Information Systems Security
Mandatory TP 1 - SHA-256 and Proof of Work

September 24th, 2025

Submit on **Moodle** your Python file(s) **.py**, before **Tuesday, October 14th, 2025 at 11:59 pm (23h59)**.

Your code should be **commented**.

TP : SHA-256 and Proof of Work

The goal of this TP is to :

1. **Implement SHA-256** with Python 3.
2. **Implement a proof of work that will resolve a "Bitcoin-like" mining challenge** : find a nonce for a given message that will give a resulting digest ending with at least 20 bits of value "0".

A file with useful constants and examples is also available on moodle.

SHA-256

To implement SHA-256, all we need is a message of any length, which will here be given as a string of characters, as well as two constants. Examples of messages with expected results, as well as the constants, are given later in this TP (constants are also included in the file "SHAConstants.py" given with this TP). The function will return a 256-bit digest.

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.

This gives you the following padded message :

Message	$\|$	1	$\|$	000...000	$\|$	L
L bits		*1 bit*		*K bits*		*64 bits*

With a total size which is a multiple of 512 bits.

Second Step : Merkle-Damgard structure

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

- Slice your padded 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 first 512-bit block to SHA-256 one way compression function, which will compute a 256-bit temporary 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 temporary output,
- Repeat this process until all 512 bit blocks of the message have been used,
- The last 256 bit output is your final digest.

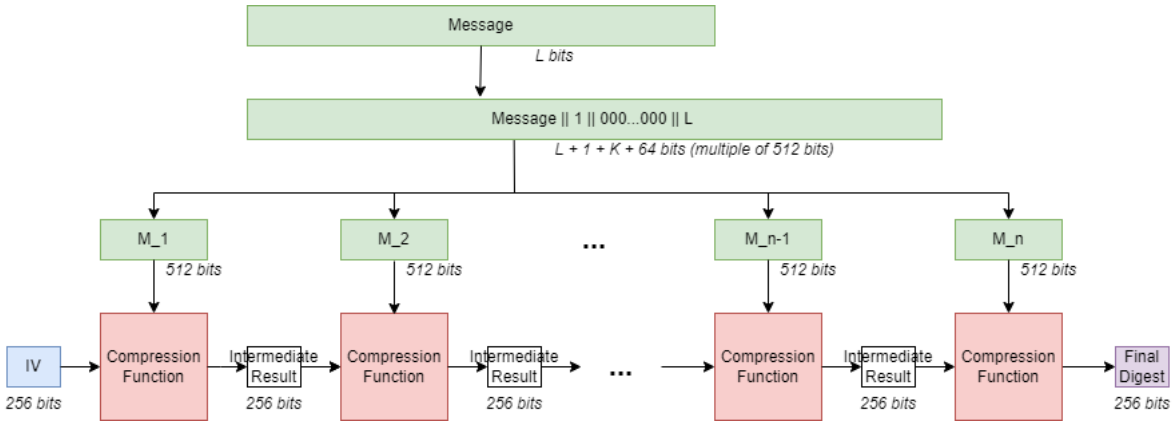

Figure 1: SHA-256 compression structure

Third Step : The one-way compression function

Now we need to explain how the SHA-256 compression function works. It is a one-way function, taking two blocks, one of 512 bits (from the message) and one of 256 bits (either the previous temporary output, or the IV if it's the first step), and returns a 256-bit output.

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 $W_1, W_2, ..., W_{16}$, 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 :

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

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

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

- Then, we will do 64 iterations of a given compression process :
  That compression process needs 3 inputs :
  - One input of 256 bits, noted as eight 32-bit words $h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7$ (this is the 256 bits in entry that are either the previous temporary output or the IV),

- And two 32-bit words for each iteration $i \in \{1, 2, ..., 64\}$.
For each iteration, the first of these two words is W_i (defined earlier and built from the 512 bits input from the entry), and the second one is a constant noted K_i (a constant different for each of the 64 iterations, and these constants are given later in the TP).

The compression process is as follows :

At the start of the first iteration, we initialise a,b,c,d,e,f,g and h with the values of $h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7$.

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 :

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

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

$$X_2 := (a \text{ rightrotate } 2) \text{ xor } (a \text{ rightrotate13}) \text{ xor } (a \text{ rightrotate } 22)$$

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

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

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

$$h := g$$

$$g := f$$

$$f := e$$

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

$$d := c$$

$$c := b$$

$$b := a$$

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

The last a,b,...,h you will get after the 64 rounds are then used in one last operation...

- ...where these last a,b,...,h are added mod 2^{32} with the initial values of $h_0, h_1, ...$ from before the 64 rounds. Here's the pseudo code for this :

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

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

...

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

And these eight new words of 32 bits are concatenated to give you the new 256-bit output (which is either the input for the next compression, or the final hash if it was the last compression).

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 and Expected Results

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

- For the empty string "", you should obtain the following hash : $(e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855)_{16}$

- "Welcome to Wrestlemania!" gives you the following hash :
  $(70eeb26f0052ebe0041e58d221e954c575f32a979cefdae7b761969e33b7934f)_{16}$
- "I will travel across the land
  Searching far and wide
  Teach Pokemon to understand
  The power that's inside" :
  $(14a4397570cbf8dc25c61e92050e7f6064fe3f101eb44486459520fd69b6e470)_{16}$

Proof of work

Now, your goal will be to find a given nonce such that the resulting digest has at least 20 bits at value 0 at the end (which can also easily be seen in hex form as 5 hex digits at value 0).

You will use the following message : your lastname, followed by a 64 bit nonce :

Lastname	‖	Nonce (64 bits)

The goal of your code is to find a nonce such that the resulting digest will finish with at least 20 bits at 0 (change the nonce value from 0 to 1, 2, ... until you find the desired digest).

Expected results

- Your SHA-256 function takes a message in input, as a string of characters, and outputs the digest in hexadecimal form (either hexadecimal result or string of hexadecimal).
- Test your function with all three examples given earlier (these examples are already provided as strings in the file given with this TP).
- As for the proof of work function, it should take your name (as a string), and do the proof of work (by trying multiple nonce values), and return the nonce value and digest once you find a digest with the desired form.

Your code should print clearly :

- For the SHA-256 part : **Print clearly each message and the corresponding digests.**
- For the proof of work : **print the message** (i.e. your name), **the nonce value, and the resulting digest** (which should have at least the last 20 bits at 0, i.e. in hexadecimal it should have the last 5 hexadecimal characters at 0).