

# Information Systems Security

## Mandatory TP 3 - SHA-256

November 10th, 2021

Submit on **Moodle** your Python file(s) **.py**, before **Tuesday, November 30th, 2021 at 11:59 pm (23h59)**.

Your code should be **commented**.

### SHA-256

The goal of this TP is to implement one of the variants of the hash function SHA-2, more precisely SHA-256.

All we need is a message, of any length, given as a string of characters, and two constants, given later. This will return a 256-bit hash.

### First Step : Padding

First, you need to pad your message (of length  $L$ ) to a multiple of 512 bits, following this algorithm :

- Add one bit of value 1 to your message, ✓
- Add  $K$  bits of value 0 to your message, with  $K \in \mathbb{N}$  the smallest non-negative integer such that  $L + 1 + K + 64$  is a multiple of 512, ✓
- Add  $L$  (length of the initial message in bits) as a 64 bits integer to your message. ✓

### Second Step : Merkle-Damgard structure

SHA-256 uses the Merkle-Damgard structure to create a fixed length output :

- Slice your message into 512-bit blocks,

- You have an initial value IV of 256 bits (it is a constant, given later in the TP),
- Give the 256-bit IV and the 512-bit block to SHA-256 one way compression function, which will compute a 256-bit output value  $h$ ,
- Then take this value  $h$  and the next 512-bit block of the message, and give them to the compression function, which will output a new 256 bit cipher,
- Repeat this process until all 512 bit blocks of the message have been used,
- The last 256 bit output is your hash.

### Third Step : Global outline of the one-way compression function

Now we just need the SHA-256 compression function, which is a one-way compression function taking two blocks, one of 512 bits (from the message) and one of 256 bits (The previous cipher, or IV in the first step), and returns a 256-bit hash.

This box works with 32 bit words. Additions are made modulo  $2^{32}$ .

- First, we need to create a list of 64 words (each one of 32 bits). Let us slice the 512 bit block from the message into 32 bit words, which will be  $W_1, W_2, \dots, W_{16}$ , the first 16 words of that list.

Then, the other words are defined with this pseudo-code. Be careful as there's both shifts and cyclic shifts : **rightrotate** is a rotation (cyclic shift) of the word to the right, **rightshift** is a shift (non cyclic) to the right :

For  $i$  from 17 to 64 :

$$s_0 = (w_{i-15} \text{ rightrotate } 7) \text{ xor } (w_{i-15} \text{ rightrotate } 18) \text{ xor } (w_{i-15} \text{ rightshift } 3)$$

$$s_1 = (w_{i-2} \text{ rightrotate } 17) \text{ xor } (w_{i-2} \text{ rightrotate } 19) \text{ xor } (w_{i-2} \text{ rightshift } 10)$$

$$w_i = (w_{i-16} + s_0 + w_{i-7} + s_1) \mod 2^{32}$$

- Then, we will do 64 iterations the compression part. That compression part needs 3 things :

An initial 256 bit hash, noted as eight 32 bit words  $h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7$  (this is either the previous hash, or the IV in the first round),

And two 32 words for each iteration  $i \in \{1, 2, \dots, 64\}$ .

The first one is  $W_i$ , and the second one is a constant  $K_i$  (that constant is different for each of the 64 iterations, and these constants are given later

in the TP).

This will give you eight 32 bit words, noted  $a, b, c, d, e, f, g, h$ .

The compression function is explained in the next subsection.

- Finally, after the 64 iterations defined in the next section, add the 256 bits (as eight 32 bit words, each time modulo  $2^{32}$ ) obtained with the initial hash (or IV) :

$$newh_0 = (h_0 + a) \mod 2^{32}$$

$$newh_1 = (h_1 + b) \mod 2^{32}$$

...

$$newh_7 = (h_7 + h) \mod 2^{32}$$

## Last Step : The Compression Part

And finally, we just need to define the compression part itself :

We start with the previous hash or IV as our initial eight words, noted  $a, b, c, d, e, f, g, h$ . The  $W_i$  and  $K_i$  are the words previously defined from the Third Step (see next subsection for constants  $K_i$ ).

Then, we follow this algorithm (rightrotate is again the rotation to the right (cyclic shift), and "xor", "not" and "and" are bitwise operators):

**All additions (+) are once again modulo  $2^{32}$ .**

For  $i$  from 1 to 64 :

$$X_1 := (e \text{ rightrotate } 6) \text{ xor } (e \text{ rightrotate } 11) \text{ xor } (e \text{ rightrotate } 25)$$

$$CH := (e \text{ and } f) \text{ xor } ((\text{not } e) \text{ and } g)$$

$$X_2 := (a \text{ rightrotate } 2) \text{ xor } (a \text{ rightrotate } 13) \text{ xor } (a \text{ rightrotate } 22)$$

$$MAJ := (a \text{ and } b) \text{ xor } (a \text{ and } c) \text{ xor } (b \text{ and } c)$$

$$temp1 := (h + X_1 + CH + K_i + W_i) \mod 2^{32}$$

$$temp2 := (X_2 + MAJ) \mod 2^{32}$$

$$h := g$$

$$g := f$$

$$f := e$$

$$e := (d + temp1) \mod 2^{32}$$

$$d := c$$

$$c := b$$

$$b := a$$

$$a := (temp1 + temp2) \mod 2^{32}$$

The last a,b,...,h you get after the 64 rounds are the ones used in the last point of the Third Step (global outline part).

## SHA-256 Constants

You need two constants, IV (256 bits) and the 64 words of 32 bits in K. These constants are already in the python file **"SHAConstants.py"** :

IV (256 bits) is defined in hexadecimal as the eight 32 bit words :

$$\begin{pmatrix} h0 := 0x6a09e667 \\ h1 := 0xbb67ae85 \\ h2 := 0x3c6ef372 \\ h3 := 0xa54ff53a \\ h4 := 0x510e527f \\ h5 := 0x9b05688c \\ h6 := 0x1f83d9ab \\ h7 := 0x5be0cd19 \end{pmatrix}$$

(which, fun fact, are defined by the first 32 bits of the fractional parts of the square roots of the first 8 primes numbers (from 2 to 19)).

The  $K_i$  (32 bit words) are defined in hexadecimal as :

$K[0..63] :=$

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

(fun fact part two, these are the first 32 bits of the fractional parts of the cube roots of the first 64 primes numbers (from 2 to 311))

## Examples

Here are some examples of what you should obtain, with the 256-bit hash expressed as an hexadecimal number (the examples are included in the **"SHA-Constants.py"** file) :

- For the empty string "", you should obtain the following hash :  
 $(e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855)_{16}$

- "Welcome to Wrestlemania!" gives you the following hash :  
(70eeb26f0052ebe0041e58d221e954c575f32a979cefdae7b761969e33b7934f)<sub>16</sub>
- "Fight for your dreams, and your dreams will fight for you!" :  
(31bba5997ae84193407798293636745b88d0126146fd105aa96e599c5f197714)<sub>16</sub>