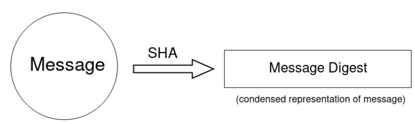
SHA-256

Saurav Kumar

December 21, 2018

1 Introduction

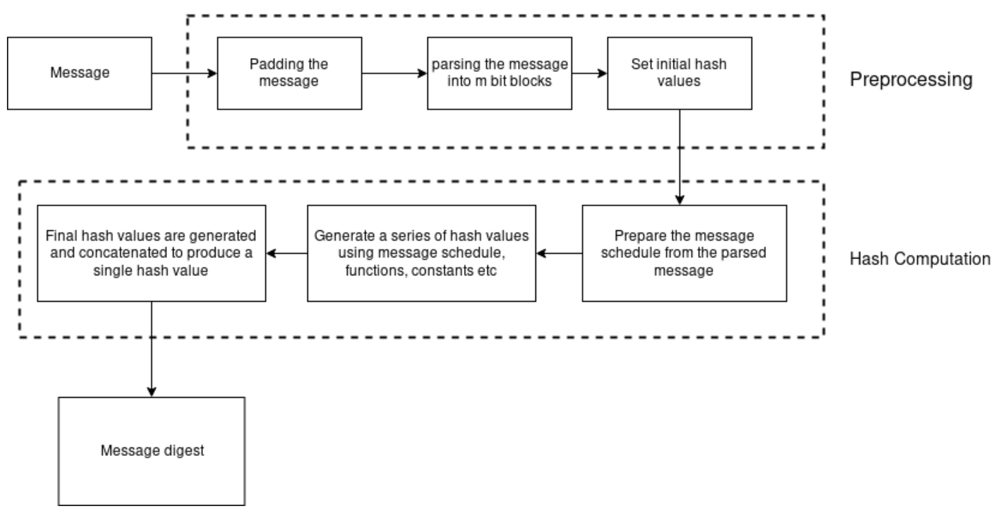
SHA: Secure hash algorithm
SHA algorithms are iterative and are one way hash functions(irreversible).



SHA enables the determination of message integrity. Any change in message will, with very probability result in a different message digest(Avalanche effect).

2 Stages of SHA

1. Preprocessing
2. Hash Computation



3 Key specs of SHA

1. Message Size: $< 2^{64}$
2. Block Size: 512
3. Word Size: 32
4. Message Digest Size: 256

4 Preprocessing

4.0.1 Padding the message

Padding is used to ensure the padded message is a multiple of 512 bits. Let the length of the message(M) is l bits. If l is not a multiple of 512 then,

1. Express message in binary values.
2. Append '1' to the message
3. Append k no. of zeros where k is the smallest, non-negative integer which ensures that l+k+1+64 is a multiple of 512.
4. Append the 64 bit block containing the length of the message,l expressed in binary form using big endian notation.

e.g. message ='abc' then, l = 8(ASCII is 8 bytes long)$*$3 = 24, k = 512-64-24-1 = 423

$$\underbrace{01100001}_{\text{“a”}}\quad \underbrace{01100010}_{\text{“b”}}\quad \underbrace{01100011}_{\text{“c”}}\quad 1\quad \overbrace{00\ldots00}^{423}\quad \overbrace{00\ldots0\underbrace{11000}_{\ell=24}}^{64}$$

4.1 Parsing the message

The message and its padding are parsed into N 512-bit blocks, $M^{(1)}, M^{(2)}, ..., M^{(N)}$. Since the 512 bits of the input block may be expressed as sixteen 32-bit words, the first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$.

4.2 Setting the initial hash values

For SHA-256, the initial hash value, $H^{(0)}$, shall consist of the following eight 32-bit words, in hex:

$$H_0^{(0)} = 6a09e667$$
$$H_1^{(0)} = bb67ae85$$
$$H_2^{(0)} = 3c6ef372$$
$$H_3^{(0)} = a54ff53a$$
$$H_4^{(0)} = 510e527f$$
$$H_5^{(0)} = 9b05688c$$
$$H_6^{(0)} = 1f83d9ab$$
$$H_7^{(0)} = 5be0cd19$$

5 Hash Computation

Notations:

1. Addition (+) is performed modulo 2^{32}.
2. $'\|'$ denotes concatenation of hex values.

Each message block, $M^{(1)}, M^{(2)}, ..., M^{(N)}$, is processed in order, using the following steps:

For i=1 to N:

{

1. Prepare the message schedule, $\{W_t\}$:

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

2. Initialize the eight working variables, $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{d}$, $\boldsymbol{e}$, $\boldsymbol{f}$, $\boldsymbol{g}$, and $\boldsymbol{h}$, with the (i-1)st hash value:

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

3. For t=0 to 63:

   {

$$T_1 = h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$$

$$T_2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

   }

4. Compute the i^{th} intermediate hash value $H^{(i)}$:

$$H_0^{(i)} = a + H_0^{(i-1)}$$

$$H_1^{(i)} = b + H_1^{(i-1)}$$

$$H_2^{(i)} = c + H_2^{(i-1)}$$

$$H_3^{(i)} = d + H_3^{(i-1)}$$

$$H_4^{(i)} = e + H_4^{(i-1)}$$

$$H_5^{(i)} = f + H_5^{(i-1)}$$

$$H_6^{(i)} = g + H_6^{(i-1)}$$

$$H_7^{(i)} = h + H_7^{(i-1)}$$

}

After repeating steps one through four a total of N times (i.e., after processing $M^{(N)}$), the resulting 256-bit message digest of the message, M, is

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$$

6 Implementation in Python 3

```python
def ROTR(x,n):
    """x is a 32 bit word"""
    return ((x>>n)|(x<<(32-n)))
```

```python

def SHR(x,n):
    return x>>n

def Ch(x,y,z):
    return (x&y)^(~x&z)

def Maj(x,y,z):
    return (x&y)^(x&z)^(y&z)

def Sigma0(x):
    return ROTR(x,2)^ROTR(x,13)^ROTR(x,22)

def Sigma1(x):
    return ROTR(x,6)^ROTR(x,11)^ROTR(x,25)

def sigma0(x):
    return ROTR(x,7)^ROTR(x,18)^SHR(x,3)

def sigma1(x):
    return ROTR(x,17)^ROTR(x,19)^SHR(x,10)

def SHA_256(M):
    l = len(M)*8 # 8 bit ASCII for each char
    M = ''.join('{0:08b}'.format(ord(x), 'b') for x in M)

    # initial hash values for SHA-256
    H0 = [0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
          0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19]

    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]

    # padding to ensure it's a multiple of the block size
    if l%512 != 0:
        k=0
        while (l+k+64+1)%512!=0:
            k=k+1
        M = M + '1' + k*'0' + '{0:064b}'.format(l)

    # parsing the message
    N = len(M)//512 # '//' is for int division else it'll return
    float
```

```python
    M = [M[i:i+512] for i in range(0, len(M), 512)] # splitting into N 512 bit blocks
    M_mat = [[]] # empty M_mat of N*16 dim
    for i in range(0,N):
        for j in range(0,512,32):
            M_mat[i].append(M[i][j:j+32])# splitting each i'th elem of M into 32 bit words each
    W=[]
    for i in range(0,N):
        # prepare the message schedule
        for t in range(0,64):
            if t<=15:
                W.append(int(M_mat[i][t],2))
            else:
                W.append((sigma1(W[t-2]) + W[t-7] + sigma0(W[t-15]) + W[t-16])%2**32)

        # initialize the eight working variables
        a = H0[0]
        b = H0[1]
        c = H0[2]
        d = H0[3]
        e = H0[4]
        f = H0[5]
        g = H0[6]
        h = H0[7]

        # compression function
        for t in range(0,64):
            T1 = (h + Sigma1(e) + Ch(e,f,g) + K[t] + W[t])%2**32
            T2 = (Sigma0(a) + Maj(a,b,c))%2**32
            h = g
            g = f
            f = e
            e = (d + T1)%2**32
            d = c
            c = b
            b = a
            a = (T1 + T2)%2**32

        # computation of next hash values
        H0[0] = (a + H0[0])%2**32
        H0[1] = (b + H0[1])%2**32
        H0[2] = (c + H0[2])%2**32
        H0[3] = (d + H0[3])%2**32
        H0[4] = (e + H0[4])%2**32
        H0[5] = (f + H0[5])%2**32
        H0[6] = (g + H0[6])%2**32
        H0[7] = (h + H0[7])%2**32
    message_digest='0x'
    for i in range(0,8):
        message_digest += hex(H0[i])[2:]
    return message_digest



print(SHA_256('abc'))
```

Output: 0xba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

It can be verified online using: https://emn178.github.io/online-tools/sha256.html

7 References

1. FIPS PUB 180-4, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf
2. SHA-2, https://en.wikipedia.org/wiki/SHA-2