Topics in Computer Science - Bitcoin: Programming the Future of Money - ITCS 4010 & 5010 - Spring 2025 - UNC Charlotte

Homework 2 - Cryptographic Hashing (30 Points)

Write Your Name Here: Hritika Kucheriya

Names of collaborators: Youtube, Stackoverflow

Submission instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
4. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, create a PDF version of the Jupyter notebook which includes visually all executed cells. This can be done in different ways depending on your specific Python/Jupyter setup. You can do that either by exporting into PDF (PDF via LaTeX / PDF via HTML), or by exporting into an HTML file first and then print the HTML site as a PDF and save that PDF.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly.
7. Submit **both** your PDF and the notebook file .ipynb on Gradescope.
8. Make sure your your Gradescope submission contains the correct files by downloading it after posting it on Gradescope.

1. Build SHA256 from scratch (25 Points)

In this exercise, you are asked to implement the algorithm applied in the SHA-256 hash function. The SHA-256 hash function is specified in the NIST standarization document FIPS PUB 180-4.

Disclaimer: Do not use the hashlib library for the entire Exercise 1.

Constants used in SHA-256

SHA256 employs a sequence of 64 constant 32-bit words, which can be considered as *fixed* random seeds to be used in the hash function. These values correspond to the first 32 bits of the fractional parts of the cube roots of the first 64 prime numbers.

They are represented in hexadecimal as shown below:

```
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
]
```

The hexadecimal encoding is to be read as follows: '0x' is a prefix that specifies that the following digits correspond to hexadecimal values, and subsequently, pairs of two hexadecimal digits correspond to a 8 bits of binary information. For example, the first of the 64 words above corresponds to the integer

```
K[1]
```

```
1899447441
```

and its hexadecimal and binary representations can be recovered via

```
hex(K[0])
```

```
'0x428a2f98'
```

and

```
bin(K[0])
```

```
'0b1000010100010100010111110011000'
```

respectively.

Initialization Vector of SHA-256

SHA-256 follows the Merkle-Damgård construction for constructing a variable-length input hash function based on a fixed-length *compression function*.

As such, SHA-256 requires the inclusion of initial hash values. These values are derived by taking the first 32 bits of the fractional parts of the square roots of the first 8 prime numbers. Expressed in hexadecimal respresentation, they can be defined as follows.

```
h0 = 0x6a09e667
h1 = 0xbb67ae85
h2 = 0x3c6ef372
h3 = 0xa54ff53a
h4 = 0x510e527f
h5 = 0x9b05688c
h6 = 0x1f83d9ab
h7 = 0x5be0cd19

H = [h0, h1, h2, h3, h4, h5, h6, h7]
```

a. Logic functions - Part 1 (2 Points)

Each of the following functions operates on 32-bit words (e.g., x, y, z), and the output of each function is a new 32-bit word.

Write functions `Ch` and `Maj` based on the defined specification:

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

Here, ⊕ corresponds to the bitwise "XOR" operation and "∧" corresponds to the bitwise "AND" operation.

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

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

import struct
```

b. Logic functions - Part 2 (1 Point)

Write function `ROTR(x,n)` that takes as inputs a `w`-bit word `x` and an integer `n` and implements a circular right shift such that, if n is an integer with $0 \leq n < w$, then output is

$$(x \text{ RIGHTSHIFT } n) \vee (x \text{ LEFTSHIFT}(w - n)).$$

Provide the length `w` as an optional input parameter that is chosen as 32 by default.

Here, x RIGHTSHIFT n for bit string x and an integer n corresponds to the operator of discarding the n rightmost bits of x, shifting the remaining ones by n to the right and padding the resulting bit string with n zeroes from the left.

```
def ROTR(x, n, w: int = 32):
   return (x >> n) | (x << (w - n)) & ((1 << w) - 1)
```

c. Logic functions - Part 3 (4 Points)

Write functions `Sigma1`, `Sigma2`, `Capsigma1` and `Capsigma2` that operate on a 32-bit word `x`.

1. `Capsigma1(x)` - Corresponds to the operation `ROTR(x,2)` ⊕ `ROTR(x,13)` ⊕ `ROTR(x,22)`.
2. `Capsigma2(x)` - Corresponds to the operation `ROTR(x,6)` ⊕ `ROTR(x,11)` ⊕ `ROTR(x,25)`.
3. `Sigma1(x)` - Corresponds to the operation `ROTR(x,7)` ⊕ `ROTR(x,18)` ⊕ (x RIGHTSHIFT 3) .
4. `Sigma2(x)` - Corresponds to the operation `ROTR(x,17)` ⊕ `ROTR(x,19)` ⊕ (x RIGHTSHIFT 10) .

```
def Capsigma1(num):
    num = ROTR(x, 2) ^ ROTR(x, 13) ^ ROTR(x, 22)
    return num

def Capsigma2(num):
    num = ROTR(x, 6) ^ ROTR(x, 11) ^ ROTR(x, 25)
    return num

def Sigma1(num):
    num = ROTR(x, 7) ^ ROTR(x, 18) ^ (x >> 3)
    return num

def Sigma2(num):
    num = ROTR(x, 17) ^ ROTR(x, 19) ^ (x >> 10)
    return num
```

Conversion of integer to byte representation

In Python, `to_bytes()` method can be used to convert an integer into its corresponding byte representation.

`Syntax: int.to_bytes(length, byteorder, *, signed=False)`

Here, length is the number of bytes that will be used to represent the integer. Basically, it defines the size of the resulting byte object.

Byteorder specifies the byte order, or endianness. In this exercise, we use 'Big' endian.

```
x = 1024
y = x.to_bytes(8, byteorder='big')
print(f'Byte representation of {x} is', y)
print('Number of bytes =', len(y))
```

```
Byte representation of 1024 is b'\x00\x00\x00\x00\x00\x00\x04\x00'
Number of bytes = 8
```

Conversion of bytes to integer

`from_bytes()` method can be used to convert a sequence of bytes into an integer.

`Syntax: int.from_bytes(bytes, byteorder='big', *, signed=False)`

```
# converts y (byte representation of 1024) back to integer
z = int.from_bytes(y, byteorder='big')
print(z)
```

```
1024
```

d. Preprocessing - Part 1 (4 Points)

Implement a function `padding` that appends bits to the input message so that the resulting message length is a multiple of 512 bits.

Steps:

1. Calculate message length: Let l be the message length in bits. For example, the message "abc" has l = 24 bits.
2. Add a "1" bit to the end of the message.
3. Append k zero bits, where k is the smallest number such that, for 'abc', l+1+k = (512-24)mod512. For 'abc' this means adding 423 zero bits
4. Append original length: Add the original message length l as a 64-bit binary value.

The total message length will now be a multiple of 512 bits.

```
def padding(message):
    if isinstance(message, str):
        message = bytearray(message, 'ascii')
    elif isinstance(message, bytes):
        message = bytearray(message)
    # Padding
    #calculate the message length in bits.
    length = len(message) * 8

    #Append 0x80 to the message
    message.append(0x80)

    #Append k zero bits
    while (len(message) * 8 + 64) % 512 != 0: #Add 64 because at the end we add 64-bit representation of the original
        message.append(0x00)

    #Add the original message length (length) as an 8-byte (64-bit) value to the end of the padded message.
    message += length.to_bytes(8, byteorder='big')
    if (len(message) * 8) % 512 != 0:
        print("Incomplete Padding")
    return message
```

Test your code: Padding

```
#test your padding function
msg = 'btc'
padded_msg = padding(msg)
if padded_msg == b'btc\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
    print("Well Done!, Testcase Passed")
else:
    Print("Testcase Failure")
```

```
Well Done!, Testcase Passed
```

e. Preprocessing - Part 2 (Points)

Complete function `parsing` that divides the padded message into N blocks, each 512 bits long.

```
def parsing(message):
    blocks = []
    #Append 64-byte (512-bit) chunk of the message to the blocks list.
    for i in range(0, len(message), 64): #512 bits = 64 bytes
       blocks.append(message[i : i + 64])
    return blocks
```

Test your code: Parsing

```
padded_msg = b'btc\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
blk = parsing(padded_msg)
if blk[0] == b'btc\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
    print("Well Done!, Testcase Passed")
else:
    print("Testcase Failure")
```

```
Well Done!, Testcase Passed
```

f. Message schedule (5 Points)

Complete the function `message_schedule` that prepares the message schedule for each parsed message block as given below:

- Iterate through 0 to 63.
    - For $0 \le t \le 15$, set schedule $W_t$ to the 32-bit words from the message block. (Append $W_t$ with 4 bytes (32-bit) at a time starting from the leftmost word)
    - For $16 \le t \le 63$, compute schedule $W_t$ using $W_t = \sigma_2(W_{t-2}) + W_{t-7} + \sigma_1(W_{t-15}) + W_{t-16}$, where $\sigma_1$ and $\sigma_2$ are `sigma1` and `sigma2` functions defined in the previous cell.

```
def message_schedule(block):
    schedule = []
    for t in range(0, 64):
        if t <= 15:
            #Append schedule list with 4 bytes at a time
            #Eg: if t = 0, append bytes(block[0:4]), if t = 1, append bytes(block[4:8]), if t = 2, append bytes(block|
            schedule.append(block[t*4:(t+1)*4])

        else:
            #Compute Sigma2() on the (t-2)th element of the schedule, converting the byte sequence to an integer first
            term1 = Sigma2(int.from_bytes(schedule[t-2], 'big'))

            # Retrieve the (t-7)th element from the schedule, convert it from bytes to an integer
            term2 = int.from_bytes(schedule[t-7], 'big')

            #Compute Sigma1() on the (t-15)th element of the schedule, converting the byte sequence to an integer firs
            term3 = Sigma1(int.from_bytes(schedule[t-15], 'big'))

            # Retrieve the (t-16)th element from the schedule, convert it from bytes to an integer
            term4 = int.from_bytes(schedule[t-16], 'big')

            temp = ((term1 + term2 + term3 + term4) % 2**32).to_bytes(4, 'big')
            schedule.append(temp)
    return schedule
```

```
sch = message_schedule(blk[0])
if sch[10] == b'\x00\x00\x00\x00':
    print("Well Done!, Testcase Passed")
else:
    print("Testcase Failure")
```

```
Well Done!, Testcase Passed
```

g. Hash computation (7 Points)

Complete the function `sha256` that computes the hash for a given message.

Steps:

1. Each message block is processed in order: Iterate over N message blocks.
2. Prepare the message schedule.
3. Initialize the working variables (a,b,c,d,e,f,g,h) with initial hash values (h0, h1, h2, h3, h4, h5, h6, h7) respectively.
4. Iterate through 0 to 63 and compute the following:

$T_1 = h + \sum_2(e) + \text{ch}(e, f, g) + K_t + W_t$

$T_2 = \sum_1(a) + \text{maj}(a, b, c)$

$h = g$

$g = f$

$f = e$

$e = d + T_1$

$d = c$

$c = b$

$b = a$

$a = T_1 + T_2$

5. After completing the 64 rounds for a message block, update the i^{th} intermediate hash values using the working variables such that;

   $h0 = (h0^{i-1} + a), h1 = (h1^{i-1} + b), h2 = (h2^{i-1} + c), h3 = (h3^{i-1} + d), h4 = (h4^{i-1} + e), h5 = (h5^{i-1} + f), h6 = (h6^{i-1} + g), h7 = (h7^{i-1} + h)$

6. After processing all N blocks, the final 256-bit message digest is the concatenation of the updated hash values

   h0 + h1 + h2 + h3 + h4 + h5 + h6 + h7

Note: To ensure the result stays within the bounds of a 32-bit integer, apply modulo 2^{32} to the sum.

```
def sha256(message, H, K):
    #Initial hash values
    h0, h1, h2, h3, h4, h5, h6, h7 = H[0], H[1], H[2], H[3], H[4], H[5], H[6], H[7]

    #Preprocess the message
    #padding
    message = bytearray(message, 'utf-8')
    orig_len_in_bits = (8 * len(message)) & 0xFFFFFFFFFFFFFFFF
    message.append(0x80)

    while (len(message) * 8) % 512 != 448:
        message.append(0x00)

    message += struct.pack('>Q', orig_len_in_bits)

    #Parsing
    blocks = [message[i:i+64] for i in range(0, len(message), 64)]

    for block in blocks:
        #prepare message schedule
        schedule = list(struct.unpack('>16L', block)) + [0] * 48

        # Initialize working variables a,b,c,d,e,f,g,h using h0,h1,h2,h3,h4,h5,h6,h7 respectively
        #Your code here
        a, b, c, d, e, f, g, h = h0, h1, h2, h3, h4, h5, h6, h7

        # Iterate for t=0 to 63
        for t in range(64):
            # t1 = (h + Capsigma2(e) + Ch(e, f, g) + constant K[t] + (schedule[t], converting bytes to an integer)), m
            t1 = (h + Capsigma2(e) + Ch(e, f, g) + K[t] + schedule[t]) & 0xFFFFFFFF

            # calculate t2 based on Capsigma1(a) and Maj(a, b, c), mod 2^32
            t2 = (Capsigma1(a) + Maj(a, b, c)) & 0xFFFFFFFF

            # Update the values for h, g, f
            #Your code here
            h = g
            g = f
            f = e

            # e: add t1 to d and take the result mod 2^32
            e = (d + t1) & 0xFFFFFFFF

            # Update the values for d, c, b
            d = c
            c = b
            b = a

            # a: sum of t1 and t2, mod 2^32
            a = (t1 + t2) & 0xFFFFFFFF

        # Compute intermediate hash value
        # eg: h0 = (h0 + a) % 2**32
        # Your code here
        h0 = (h0 + a) & 0xFFFFFFFF
        h1 = (h1 + b) & 0xFFFFFFFF
        h2 = (h2 + c) & 0xFFFFFFFF
        h3 = (h3 + d) & 0xFFFFFFFF
        h4 = (h4 + e) & 0xFFFFFFFF
        h5 = (h5 + f) & 0xFFFFFFFF
        h6 = (h6 + g) & 0xFFFFFFFF
        h7 = (h7 + h) & 0xFFFFFFFF

    return (((h0).to_bytes(4, 'big') + (h1).to_bytes(4, 'big') +
            (h2).to_bytes(4, 'big') + (h3).to_bytes(4, 'big') +
            (h4).to_bytes(4, 'big') + (h5).to_bytes(4, 'big') +
            (h6).to_bytes(4, 'big') + (h7).to_bytes(4, 'big')).hex())
```

```
message = "bitcoin"
print(sha256(message, H, K))
```

```
f6e0865b815ae0d2b990610760988f6006757081b5518f6009652c88d2a75609
```

```
message = "lightning"
print(sha256(message, H, K))
```

```
5d54787b48ad0b85d26db42fe47fbc941db501864ebc4bc9f95b49cb31629775
```

2. SHA256 using hashlib library (5 Points)

a. Hashlib - sha256 (5 Points)

write the `hash_string` function that uses sha256 from `hashlib` library to convert input string to hash.

- Convert the input string to bytes (use .encode())
- Use sha256 function of hashlib to get the hash from the input string bytes.
- Return the hashed text in hex.

```
import hashlib
def hash_string(message):
   message_bytes = message.encode()
   text = hashlib.sha256(message_bytes)
   return text.hexdigest()
```

```
message = "bitcoin"
print(hash_string(message))
```

```
6b88c087247aa2f07ee1c5956b8e1a9f4c7f892a70e324f1bb3d161e05ca107b
```

```
message = "lightning"
print(hash_string(message))
```

```
01db71ab8048f74a4b92c26ba77285ade0687ac192758e8185ad52701f649ef2
```

```
len(hash_string(message)*4)
```

```
256
```

Verify SHA-256 implementation against Python's hashlib library

Compare the output of your SHA-256 implementation with the output of the hashlib SHA-256 function. Both should produce identical results.

```
message = 'satoshi'
print(sha256('satoshi', H, K))
print(hash_string(message))
print(sha256('satoshi', H, K) == hash_string(message))
```

```
e3a2787392f055461f364b5351e7e818e5365a4d781b2548d422ad08994b100d
da2876b3eb31edb4436fa4650673fc6f01f90de2f1793c4ec332b2387b09726f
False
```