Mohcine CHIHI

Coding Project – Cryptography

Implementation of SHA-256 in Python

Demonstration :

The code is organized into multiple helper functions and a main hashing procedure, all of which collectively implement the SHA-256 specification.

Functions :

rotate_right(value, shift, bits=32) :

This function rotates the bits of a 32-bit integer value to the right by shift positions. Rotations are used extensively in cryptographic hash functions to achieve non linearity.

```
def rotate_right(value: int, shift: int, bits: int = 32) -> int:
    return ((value >> shift) | (value << (bits - shift))) & 0xffffffff
```

sha256_ch(x: int, y: int, z: int) :

The "choose" function returns (x & y) ^ (~x & z). It selects bits from y or z based on the bits of x.

```
    #SHA256 choose
def sha256_ch(x: int, y: int, z: int) -> int:
    return (x & y) ^ (~x & z)
```

sha256_maj(x: int, y: int, z: int) :

The "majority" function (x & y) ^ (x & z) ^ (y & z) picks the majority bit value among x, y, and z at each position.

```
    #SHA256 majority
def sha256_maj(x: int, y: int, z: int) -> int:
    return (x & y) ^ (x & z) ^ (y & z)
```

sha256_bigsigma0(w), sha256_bigsigma1(w), sha256_smallsigma0(w) and sha256_smallsigma1(w) :

These functions apply a series of bit rotations and shifts to produce "sigma" values used in both generating the message schedule and updating the working variables. They are designed to spread and diffuse the input bits, increasing the algorithm’s security.

```python
def sha256_bigsigma0(w: int) -> int:
    #BigSigma0 : ROTR 2, 13, 22
    return (rotate_right(w, 2) ^ rotate_right(w, 13) ^ rotate_right(w, 22))

def sha256_bigsigma1(w: int) -> int:
    #BigSigma1 : ROTR 6, 11, 25
    return (rotate_right(w, 6) ^ rotate_right(w, 11) ^ rotate_right(w, 25))

def sha256_smallsigma0(w: int) -> int:
    #SmallSigma0 : ROTR 7, 18 and SHR 3
    return (rotate_right(w, 7) ^ rotate_right(w, 18) ^ (w >> 3))

def sha256_smallsigma1(w: int) -> int:
    #SmallSigma1 : ROTR 17, 19 and SHR 10
    return (rotate_right(w, 17) ^ rotate_right(w, 19) ^ (w >> 10))
```

pad_message(data) :

This function pads the original message to ensure its length in bits is congruent to 448 modulo 512, and then appends the original length in 64 bits. Proper padding ensures the message can be divided into fixed size 512-bit blocks as required by the SHA-256 specification.

```python
def pad_message(data) -> bytearray:
    """Perform SHA-256 padding on the input message."""
    if isinstance(data, str):
        data = bytearray(data, 'ascii')
    elif isinstance(data, bytes):
        data = bytearray(data)
    elif not isinstance(data, bytearray):
        raise TypeError("Expected str, bytes or bytearray")

    original_len = len(data) * 8
    #Append 0x80 (10000000 in binary)
    data.append(0x80)
    #Append 0x00 until the length (in bits) modulo 512 is 448
    while (len(data)*8 + 64) % 512 != 0:
        data.append(0x00)
    #Append the original length as a 64 bit big endian integer
    data += original_len.to_bytes(8, 'big')
    return data
```

prepare_schedule(chunk) :

For each 512-bit block (64 bytes) of the padded message, this function generates an array of 64 32-bit words. The first 16 words are extracted directly from the block, and the remaining 48 words are derived using the small sigma functions.

```python
def prepare_schedule(chunk: bytes) -> list:
    """Prepare the message schedule (64 words) for the given 512-bit chunk."""
    w = []
    #First 16 words come directly from the chunk
    for i in range(16):
        w.append(int.from_bytes(chunk[i*4:(i*4)+4], 'big'))
    #Remaining words are generated using the small sigma functions
    for i in range(16, 64):
        s0 = sha256_smallsigma0(w[i-15])
        s1 = sha256_smallsigma1(w[i-2])
        w.append((w[i-16] + w[i-7] + s0 + s1) & 0xffffffff)
    return w
```

Mohcine CHIHI

process_block(block: bytes, h: list) :

This function takes a 512-bit block and the current hash state (h) as inputs. It iterates through 64 rounds, updating temporary variables and applying the ch, maj, sigma functions, and the SHA-256 constants. At the end of processing each block, it updates the hash state (the array h) so that it accumulates changes block by block.

```
def process_block(block: bytes, h: list) -> None:
    """Process a 512-bit block and update the hash values in h."""
    w = prepare_schedule(block)
    a, b, c, d, e, f, g, hh = h

    #Main loop : 64 rounds
    for i in range(64):
        t1 = (hh + sha256_bigsigma1(e) + sha256_ch(e, f, g) + SHA256_CONSTANTS[i] +
        t2 = (sha256_bigsigma0(a) + sha256_maj(a, b, c)) & 0xffffffff
        hh = g
        g = f
        f = e
        e = (d + t1) & 0xffffffff
        d = c
        c = b
        b = a
        a = (t1 + t2) & 0xffffffff

    #Update the hash values
    h[0] = (h[0] + a) & 0xffffffff
    h[1] = (h[1] + b) & 0xffffffff
    h[2] = (h[2] + c) & 0xffffffff
    h[3] = (h[3] + d) & 0xffffffff
    h[4] = (h[4] + e) & 0xffffffff
    h[5] = (h[5] + f) & 0xffffffff
    h[6] = (h[6] + g) & 0xffffffff
    h[7] = (h[7] + hh) & 0xffffffff
```

sha256_modified(message) :

This is the main orchestrating function. It pads the message, breaks it into 512-bit blocks, calls process_block for each block, and finally combines the updated hash state into a 32-byte (256-bit) digest. This final digest is the SHA-256 hash of the input message.

Running the code :

Main :

```
if __name__ == "__main__":
    #Asking for input
    user_input = input("Enter a message : ")
    hash_value = sha256_modified(user_input)
    print("SHA-256:", hash_value.hex())
```

Input :

```
In [2]: runfile('C:/Users/mohci/CodingProject.py', wdir='C:/Users/mohci')

Enter a message :
```

I will try with my name.

Mohcine CHIHI

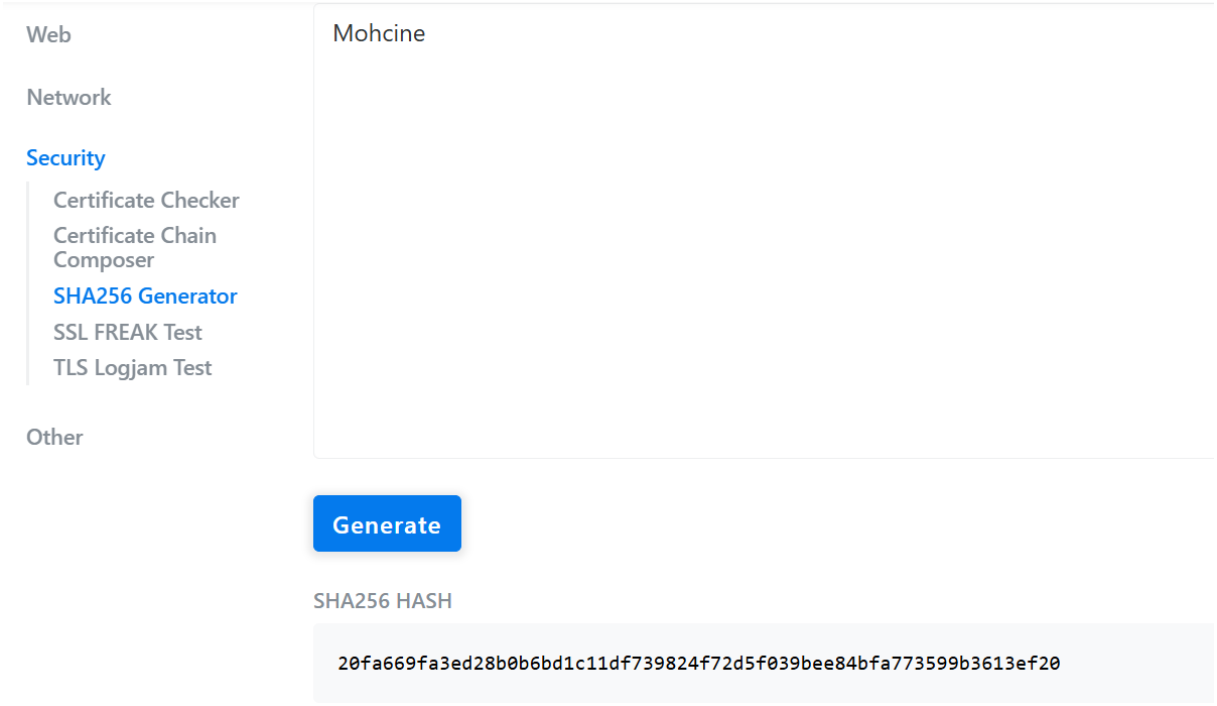
Output :

```
In [2]: runfile('C:/Users/mohci/CodingProject.py', wdir='C:/Users/mohci')

Enter a message : Mohcine
SHA-256: 20fa669fa3ed28b0b6bd1c11df739824f72d5f039bee84bfa773599b3613ef20

In [3]:
```

I will check with an online generator :



This generator produced the exact same output, confirming that our implementation is accurate.