DESCRIPTION OF THE LOGICS BEHIND SHA-256 WITH PYTHON IMPLEMENTATION

Table of Contents

1. INTRODUCTION

1.1 HASH FUNCTIONS

Cryptographic hash functions often referred to as one-way hash functions, are algorithms that receive a message of arbitrary length as input and transform it into a fixed length output, the hash code. They have proved to be one of the pillars of modern cryptography especially after the invention of public key cryptography in 1976, when they became an integral part of it. *(Al-Kuwari et al. 2010)*

For a hash function to be considered robust, it has to preserve three basic properties: collision resistance, pre-image resistance and second pre-image resistance. *(Gilbert et al. 2004)*

a) Collision resistance happens when two messages produce identical hash code. For a collision-resistant hash function, it should be computationally infeasible to find two messages that produce the same hash.

b) For a hash function to be pre-image resistant, once the hash is computed it should be computationally infeasible to retrieve the input message.

c) Given a hash function and a message M, for the function to be second pre-image resistant, it should be computationally infeasible to find another message M' with an identical hash value.

1.2 SHA-2 FAMILY HASH FUNCTIONS

The SHA-2 family of hash functions is standardized by NIST as a part of the Secure Hash Standard in FIPS 180-4. The standard defines two main algorithms, SHA-256 and SHA-512, with truncated variants SHA-224, which is based on SHA-256 and SHA-512/224, SHA-512/256, and SHA-384, based on SHA-512 algorithm. *(Dobraunig et al., 2015)*

The SHA2 family was first introduced by the National Institute of Standards and Technology (NIST) with the publication of FIPS PUB 180-2 in 2001. The publication became the new Secure Hash Standard, replacing FIPS PUB 180-1 in August 2002.

The Standard specified four secure hash algorithms, SHA-1, SHA-256, SHA-384, SHA512. Later in 2004, SHA-224 was added, and in 2014, with the introduction of the FIPS PUB 180-4, SHA-512/224 and SHA-512/256 were also added to the standard. All the algorithms are intended as iterative and one-way hash functions that could process a message to produce a so-called message digest.

Each algorithm enables the determination of a message's integrity: any change to the message will, with an awfully high probability, lead to a different message digest. These algorithms can be described in two stages: the preprocessing stage and the hash computation stage. The preprocessing part starts with padding the initial message, then the padded output is parsed into m-bit blocks, and in the end the initial hash value, that is going to be used in the hash computation, is set. On the other hand, the hash computation part generates a message schedule from the padded message and uses that schedule, together with logical functions, constants, and word operations to iteratively generate a series of hash values. The final hash value generated by the hash computation is used in determining the message digest. *(FIPS PUB 180-2, 2002)*

2. SHA-256 HASH FUNCTION

2.1 DESCRIPTION

One of the most important hash functions of the SHA-2 family is SHA-256. The compression function of the algorithm works on a 512-bit message block and a 256-bit intermediate hash value. Fundamentally, the SHA-256 hash function is a 256-bit block cipher algorithm that uses the message block as a key to encrypt the intermediate hash value. (*NIST, 2001)*

SHA-256 manages to preserve the three basic properties stated in the first section, making it one of the most secure operating hashing functions. The algorithm is computationally infeasible since a brute-force attack would need to make 2^{256} attempts to break it. Moreover, with a variety of 2^{256} possible hash values, it is extremely difficult to find messages with the same hash value. And finally, even a minor change to the initial message, just as simple as an addition or removal of a comma, alters the hash value so much that it's it is totally different from the previous hash. This property is also referred to as the avalanche effect.

As stated by the FIPS PUB 180-4 standard, SHA-256 may be used to hash a message, M, having a length of *l* bits, where $0 \le l \le 2^{64}$.
The algorithm uses a message schedule of sixty-four 32-bit words, eight working variables of 32 bits each, and a hash value of eight 32-bit words. Its final result is a 256-bit message digest. The words of the message schedule are labelled W_0, W_1,…, W_{63}, while the eight working variables are labelled a, b, c, d, e, f, g, and h. The words of the hash value are labelled $H_0^{(i)}$, $H_1^{(i)}$,…, $H_7^{(i)}$, which will hold the initial hash value, $H^{(0)}$, replaced by each successive intermediate hash value $H^{(i)}$, and ending with the final hash value, $H^{(n)}$. Furthermore, two temporary words are used; T_1 and T_2.

Basic Operations

SHA-256 uses several basic operations:

- four bitwise operations AND, OR, XOR and complement ($\wedge$, $\vee$, $\oplus$,$\neg$),
- addition modulo 2^w
- $ROTR^n(x)$ operation which denotes the circular right shift of n bits of the binary word x.
- $SHR^n(x)$ operation which denotes the right shift of n bits of the binary word x.

Logical Functions

SHA-256 uses six logical functions, where each function operates on 32-bit words, that are represented as x, y, and z. Each function produces a new 32-bit word. The functions are defined by the standard in the following way:

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$
$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\sum_0^{\{256\}}(x) = ROTR^{2}(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$$
$$\sum_1^{\{256\}}(x) = ROTR^{6}(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$$
$$\sigma_0^{\{256\}}(x) = ROTR^{7}(x) \oplus ROTR^{18}(x) \oplus SHR^{3}(x)$$
$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

Constants

SHA-256 uses the same sequence of sixty-four constant 32-bit words, $K_0^{(256)}$, $K_1^{(256)}$,…, $K_{63}^{(256)}$. To create these words we use only the first 32 bits extracted from the fractional parts of the cube roots of the first 64 prime numbers.

2.2 STEPS

As for all the SHA-2 algorithms, SHA-256 can be described in two stages: Preprocessing and Hash Computation.

Preprocessing

Preprocessing consists of three steps: padding the message M, parsing it into message blocks, and setting the initial hash value $H^{(0)}$.

a) The purpose of padding is to ensure that the padded message is a multiple of 512. We start by converting our message M into bits and then appending 1 bit to the end of it. Subsequently, a number of *k* zero bits is added. We consider *k* the smallest, positive solution to the equation $(l+1+k \equiv 448 \bmod 512)$. After that, we append the 64-bit block that is equal to the length *l* of the message expressed in bits. The length of

the padded message after this procedure is a multiple of 512 bits. In the python implementation, in this phase we will need to use a different code implementation for messages smaller than 448 bits, messages between 448 and 512 bits and messages longer than 512 bits.

b) The message and its padding then are parsed into N 512-bit blocks, $M_{(1)}$, $M_{(2)}$,…, $M_{(N)}$. Since the 512 can 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 $M_{15}^{(i)}$.

c) Before we start with the hash computation for each of the secure hash algorithms, we should set the initial hash value, $H^{(0)}$. This hash value consists of eight 32-bit words. To create these words, we first calculate the square roots of the first 8 prime numbers and then use only the first 32 bits of their fractional parts.

Hash Computation

The SHA-256 Hash Computation stage uses the previously defined logical functions and 32-bit constants. Each message block produced by the preprocessing stage, is handled using the following steps:

a) We start by creating the sixty-four 32-bit word message schedule W_t. The first words that are going to be added are the 16 original parsed blocks, while the remaining 48 words we are going to be created using the following recurrence:

$$\sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16}$$

b) Subsequently, we initialize the eight working variables, a, b, c, d, e, f, g, and h, with the $(i-1)^{st}$ hash value from the initial hash values list.

c) After creating the message schedule and initializing the eight working variables, we will perform 64 iterations of the compression function that is going to update the variables (a, b,…, h), with the help of two temporary words created using choose Ch and majority Maj functions, the constant words $K_t^{\{256\}}$, the message schedule W_t, and the eight working variables themselves.

$$T_1 = h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$$
$$T_2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$$
$$h = g$$
$$g = f$$
$$f = e$$
$$e = d + T_1$$
$$d = c$$
$$c = b$$
$$b = a$$
$$a = T_1 + T_2$$

d) After iterating the compression function for 64 times, an intermediate hash value $H^{(i)}$ is calculated.

Repeating all hash computation steps above for a total of N times (i.e., after processing $M^{(N)}$), we get a 256-bit message digest of the message, M, which is the union of all hashes.

3. TESTING THE HASH FUNCTION

To test the functionality of our algorithm implementation in python, I am going to encrypt several words and compare the results with the output produced by the SHA-256 algorithm implemented by the hashlib dependency.
The messages that are going to be tested are:

- “message digest”
- “ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789”
- "12345678901234567890123456789012345678901234567890123456789012345678901234567890”

As we can see the words selected above have different lengths, the first one is shorter than 448 bits, the second one has a length between 448 and 512 and the last one is longer than 512 bits. The aim is to test how our algorithm deals with messages of different length.
The results are as follows:

Message:
“message digest”
Our hash value:
f7846f55cf23e14eebeab5b4e1550cad5b509e3348fbc4efa3a1413d393cb650
Hashlib Hash value:
f7846f55cf23e14eebeab5b4e1550cad5b509e3348fbc4efa3a1413d393cb650

Message:
“ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789”
Our hash value:
db4bfcbd4da0cd85a60c3c37d3fbd8805c77f15fc6b1fdfe614ee0a7c8fdb4c0
Hashlib Hash value:
db4bfcbd4da0cd85a60c3c37d3fbd8805c77f15fc6b1fdfe614ee0a7c8fdb4c0

Message:
“12345678901234567890123456789012345678901234567890123456789012345678901234567890”
Our hash value:
f371bc4a311f2b009eef952dd83ca80e2b60026c8e935592d0f9c308453c813e
Hashlib Hash value:
f371bc4a311f2b009eef952dd83ca80e2b60026c8e935592d0f9c308453c813e

4. SUMMARY

The aim of this paper was to provide a simple theoretical explanation of the logics behind SHA-256 hash function and its implementation in python. The algorithm takes a message of arbitrary length and converts it into a fixed length hash code. The process of the hash function can be described in two stages: preprocessing and hash computation. Although, it is relatively simple to implement the algorithm in python, it has proven to be computationally infeasible and extremely resistant to collisions. However, it is vulnerable from both a dictionary and brute force attack, so it would be better if some salt was added.(Buchanan, 2017)

References

1. Al-Kuwari, S, Davenport, JH & Bradford, RJ 2010, Cryptographic hash functions: recent design trends and security notions. In *Short Paper Proceedings of 6th China International Conference on Information Security and Cryptology (Inscrypt '10).* Science Press of China, pp. 133-150.

2. Gilbert, H. and Handschuh, H., 2004. Security Analysis of SHA-256 and Sisters. Selected Areas in Cryptography, pp.175-193.

3. Dobraunig, C., Eichlseder, M. and Mendel, F., 2015. Analysis of SHA-512/224 and SHA-512/256. Advances in Cryptology – ASIACRYPT 2015, pp.612-630.

4. Secure Hash Standard. Federal Information. Processing Standards Publication 180-2. August 2002 (FIPS PUB 180-2)

5. NIST, "Descriptions of SHA-256, SHA-384, and SHA-512". May 2001

6. Secure Hash Standard. Federal Information. Processing Standards Publication 180-4. August 2015. (FIPS PUB 180-4)

7. Buchanan, W.J. 2017, Cryptography, River Publishers, Aalborg.

APPENDIX

This implementation is intended to explain the stages in which the hash function goes through.

In [1]:
```
# create three different messages
input_1 = "message digest"
input_2 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"
input_3 = "2345678901234567890123456789012345678901234567890123456789012345678901234567890"
message_input = input_1
```

Basic Operations

In [2]:
```
def XOR(i,j):
    return list(a^b for a,b in zip(i,j))
def AND(i,j):
    return [a and b for a, b in zip(i,j)]
def NOT(i):
    return [int(not(a)) for a in i]
def MAJ(i,j,k):
    return max([i,j,], key=[i,j,k].count)
def ROTR(x, n):
    return x[-n:] + x[:-n]
def SHR(x, n):
    return n * [0] + x[:-n]
def ADD(i, j):
    length = len(i)
    sums = list(range(length))
    #initial input needs a carry over bit as 0
    c = 0
    for a in range(length-1,-1,-1):
        #add the input bits with a double xor
        sums[a] = (i[a]^j[a])^c
        c = MAJ(i[a], j[a], c)
    return sums
```

Constant words calculation

In [3]:
```
# function for generating first N prime numbers
def generate_n_primes(N):
    primes  = []
    chkthis = 2
    while len(primes) < N:
        ptest    = [chkthis for i in primes if chkthis%i == 0]
        primes  += [] if ptest else [chkthis]
        chkthis += 1
    return primes
first_64_primes = generate_n_primes(64)

"""function for generating constant words that represent the first 32 bits
of the fractional parts of the cube roots of the first 64 prime numbers"""

K=[]
for x in first_64_primes:
    cube_root = x ** (1./3.)
    fractional = cube_root % 1
    fractional = hex(int(fractional * (2**32)))
    K.append(fractional)
```

Stages:

1. Preprocessing
2. Hash Computation

PREPROCESSING

Preprocessing consists of three steps:

1. The first step: padding the message
2. The second step: parsing the paded message
3. Third step: computing the initial hash value

In [4]:
```
# before we start, we convert our message first into unicode values then into bits
def message_to_bit(message):
    #string characters to unicode values
    unicode_val = [ord(c) for c in message]
    #unicode values to 8-bit strings (removed binary indicator)
    byte_string = []
    for val in unicode_val:
        byte_string.append(bin(val)[2:].zfill(8))# add bit "0" in front if bit length less than 8
    #8-bit strings to list of bits as integers
    bits = []
    for byte in byte_string:
        for bit in byte:
            bits.append(int(bit))
    return bits

# append k zeros function (k = length - bits -1)
def append_zeros(bits, length):
    l = len(bits)
    for i in range(l, length):
        bits.append(0)
    return bits                                                    # PADDING

                        #The purpose is to create a padded message that is a multiple of 512.
                                    #1. We start by converting our message to bits.
                                    #2. Caculate its length and express it in bits.
                                    #3. The result of the padded message will be:
                #The value of the message M in bits + 1 + k zero bits + 64-bit value of the length of the message.

def padded_message(message):
    # convert message into bits
    bits = message_to_bit(message)
    #calculate the message length as a 64-bit block
    message_len = [int(b) for b in bin(len(bits))[2:].zfill(64)]
    if len(bits) < 448:
        #append single 1
        bits.append(1)
        #append k zeros until it is congruent with 448mod512 bits
        bits = append_zeros(bits, 448)
        #add the 64 bits representing the length of the message
        bits = bits + message_len
        #return the block
        return [bits]
    elif 448 <= len(bits) <= 512:
        bits.append(1)
        #move to next message block
        bits = append_zeros(bits, 1024)
        #replace the last 64 bits of the multiple of 512 with the original message length
        bits[-64:] = message_len
        #return it in 512 bit chunks
        return parser(bits, 512)
    else:
        bits.append(1)
        # loop until multiple of 512
        while len(bits) % 512 != 0:
            bits.append(0)
        bits[-64:] = message_len
    return parser(bits, 512)

"""            If the message is shorter than 448 bits, append the 1 bit , add k 0 (until it reaches 448 bits)
                                        and then append the length expressed in bits.
             If message's length is between 448 and 512 bits, append the 1 bit then add k 0 (until it reaches 1024 bits),
                                then replace the last 64 bits with the length of the message.
          If the message is longer than 512 bits, append the 1 bit, then append 0 until we reach the nearest multiple of 512,
                                then replace the last 64 bits with the length of the message
"""


                                                    # PARSING

                    #While parsing we express our 512 bit padded message into 16 blocks with 32-bit each
# define the parser function
def parser(bits, block_length=8):
    parsed = []
    for i in range(0, len(bits), block_length):
        parsed.append(bits[i:i+block_length])
    return parsed
# parse the message in 32-bit blocks
for block in padded_message(message_input):
    w = parser(block, 32)

print("Number of M-bit blocks: ",len(w))
print("Number of bits per block: ",len(w[0]))



                                        # SETTING THE INITIAL HASH VALUE

                    #The first initial hash values are obtained by taking the first thirty-two bits
                    #of the fractional parts of the square roots of the first eight prime numbers.

initial_hash_list=[]
# generate first 8 prime numbers
first_8_primes = generate_n_primes(8)
print("First 8 prime numbers: ", first_8_primes)
# calculate the fractional parts of the square roots of the first eight prime numbers
for i in first_8_primes:
    square_root = i ** (1./2.)
    fractional = square_root % 1
```

```
    fractional = hex(int(fractional * (2**32)))
    initial_hash_list.append(fractional)
print("Initial Hash Value: ", initial_hash_list)
```

```
Number of M-bit blocks:  16
Number of bits per block:  32
First 8 prime numbers:  [2, 3, 5, 7, 11, 13, 17, 19]
Initial Hash Value:  ['0x6a09e667', '0xbb67ae85', '0x3c6ef372', '0xa54ff53a', '0x510e527f', '0x9b05688c', '0x1f83d9ab', '0x5be0cd19']
```

HASH COMPUTATION

Hash Computation consists of four steps:

1. The first step: preparing the message schedule
2. The second step: initializing the eight working variables
3. The third step: performing 64 iterations of the compression function
4. The fourth step: calculating the intermediate hash value

Hash computation

In [5]:
```
"""Before we start with the hash computation stage we first initilize the constants and initial hash value
and convert them in bits"""
# insert zeros at the beginning
def zeros_insert(bits,length):
    l = len(bits)
    while l < length:
        bits.insert(0, 0)
        l = len(bits)
    return bits

def init(values):
    #convert from hex to python binary string (with cut bin indicator ('0b'))
    binaries = [bin(int(v, 16))[2:] for v in values]
    #convert from python string representation to a list of 32 bit lists
    words = []
    for binary in binaries:
        word = []
        for b in binary:
            word.append(int(b))
        words.append(zeros_insert(word, 32))
    return words

h0, h1, h2, h3, h4, h5, h6, h7 = init(initial_hash_list)
k = init(K)



                                    #PREPARE THE MESSAGE SCHEDULE

                          #we start by adding to our initial 16 parsed blocks 48 blocks
for block in padded_message(message_input):
    w = parser(block, 32)
    for t in range(48):
        w.append(32 * [0])
    for t in range(16, 64):
        s0 = XOR(XOR(ROTR(w[t-15], 7), ROTR(w[t-15], 18)), SHR(w[t-15], 3) )
        s1 = XOR(XOR(ROTR(w[t-2], 17), ROTR(w[t-2], 19)), SHR(w[t-2], 10))
        w[t] = ADD(ADD(ADD(w[t-16], s0), w[t-7]), s1)



                                    #INITIALIZE THE EIGHT WORKING VARIABLES

                #we assign the working variables with the (i-1)st hash value from the initial hash values list
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4
    f = h5
    g = h6
    h = h7



                                    #COMPRESSION FUNCTION MAIN LOOP

            #Perform 64 iterations of the compression function that is going to update the variables (a, b,…, h),
    #with the help of two temporary words created using choose Ch and majority Maj functions, the constant words Kt{256},
                          #the message schedule Wt, and the eight working variables themselves
    for j in range(64):
        S1 = XOR(XOR(ROTR(e, 6), ROTR(e, 11)), ROTR(e, 25) )
        ch = XOR(AND(e, f), AND(NOT(e), g))
        temp1 = ADD(ADD(ADD(ADD(h, S1), ch), k[j]), w[j])
        S0 = XOR(XOR(ROTR(a, 2), ROTR(a, 13)), ROTR(a, 22))
        m = XOR(XOR(AND(a, b), AND(a, c)), AND(b, c))
        temp2 = ADD(S0, m)
        h = g
        g = f
        f = e
        e = ADD(d, temp1)
        d = c
        c = b
        b = a
        a = ADD(temp1, temp2)



                                    #CALCULATE THE INTERMEDIATE HASH VALUE

                            #Add the compressed chunk to the current hash value
    h0 = ADD(h0, a)
    h1 = ADD(h1, b)
    h2 = ADD(h2, c)
    h3 = ADD(h3, d)
    h4 = ADD(h4, e)
    h5 = ADD(h5, f)
    h6 = ADD(h6, g)
    h7 = ADD(h7, h)



                                    #PRODUCE THE FINAL HASH VALUE (big-endian)

                        #256-bit message digest of the message, M, is the union of all hashes

# function to convert the list of 32 bits into a hexadecimal value
def bit_to_hex(value):
```

```
    value = ''.join([str(x) for x in value])
    binaries = []
    for d in range(0, len(value), 4):
        binaries.append('0b' + value[d:d+4])
    hexes = ''
    for b in binaries:
        hexes += hex(int(b ,2))[2:]
    return hexes

digest = ''
for hash_value in [h0, h1, h2, h3, h4, h5, h6, h7]:
    digest += bit_to_hex(hash_value)
print("The hash value of the message ", message_input, " produced from our algorithm is: ", digest)
```

```
The hash value of the message  message digest  produced from our algorithm is:  f7846f55cf23e14eebeab5b4e1550cad5b509e3348fbc4efa3a14
13d393cb650
```

Test

In [6]:
```
import hashlib
encoded_my_name=message_input.encode()
digest_hlib = hashlib.sha256(encoded_my_name).hexdigest()
print("The hash value of the message ", message_input, "produced by the hashlib algorithm is: ", digest_hlib)
```

```
The hash value of the message  message digest produced by the hashlib algorithm is:  f7846f55cf23e14eebeab5b4e1550cad5b509e3348fbc4ef
a3a1413d393cb650
```

In [7]:
```
# Code Reference
#https://www.codespeedy.com/perform-xor-on-two-lists-in-python/
#https://blog.boot.dev/cryptography/how-sha-2-works-step-by-step-sha-256/
#https://stackoverflow.com/questions/16312730/comparing-two-lists-and-only-printing-the-differences-xoring-two-lists
#https://stackoverflow.com/questions/1628949/to-find-first-n-prime-numbers-in-python
```