进制和二进制的显示格式。

在 SHA-3 的实现过程中，我遇到了一些挫折。首先是 SHA-3 的状态张量较

难理解，其采用的列优先也是较为罕见的一种方式。在编写好整个函数功能后，我发现我的输出和标准有所差异，这时我重新仔细阅读了 FIPS 202 标准，以及在网上查阅了许多相关资料，发现 SHA-3 与原来的 Keccak 函数有一些细微的差异，主要体现在填充方式上，SHA-3 使用了一种字节对齐的方式进行填充。修改好填充方式之后，我发现结果依然不正确，这时我查看了 FIPS 202 标准附带的一些示例输入，经过逐比特的对照，发现 SHA-3 在存储数据采用了小端存储，即每个字节中最高有效位在低位，对于需要处理的消息，需要先进行小端转换，经过进一步的代码修改，我编写的 SHA-3 函数终于和标准函数有了一致的输出，在不断 debug 的过程中，我也对 SHA-3 的各个操作有了很深的记忆。

HMAC 的实现是本次实验中最简单的一个，因为其安全性主要由内部 Hash 函数提供，所以 HMAC 结构相对比较简单，实现过程中并没有遇到什么困难，最终也得到了与标准输出相一致的结果。

在实现生日攻击的过程中，我最初尝试对我编写的 SHA-1 函数进行攻击测试，发现耗时很长以至于得不到结果，初步分析可能是由于我编写的 SHA-1 函数实现上可能较慢，于是更换了标准库中的 SHA-1 函数，但是等待很长时间依然得不到结果。经过简单的分析，SHA-1 的输出长度为 160 位，如果实现生日攻击，则需要在2^{80}次尝试左右才可以得到碰撞结果，这是我们目前的计算性能达不到的，所以我想到利用 SHA-3 函数输出长度可变的特性，实现了一个 SHA3-16 函数，这样我们就可以在可接受的时间内对我们的生日攻击算法进行测试。

最后实现消息变形的过程中，我考虑了好几种不同的消息变形方案，但是有一些会产生与原消息相同的变形，所以淘汰掉这部分方案，最终筛选出一种可以在插入长度较短的情况下产生大量消息变形的较优方案，且不会产生相同的消息变形，经过一些数学上的证明与计算，这种方案的确可以产生高质量的消息变形，我们可以在今后使用这种方案尝试进行 Hash 函数的碰撞攻击。

总而言之，这次实验内容虽然消耗了我很多的精力，但同时也收获了许多，在高中我就曾接触过 Hash 函数，到现在自己亲手实现一次后，对 Hash 函数有了许多新的认识，希望将来在我的科研中也可以用到现在所学的这些内容。