密码学实验报告 9

张天辰 17377321

2019 年 5 月 30 日

1 SHA1 算法

1.1 算法原理

SHA1 算法的总体流程分为以下三部分：

1 填充与分组

首先在明文结尾填充 1，然后填充若干个 0，使其长度为 448 mod 512，此后再在明文结尾填充 64bit 的原始明文的长度，完成全部填充。然后将填充后的明文每 512bit 为一组，完成分组。

2 分组扩展

对于每一个 512bit 的明文分组，先将其每 32bit 分成小组，这样一共得到 16 组。此后根据这 16 组的值，用位操作继续扩展，直到最后总共得到 80 个小组。这 80 组将用于每个明文分组的操作之中。

3 计算 Hash 值

为了计算 Hash 值，首先要有 5 个 32bit 的链接向量，其拥有标准规定的初始值。对于每个明文分组，都要进行 4 轮每轮 20 步共 80 步的逻辑函数操作，在这 80 步中会用到其扩展出的 80 小组。每次操作结束后就把得到的值写入链接向量，并作为下一个明文分组的初始链接向量。在所有明文分组都进行过操作后，将 5 个链接向量组合就得到最终的 160bit 输出。其中每轮的逻辑函数是非对称的类 Feistel 密码结构，全部由位操作组成，因此易于在硬件实现，速度较快。

具体的操作会在下一节详述。

1.2 算法实现

定义“«<”符号为循环左移。

1. 轮函数

下面的算法中，CV 代表链接向量，其初始值为 0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, 0xC3D2E1F0。w 为明文分组扩展得到的 80 个小组。

```
def encode():
    A = CV[0]
    B = CV[1]
```

```
        C = CV[2]
        D = CV[3]
        E = CV[4]
        for round in range(20):
            temp = ((A <<< 5) +
                ((B & C) | ((B ^ 0xffffffff) & D)) +
                E + w[round] + 0x5A827999) & 0xffffffff
            E = D
            D = C
            C = B <<< 30
            B = A
            A = temp
        for round in range(20, 40):
            temp = ((A <<< 5) +
                (B ^ C ^ D) +
                E + w[round] + 0x6ED9EBA1) & 0xffffffff
            E = D
            D = C
            C = B <<< 30
            B = A
            A = temp
        for round in range(40, 60):
            temp = ((A <<< 5) +
                ((B & C) | (B & D) | (C & D)) +
                E + w[round] + 0x8F1BBCDC) & 0xffffffff
            E = D
            D = C
            C = B <<< 30
            B = A
            A = temp
        for round in range(60, 80):
            temp = ((A <<< 5) +
                (B ^ C ^ D) +
                E + w[round] + 0xCA62C1D6) & 0xffffffff
            E = D
            D = C
            C = B <<< 30
            B = A
            A = temp
        CV[0] = (CV[0] + A) & 0xffffffff
        CV[1] = (CV[1] + B) & 0xffffffff
        CV[2] = (CV[2] + C) & 0xffffffff
        CV[3] = (CV[3] + D) & 0xffffffff
```

```
47        CV[4] = (CV[4] + E) & 0xffffffff
```

2. 分组扩展

```
1   def setBlock(plain512):
2       w = [0] * 80
3       for i in range(16):
4           w[i] = plain512[i*4:(i+1)*4] to_bytes
5       for i in range(16, 80):
6           w[i] = (w[i - 3] ^ w[i - 8] ^ w[i - 14] ^ w[i - 16]) <<< 1
```

3. 其余内容

填充的过程比较简单，就是按照上面的逻辑进行即可。特别地，我用位运算实现了循环左移，方法如下：

```
1   def lshift(n, k):
2       return (n >> (32 - k)) + ((n & ((1 << (32 - k)) - 1)) << k)
```

1.3 算法测试

为了测试，对一个大小约为 75KB 的文件进行 Hash。为了检验算法正确性，我同时调用了 python 的 hashlib 库进行对照，并进行了计时比较。如图 1，16 进制小写的是 hashlib 的结果，16 进大写的是我的结果。我的算法是正确的，但是速度比 hashlib 库慢了很多。我不清楚 hashlib 的内部实现，或许调用了底层的一些内容。

```
87cd5eae739272c036642067206945afbed27249
Runtime is 0.0003771781921386719s
87CD5EAE739272C036642067206945AFBED27249
Runtime is 0.5683388710021973s
```

图 1: SHA1 算法测试

2 SHA3 算法

2.1 算法原理

SHA3 采用海绵结构，在对明文进行填充后进行分组，再对每一个分组尾部进行填充。初始化矩阵 S 为全 0 向量，对每个分组，首先将 S 异或该分组更新 S，然后将 S 转化为三维矩阵 A，并对 A 进行几个逻辑步函数，得到的输出再更新 S。再对每个分组都进行操作后，进入挤水阶段。每次将 S 中固定长度的子串取出，然后对 S 再进行一次步函数。如此往复直到取出的所有子串拼接在一起不小于需要的长度，就截取如此长度的子串输出。

SHA3 有几种算法：SHA3—224，SHA3-256，SHA3-384，SHA3-512。这几种算法除了参数略有不同以外，并没有很大差异。

2.2 算法实现

我在真正实现时使用了 python 的 numpy 相关方法，用于加速以及使代码简洁清晰。这里只给出逻辑。

1. SHA3 的几种算法的含义

```
def SHA3_224(M):
    return bits_to_hex(Keccak(448, M || [0, 1], 224))

def SHA3_256(M):
    return bits_to_hex(Keccak(512, M || [0, 1], 256))

def SHA3_384(M):
    return bits_to_hex(Keccak(768, M || [0, 1], 384))

def SHA3_512(M):
    return bits_to_hex(Keccak(1024, M || [0, 1], 512))

def Keccak(c, N, d):
    return sponge(1600 - c, N, d)
```

2. 海绵函数

```
def pad(x, m):
    j = (- m - 2) % x
    return [1] || [0] * j || [1]

def sponge(r, N, d):
    P = N || pad(r, len(N))
    n = len(P) // r
    c = b - r
    part = []
    for i in range(n):
        part.append(P[i*r:(i+1)*r])
    S = [0] * b
    zeroc = [0] * c
    for i in range(n):
        S = Keccak_p(S ^ (part[i] || zeroc))
    Z = []
    while True:
        Z = Z || S[:r]
        if d <= len(Z):
            return Z[:d]
        S = Keccak_p(S)
```

3. 每一步的操作

```
def Keccak_p(S):
    A = string_to_state(S)
    for ir in range(12 + 2 * l − nr, 12 + 2 * l):
        A = Rnd(A, ir)
    return state_to_string(A)

def Rnd(A, ir):
    return iota(chi(pi(rho(theta(A)))), ir)

def theta(A):
    C = A[:, 0, :] ^ A[:, 1, :] ^ A[:, 2, :] ^ A[:, 3, :] ^ A[:, 4, :]
    D = (Cx 方向循环右移 1) ^ (Cx 方向循环左移 1，z 方向循环右移 1)
    return A[x, y, z] ^= D[x, z]

def rho(A):
    A1 = 5 * 5 * w 全 0 矩阵
    A1[0][0] = A[0][0].copy()
    x, y = 1, 0
    for t in range(24):
        A1[x][y] = A[x][y] >>> ((t+1)*(t+2)//2)
        x, y = y, (2 * x + 3 * y) % 5
    return A1

def pi(A):
    A1 = 5 * 5 * w 全 0 矩阵
    for x in range(5):
        for y in range(5):
            A1[x][y] = A[(x+3*y) mod 5][x].copy()
    return A1

def chi(A):
    return A ^ ((Ax 方向循环左移 1 ^ (5 * 5 * w 全 1 矩阵)) & （Ax 方向循环左移 1）

def rc(t):
    if t mod 255 == 0:
        return 1
    r = [1, 0, 0, 0, 0, 0, 0, 0]
    for i in range(t mod 255):
        r = [0] + r
        r[0] ^= r[8]
        r[4] ^= r[8]
        r[5] ^= r[8]
```

```
            r[6] ^= r[8]
            r = r[:8]
        return r[0]

    def iota(A, ir):
        A1 = A.copy()
        RC = [0] * w
        for j in range(l + 1):
            RC[(1 << j)-1] = rc(j + 7 * ir)
        A1[0][0] ^= RC
        return A1
```

2.3 算法测试

或许是我的数据结构选取不当，我的 SHA3 算法速度极慢，只有 2～3KB 每秒的速度。尽管如此，通过与 hashlib 的检验，我的算法是正确的。我这里对一个 75KB 的文件进行 SHA3 的四种算法的测试，我只在程序里验证了 SHA3-224 的正确性（其余的正确性已经通过标准样例验证），如图 2所示。

```
d97f497a176efc176a60939a6196f7f85c2991acdadc263553e7e62d
0.0012531280517578125
D97F497A176EFC176A60939A6196F7F85C2991ACDADC263553E7E62D
26.0591459274292
E742F965581D62F24D3A2A4F9A4E935B586CA8A6D6340212DDCEFC2DA3E2BC73
A91F8296F6605C1C07CA6523BEA7C9147EE86A1984A3744637ADD270470FD6A1E376608DB755439B7455E444B62A0FFB
9A813DD51609084E84E478EF16671ED22C71DE7CDD473F566F8656C3466E429108176FBF69275A44349BBED3CA69B6DF53BD318CFF298A99A1D62D
5F771BF840
```

图 2: SHA3 测试

3 HMAC

3.1 算法原理

HMAC 的算法原理就是把密钥和 Hash 函数结合起来，实现消息认证的目的。其总体逻辑可以概括为以下表达式：

$$MAC(text) = HMAC(K, text) = H((K_0 \oplus opad)||H((K_0 \oplus ipad)||text))$$

3.2 算法实现

```
def mac(text):
    ipad = b'\x36' * B
    opad = b'\x5c' * B
    if len(key) > B:
        key = hashFunc(key)
    if len(key) < B:
        key = key.ljust(B, b'\x00')
```

```
        ip = ipad ^ key
        hipad = hashFunc(b''.join([ip, text]))
        op = opad ^ key
        return hashFunc(b''.join([op, hipad]))
```

3.3 算法测试

我验证了基于 SHA1 和 SHA3-512 的 HMAC，并将我的 HMAC 和 python 的 hmac 库内置函数进行比较，结果相同。如图 3，十六进制大写是我的结果，十六进制小写是内置库的结果，结果一致。

```
67ACC19B2DC6DDFEA3D3AEACE95C74C97A8D591A
67acc19b2dc6ddfea3d3aeace95c74c97a8d591a
6A0B2A85D73D5B991572A2DF707D242499D543B73D630FC6BFDA31E6F2C9C801A50209C24055BEC0B5127FFE1480FAEBB2C810044FF95C5D8A7766
442003832A
6a0b2a85d73d5b991572a2df707d242499d543b73d630fc6bfda31e6f2c9c801a50209c24055bec0b5127ffe1480faebb2c810044ff95c5d8a7766
442003832a
```

图 3: HMAC 测试

4 简单生日攻击

4.1 生日攻击原理

生日攻击是一种利用生日悖论攻击 Hash 函数的攻击。将真消息与假消息同时进行若干同义变换并输入进 Hash 函数，利用生日悖论有大概率可以找到其中的一对真假消息变形拥有同样的 Hash 值，从而达到伪造消息也能通过 Hash 验证的目的。本算法只实现用随机数的方式完成寻找 Hash 碰撞。

4.2 算法实现

算法利用 python 的字典实现。将 Hash 值作为字典的 Key，被 Hash 的消息作为 Key 对应的 Value。每当得到新的消息-Hash 对，就以新的 Hash 值为 Key 去访问字典，如果发生空 Key 的错误，就说明没有发生碰撞，则把这一对也加入字典并继续；否则说明有碰撞，直接返回碰撞值。利用 try-catch-else 的方法可以很好地实现上述算法。

```
  def attack(length):
     birth = {}
     for i in range(1 << 40):
        n = randint(0, 1 << L)
        h = hashFunc(n).hexdigest()[:length]
        try:
           birth[h]
        except KeyError as e:
           birth[h] = n
        else:
           return self.birth[h], n, i
```

4.3 算法测试

实际上，想寻找一对完全的 Hash 碰撞并不容易。这里只能截取 Hash 输出的前几位进行测试，并调用系统函数库，才能在有生之年看到结果。这里选择的测试结果也只能在十几秒内找到一对前 20bit 相同的 Hash 结果。如图 4，这里给出了 SHA1 算法发生局部碰撞的字节串和其 Hash 结果。

```
b'K\x0fYj\xa7\xff\xcb[\xec\x95\x13\x0fxL\xdf1U\x03\xa6\x82'
c0dcd5e91d9a19dc791b50a124d3f4e6e60f500b
b'\x86\xd9\xd6N\xeb\xee\xfa#\r\x84XrY\x921\x7f\xdb.\xc7\x01'
c0dcd5e91de1c802106766e0e153be5e546bf2c1
```

图 4: 简单生日攻击测试

5 感想

说实话，这种和系统内置库对比的实验任务，真的会一下子让那种“终于把算法调对”的喜悦消失殆尽。或许这种简单实现的算法在效率上就是比不上使用底层库的内置函数——甚至可以说是天壤之别。系统函数在千分之一秒内完成的任务我的算法需要用十几秒才能完成确实让人有挫败感。尽管如此，通过实验透彻理解算法确实让我受益匪浅。功不唐捐。