Step by Step

Hash algorithm SHA-1 and SHA-256

By Nipat Chapakdee

สารบัญ

ReadMe

Source code ที่ อยู่ในเอกสารฉบับนี้ สามารถหาได้ทั่วไปตามอินเทอร์เน็ต เนื่องจาก SHA-1 และ SHA-256 เป็น Algorithm ที่รู้จักกันในวงกว้างจึงมีการเปิดเผย pseudocode หรือแม้กระทั่ง Source code อยู่มากมายตามเว็บต่างๆ หากต้องการแหล่งอ้างอิงที่หน้าเชื่อถือ ควรศึกษาเอกสารผู้สร้าง Algorithm ตามลิงค์นี้เลย http://dx.doi.org/10.6028/NIST.FIPS.180-4

การจัดทำเอกสารครั้งนี้ ไม่ได้จะเอางานของผู้อื่นมาแอบอ้างเป็นงานของตัวเองแต่อย่างใด ผมเพียงหยิบมาอธิบายกระบวนการทำงาน ในแต่ละบรรทัด ในแต่ละ function ในรูปแบบ ภาษาไทย เพื่อให้ผู้อ่านสามารถเข้าใจการทำงานของ SHA-1 SHA-256 Algorithm ได้ง่ายขึ้นไม่มากก็น้อย

ความรู้พื้นฐานที่ผู้อ่านควรมี คือ Bitwise operators, binary number system และ hexadecimal number system เนื่องจาก SHA Algorithm เป็นการทำงานในระดับ bit จะใช้เรื่องพวกนี้ค่อนข้างเยอะพอสมควร หากผู้อ่านยังไม่เข้าใจเรื่องเหล่านี้ ก็เป็นไปได้ยากที่จะทำความเข้าใจ SHA Algorithm

Introduction

Hash algorithm คือ ฟังก์ชันทางคณิตศาสตร์ที่รับข้อมูลเข้าและสร้างเอาต์พุตขนาดคงที่ ซึ่งเรียกว่าค่าแฮชหรือการแยกย่อยข้อความ hash algorithm ยังเป็น algorithm ที่เข้ารหัสทางเดียวอีกด้วย หมายความว่า จะไม่สามารถนำ output ย้อนกลับไปหา input ได้ หลักๆ แล้ว hash algorithm เหมือนกันตามที่กล่าวไป แต่จะแตกต่างกันแค่กระบวนการข้างใน และขนาดของ output

SHA-1 และ SHA-256 แบ่งออกเป็น 2 ส่วนใหญ่ๆ คือ ส่วนขยายข้อความ (Padding the Message) และส่วน Algorithm ซึ่งส่วนขยายข้อความ (Padding the Message) ก็มีแบ่งออกเป็นกลุ่มๆ ตามผลทวีคูณใน Algorithm จะมี 512-bit และ 1024-bit ซึ่ง SHA-1 และ SHA-2 ในกลุ่มด้วยกันคือ 512-bit จึงทำให้ส่วนขยายข้อความ (Padding the Message) นั้นเหมือนกัน

ส่วนขยายข้อความ (Padding the Message) เป็นส่วนที่แปลงข้อความให้มีขนาดตามผลทวีคูณใน algorithm

Padding the Message

ตามที่กล่าวไปข้างต้นว่า SHA-1 และ SHA-256 อยู่ในกลุ่มเดียวกันคือ 512-bit ดังนั้น ในส่วน Padding the Message ก็ต้องเตรียมข้อความให้ได้ 512-bit โดยมีหลักการดังนี้คือ แปลง input ให้อยู่ในรูปของ binary จากนั้น เพิ่ม “1” เข้าไปที่ตัวสุดท้ายของข้อความ นำข้อความที่ผ่านการเพิ่ม “1” ต่อท้ายแล้วมา mod 448 และเติม “0” ไปเรื่อย ๆ จนขนาดเท่า 448-bit หรือ mod 448 = 0 สุดท้ายเราจะบวกด้วย 64-bit ที่เหลือ โดยการนำความยาวของ input มาแปลงเป็นเลข binary จบกระบวนการ Padding the Message จะได้ข้อความที่มีขนาด 512-bit พอดี

กระบวนก่อนจะนำไปบวก 64-bit สุดท้าย จะเป็นไปตามสมการนี้ $l + 1 + k \equiv 448 \bmod 512$ โดยที่

l คือ ความยาวของ input

k คือ จำนวน “0” ที่เติมเข้าไป

k สามารถคำนวณได้จาก ดังนี้ k = 448 – (ความยาวของ input ก่อนแปลงเป็น binary + 1)

** อธิบายสมการ $l + 1 + k \equiv 448 \bmod 512$ เพิ่มเติม **

l + 1 + k จะต้องมีค่าที่ mod 448 แล้วเท่ากับ 0 เหตุที่ไปสมมูลกับ 448 mod 512 ก็เพราะ 448 mod 512 ได้ 64 ซึ่งความหมายของมันคือ เมื่อเราได้ค่า l + 1 + k มาแล้ว เราต้องนำไปบวกอีก 64-bit ที่เหลือเพื่อให้จบกระบวนการ

64-bit สุดท้าย จะคำนวณโดย นำความยาวของ input หลังแปลงเป็น binary แล้ว มาแปลงเป็น เลข binary ขนาด 64-bit เช่น input คือ “Cat”

แปลง เป็น binary จะเท่ากับ 01000011 01100001 01110100 ความยาวที่ได้คือ 24 ตัว คิดง่ายๆ จะได้ 1 character เท่ากับ 8 bit ในตัวอย่างมี 3 character นำ 3×8=24 ดังนั้น 64-bit สุดท้ายที่จะเพิ่มเข้าไป คือ

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00011000

ตัวอย่าง Padding the Message

Input => “Cat”

l = 3

k = 448 – (3 +1) => 444

64-bit สุดท้าย = 00000000 00011000

Output หลังผ่าน Padding the Message จะได้

b' Cat / 1 / 00 .... 00 / 00000000 ..... 00011000'

binary	ค่าคงที่	0 442 ตัว	24 ในรูปแบบ binary 64 digits
3-bit	1-bit	442-bit	64-bit

3 + 1 + 444 + 64 = 512-bit

Source code Padding the Message

```
def Padding_the_Message(message):
    message = bytearray(message, 'utf-8')
    bit = len(message) * 8
    message.append(0x80)
    while len(message) % 64 != 56:
        message.append(0)
    message += bit.to_bytes(8, byteorder='big')
    return message
```

คำอธิบาย

```
message = bytearray(message, 'utf-8')
```

Function นี้ รับ input เข้ามาแล้วแปลงเป็น bytearray encoding = utf-8 ผลที่ได้คือ array input ที่ผ่านการเข้ารหัส utf-8

```
 byte = len(message) * 8
```

ซึ่งการเข้ารหัส character นั้น 1 character = 1-byte หากเราอยากได้ความยาวที่เป็น bit ก็แค่ คูณด้วย 8

```
message.append(0x80)
```

ต่อมา เพิ่ม “1” ต่อท้ายข้อความ เพื่อจะทำให้ดูง่ายขึ้น แปลงเป็นเลขฐาน 16 ขนาด 2 digits ซึ่ง 0x เป็นตัวบ่งบอกว่าเลขที่อยู่ข้างหลังต่อไปนี้จะเป็นเลขฐาน 16

```
while len(message) % 64 != 56:
        message.append(0)
```

วน loop เพิ่ม “0” จนกว่าจะ mod 64 แล้วได้ค่าเท่ากับ 56 เนื่องจากเราเก็บข้อมูลเป็น byte ตัวเลขที่นำมา mod จึงต้องหาร 8 ** ข้อสังเกต 64 * 8 = 512 **

```
message += bit.to_bytes(8, byteorder='big')
```

สุดท้ายเราจะเพิ่ม 64-bit สุดท้ายด้วยค่าความยาวของ input หลังจากแปลงเป็น binary แล้ว 8 ตัวนั้นคือขนาด byte ซึ่ง 8-byte = 64-bit

```
return message
```

return ข้อความสุดท้ายที่ได้หลังผ่านกระบวนการทั้งหมด message จะอยู่ในรูปของ bytearray

SHA-1

SHA-1 มี output ขนาด 160-bit โดยมี input ที่เข้ามา ขนาด 512-bit (input หลังผ่าน Padding the Message) input ที่เข้ามาใน SHA-1 algorithm ผ่านการคำนวณ ทั้ง 80 ×32-bit ต่อ word ซึ่ง 1-word ค่าเท่ากับ 4-byte หรือ 32-bit โดยในแต่ละรอบการคำนวณ จะมีการบวก, and logical bitwise, rotations to the left แตกต่างกันไปในแต่ละรอบ โดยการคำนวณ 80 รอบ จะแยกคำนวณ 4 กลุ่ม กลุ่มละ 20 รอบ และแต่ละกลุ่มจะมี ค่าคงที่ของตัวเองด้วย (K) โดยค่า K ในแต่ละรอบจะมีค่า ดังนี้

กลุ่มที่ 1 รอบที่ 0 – 19

K_1 = 0x5A827999

กลุ่มที่ 2 รอบที่ 20 – 39

K_2 = 0x6ED9EBA1

กลุ่มที่ 3 รอบที่ 40 – 59

K_3 = 0x8F1BBCDC

กลุ่มที่ 4 รอบที่ 60 – 79

K_4 = 0xCA62C1D6

ก่อนจะเข้าไปที่ algorithm จะมี ค่าเริ่มต้น(H) ที่ถูกกำหนดไว้ก่อนแล้ว ใน SHA-1 มีทั้งหมด 5 ตัว ดังนี้

H_0 = 0x67452301

H_1 = 0xEFCDAB89

H_2 = 0x98BADCFE

H_3 = 0x10325476

H_4 = 0xC3D2E1F0

เริ่มต้น Algorithm จาก input ที่เข้ามามี 512-bit เราจะแบ่งเป็น 80-words words ละ 32-bit เราจะสามารถทำได้ 16-words เท่านั้น (32 x 16 = 512) ส่วน 64-words ที่เหลือจะได้จาก 16-words ก่อนหน้าที่ผ่านการคำนวณโดย เอาแต่ละ words ตามที่ algorithm กำหนด 4 word มา XOR กัน และ leftrotate อีก 1 หนึ่งครั้ง

(words1 **XOR** words2 **XOR** words3 **XOR** words4) **leftrotate** 1

ตอนนี้เราก็ได้ 80 words แล้ว

ต่อไปเราจะเริ่มทำการ อัพเดทค่าใหม่ในแต่ละ รอบ โดยแบ่งเป็น 80 รอบ 4 กลุ่ม กลุ่ม 20 รอบ ตามที่กล่าวไปข้างต้น ก่อนจะเข้า ใน loop 80 รอบนั้น จะทำการกำหนดชื่อ ค่าเริ่มต้นที่มีในตอนแรกเสียก่อน เพื่อให้ง่ายต่อการใช้งาน ทำได้ดังนี้

A = H_0

B = H_1

C = H_2

D = H_3

E = H_4

เริ่มต้น 80 รอบการคำนวณ

Group	Rounds	Function	K
1	0 – 19	(B and C) **XOR** (NOT B and D)	0x5A827999
2	20 – 39	B **XOR** C **XOR** D	0x6ED9EBA1
3	40 – 59	(B and C) **XOR** (B and D) **XOR** (C and D)	0x8F1BBCDC
4	60 - 79	B **XOR** C **XOR** D	0xCA62C1D6

**หมายเหตุ and XOR จะเป็น bitwise operator ทั้งหมด

และในแต่ละรอบการคำนวณเราจะอัพเดทค่า A,B,C,D,E ตามนี้

F คือ function ในแต่ละรอบตามตารางด้านบน

K คือ ค่าคงที่ในแต่ละตามตารางด้านบน

i คือ จำนวนรอบ

Temp = **leftrotate 5** (A) + F + E + K + words[i]

E = D

D = C

C = **leftrotate 30** (B)

B = A

A = Temp

จบกระบวนวนรอบ 80 รอบ

สุดท้ายเราจะเอา ค่าเริ่มต้นมาบวก กับ ค่า A,B,C,D,E ที่อัพเดทไว้ ตามนี้

$$H_0 = H_0 + A$$
$$H_1 = H_1 + B$$
$$H_2 = H_2 + C$$
$$H_3 = H_3 + D$$
$$H_4 = H_4 + E$$

ค่า hash จาก SHA-1 จะมีค่าเท่ากับ H_0 ต่อกันจน H_4 รูปแบบที่แสดง H แต่ละตัวในเลขฐาน 16

Source code SHA-1

```
def SHA1(message) :
    h0 = 0x67452301
    h1 = 0xEFCDAB89
    h2 = 0x98BADCFE
    h3 = 0x10325476
    h4 = 0xC3D2E1F0
    padded_message = Padding_the_Message(message)
    for i in range(0, len(padded_message), 64):
        chunk = padded_message[i: i+64]
        words = [0] * 80
        for j in range(16):
            words[j] = int.from_bytes(chunk[j*4:j*4+4], byteorder='big')
        for j in range (16,80):
            words[j] = left_rotate(words[j-3] ^ words[j - 8] ^ words[j -14] ^
words[j - 16],1)
        A = h0
        B = h1
        C = h2
        D = h3
        E = h4
        for j in range(80):
            if j < 20:
                F = (B & C) ^ ((~B) & D)
                K = 0x5A827999
            elif  j < 40:
                F = (B ^ C)^ D
                K = 0x6ED9EBA1
            elif j < 60:
                F = (B & C) ^ (B & D) ^ (C & D)
                K = 0x8F1BBCDC
            else:
                F = B ^ C ^ D
                K = 0xCA62C1D6
```

```
            temp = left_rotate(A,5) + F + E + K + words[j] & 0xffffffff
            E = D
            D = C
            C = left_rotate(B,30)
            B = A
            A = temp
        h0 = (h0 + A) & 0xffffffff
        h1 = (h1 + B) & 0xffffffff
        h2 = (h2 + C) & 0xffffffff
        h3 = (h3 + D) & 0xffffffff
        h4 = (h4 + E) & 0xffffffff
    return '{:08x}{:08x}{:08x}{:08x}{:08x}'.format(h0, h1, h2, h3, h4)
```

คำอธิบาย

กำหนดค่าเริ่มต้น (H)

```
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0
```

เรียกใช้ function Padding_the_Message เพื่อขยายหรือบีบอัดข้อความ ให้ได้ 512-bit

```
padded_message = Padding_the_Message(message)


for i in range (0, len(padded_message), 64):
        chunk = padded_message [i: i+64]
        words = [0] * 80
```

chunk คือ กำหนดว่าจะเอาทั้งหมด ของ message

words = [0] * 80 คือ การสร้างที่ไว้รอรับคำทั้งหมด 80 words

```
for j in range (16):
            words[j] = int.from_bytes(chunk[j*4:j*4+4], byteorder='big')
```

loop ใน 16 รอบแรก จะใช้ chunk 4 ตัว ซึ่ง chunk มีทั้งหมด 64 byte จะหยิบมาทีละ 4 byte ไปลงที่ words ตัวแรกถึงตัวที่ 16 ไม่ซ้ำกันไปเรื่อย ๆ จนครบ

```
        for j in range (16,80):
            words[j] = left_rotate(words[j-3] ^ words[j - 8] ^ words[j -14] ^
words[j - 16],1)
```

64 รอบที่เหลือจะ นำเอาแต่ละ words ตามที่ algorithm กำหนด 4-word มา XOR กัน และ leftrotate อีก 1 หนึ่งครั้ง

```
 A = h0
 B = h1
 C = h2
 D = h3
 E = h4
```

กำหนดชื่อ ค่าเริ่มต้น เพื่อให้ง่ายต่อการคำนวณ

```
        for j in range(80):
            if j < 20:
                F = (B & C) ^ ((~B) & D)
                K = 0x5A827999
            elif  j < 40:
                F = (B ^ C )^ D
                K = 0x6ED9EBA1
            elif j < 60 :
                F = (B & C) ^ (B & D) ^ (C & D)
                K = 0x8F1BBCDC
            else:
                F = B ^ C ^ D
                K = 0xCA62C1D6
            temp = left_rotate(A,5) + F + E + K + words[j] & 0xffffffff
            E = D
            D = C
            C = left_rotate(B,30)
            B = A
            A = temp
```

Loop การคำนวณ 80 รอบ และการอัพเดทค่า A,B,C,D,E ตาม algorithm

```
h0 = (h0 + A) & 0xffffffff
h1 = (h1 + B) & 0xffffffff
h2 = (h2 + C) & 0xffffffff
h3 = (h3 + D) & 0xffffffff
h4 = (h4 + E) & 0xffffffff
```

นำเอา ค่าเริ่มต้นมาบวก กับ ค่า A,B,C,D,E ที่อัพเดทไว้ เหตุผลที่ and กับ 0xffffffff ก็เพราะต้องการให้แน่ใจว่า ค่าจะยังอยู่ในระบบเลขฐาน 16 อยู่

```
 return '{:08x}{:08x}{:08x}{:08x}{:08x}'.format(h0, h1, h2, h3, h4)
```

สุดท้าย return โดยการนำค่า H_0 ต่อกันจน H_4 รูปแบบที่แสดง H แต่ละตัวในเลขฐาน 16

SHA-256

SHA-256 มี output ขนาด 256-bit โดยมี input ที่เข้ามา ขนาด 512-bit (input หลังผ่าน Padding the Message) input ที่เข้ามาใน SHA-2 algorithm ผ่านการคำนวณ ทั้ง 64 ×32-bit ต่อ word ซึ่ง 1-word ค่าเท่ากับ 4-byte หรือ 32-bit เหมือนกันกับ SHA-1 โดยในแต่ละรอบการคำนวณ จะมีการบวก, and logical bitwise, rotations to the left แตกต่างกันไปในแต่ละรอบ โดยการคำนวณ 64 รอบ แต่ละรอบจะมี ค่าคงที่ของตัวเองด้วย (K) โดยค่า K ในแต่ละรอบจะมีค่า ดังนี้

```
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
```

ก่อนจะเข้าไปที่ algorithm จะมี ค่าเริ่มต้น(H) ที่ถูกกำหนดไว้ก่อนแล้ว ใน SHA-256 มีทั้งหมด 8 ตัว ดังนี้

H_0 = 0x6a09e667

H_1 = 0xbb67ae85

H_2 = 0x3c6ef372

H_3 = 0xa54ff53a

H_4 = 0x9b05688c

H_5 = 0x510e527f

H_6 = 0x1f83d9ab

H_7 = 0x5be0cd19

เริ่มต้น Algorithm จาก input ที่เข้ามามี 512-bit เราจะแบ่งเป็น 64-words words ละ 32-bit เราจะสามารถทำได้ 16-words เท่านั้น (32 x 16 = 512) เหมือนกันกับ SHA-1 ส่วน 48-words ที่เหลือจะได้จาก 16-words ก่อนหน้าที่ผ่านการคำนวณตาม Designation ที่กำหนด ดังตารางนี้

Designation	Function
Maj(x,y,z)	(x and y) **XOR** (x and z) **XOR** (y and z)
Ch(x,y,z)	(x and y) **XOR** (NOT x and z)
$\sum_0(x)$	Right rotate 2 (x) **XOR** Right rotate 13 (x) **XOR** Right rotate 22 (x)
$\sum_1(x)$	Right rotate 6 (x) **XOR** Right rotate 11 (x) **XOR** Right rotate 25 (x)
$\sigma_0(x)$	Right rotate 7 (x) **XOR** Right rotate 18 (x) **XOR** Right shift 3 (x)
$\sigma_1(x)$	Right rotate 17 (x) **XOR** Right rotate 19 (x) **XOR** Right shift 10 (x)

**หมายเหตุ and XOR จะเป็น bitwise operator ทั้งหมด

และในแต่ละรอบการคำนวณหา words ที่เหลือ จะทำตามที่กล่าวดังต่อไป

i = จำนวนรอบ

words[i] = words [i - 16] + $\sum_0(x)$ + words [i - 7] + $\sum_1(x)$

ต่อไปเราจะเริ่มทำการ อัพเดทค่าใหม่ในแต่ละ รอบ ก่อนจะเข้า ใน loop 64 รอบนั้น จะทำการกำหนดชื่อ ค่าเริ่มต้นที่มีในตอนแรกเสียก่อนเพื่อให้ง่ายต่อการใช้งาน ทำได้ดังนี้

A = H_0

B = H_1

C = H_2

D = H_3

E = H_4

F = H_5

G = H_6

H = H_7

เริ่มต้น 64 รอบการคำนวณ

ในแต่ละรอบการคำนวณเราจะอัพเดทค่า A,B,C,D,E,F,G,H ตามนี้

K = ค่าคงที่ในแต่ละรอบ

i = จำนวนรอบ

Temp1 = H + $\sum_1(e)$ + ch (e, f, g) + K[i] + words[i]

Temp2 = $\sum_0(x)$ + Maj (a, b, c)

h = g

g = f

f = e

e = (d + Temp1)

d = c

c = b

b = a

a = (Temp1 + Temp2) % 2**32

จบกระบวนวนรอบ 64 รอบ

สุดท้ายเราจะเอา ค่าเริ่มต้นมาบวก กับ ค่า A,B,C,D,E ที่อัพเดทไว้ ตามนี้

$$H_0 = H_0 + A$$
$$H_1 = H_1 + B$$
$$H_2 = H_2 + C$$
$$H_3 = H_3 + D$$
$$H_4 = H_4 + E$$
$$H_5 = H_5 + F$$
$$H_6 = H_6 + G$$
$$H_7 = H_7 + H$$

ค่า hash จาก SHA-256 จะมีค่าเท่ากับ H_0 ต่อกันจน H_7 รูปแบบที่แสดง H แต่ละตัวในเลขฐาน 16

Source code SHA-256

```
def _sigma0(num: int):
    num = (_rotate_right(num, 7) ^_rotate_right(num, 18) ^(num >> 3))
    return num

def _sigma1(num: int):
    num = (_rotate_right(num, 17) ^_rotate_right(num, 19) ^(num >> 10))
    return num

def _capsigma0(num: int):
    num = (_rotate_right(num, 2) ^_rotate_right(num, 13) ^
           _rotate_right(num, 22))
    return num

def _capsigma1(num: int):
    num = (_rotate_right(num, 6) ^_rotate_right(num, 11) ^ _rotate_right(num,
25))
    return num

def _ch(x: int, y: int, z: int):
    return (x & y) ^ (~x & z)

def _maj(x: int, y: int, z: int):
    return (x & y) ^ (x & z) ^ (y & z)
```

```
def _rotate_right(num: int, shift: int, size: int = 32):
    return (num >> shift) | (num << size - shift)
def SHA256(message_input):
    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]

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

    message = Padding_the_Message(message_input)
    words = []
    for i in range(0, len(message), 64):
        words.append(message[i:i+64])



    for message_block in words:
        message_schedule = []
        for t in range(0, 64):
            if t <= 15:
                message_schedule.append(bytes(message_block[t*4:(t*4)+4]))
            else:
                term1 = _sigma1(int.from_bytes(message_schedule[t-2], 'big'))
                term2 = int.from_bytes(message_schedule[t-7], 'big')
                term3 = _sigma0(int.from_bytes(message_schedule[t-15], 'big'))
                term4 = int.from_bytes(message_schedule[t-16], 'big')

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

```
        a = h0
        b = h1
        c = h2
        d = h3
        e = h4
        f = h5
        g = h6
        h = h7

        for t in range(64):
            t1 = ((h + _capsigma1(e) + _ch(e, f, g) + K[t] +
int.from_bytes(message_schedule[t], 'big')) % 2**32)
            t2 = (_capsigma0(a) + _maj(a, b, c)) % 2**32
            h = g
            g = f
            f = e
            e = (d + t1) % 2**32
            d = c
            c = b
            b = a
            a = (t1 + t2) % 2**32
        h0 = (h0 + a) % 2**32
        h1 = (h1 + b) % 2**32
        h2 = (h2 + c) % 2**32
        h3 = (h3 + d) % 2**32
        h4 = (h4 + e) % 2**32
        h5 = (h5 + f) % 2**32
        h6 = (h6 + g) % 2**32
        h7 = (h7 + h) % 2**32

    return '{:08x}{:08x}{:08x}{:08x}{:08x}{:08x}{:08x}{:08x}'.format(h0, h1,
h2, h3, h4, h5, h6, h7)
```

คำอธิบาย

Designation function in SHA-256

Right rotate

```
def _rotate_right(num: int, shift: int, size: int = 32):
    return (num >> shift) | (num << size - shift)
```

$\sigma_0(x)$ = Right rotate 7 (x) **XOR** Right rotate 18 (x) **XOR** Right shift 3 (x)

```
def _sigma0(num: int):
    num = (_rotate_right (num, 7) ^ _rotate_right(num, 18) ^ (num >> 3))
    return num
```

$\sigma_1(x)$ = Right rotate 17 (x) **XOR** Right rotate 19 (x) **XOR** Right shift 10 (x)

```
def _sigma1(num: int):
    num = (_rotate_right(num, 17) ^ _rotate_right(num, 19) ^ (num >> 10))
    return num
```

$\sum_0(x)$ = Right rotate 2 (x) **XOR** Right rotate 13 (x) **XOR** Right rotate 22 (x)

```
def _capsigma0(num: int):
    num = (_rotate_right(num, 2) ^ _rotate_right(num, 13) ^
           _rotate_right(num, 22))
    return num
```

$\sum_1(x)$ = Right rotate 6 (x) XOR Right rotate 11 (x) XOR Right rotate 25 (x)

```
def _capsigma1(num: int):
    num = (_rotate_right(num, 6) ^_rotate_right(num, 11) ^ _rotate_right(num,
25))
    return num
```

Ch(x,y,z) = (x and y) **XOR** (NOT x and z)

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

Maj(x,y,z) = (x and y) **XOR** (x and z) **XOR** (y and z)

```
def _maj(x: int, y: int, z: int):
    return (x & y) ^ (x & z) ^ (y & z)
```

Function ด้านบน คือ Designation function ที่อยู่ในตาราง

เริ่มเข้า function ข้างใน SHA-256

```
def SHA256(message_input):
    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]
```

ค่าคงที่(K) ของแต่ละรอบ

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

กำหนดค่าเริ่มต้น (H)

```
message = Padding_the_Message(message_input)
    words = []
```

เรียกใช้ function Padding_the_Message เพื่อขยายหรือบีบอัดข้อความ ให้ได้ 512-bit และ สร้าง list เพื่อเก็บ words แต่ละตัว

```
    for i in range(0, len(message), 64):
        words.append(message[i:i+64])
```

สิ่งที่ append เข้าไปใน words คือ การกำหนดว่าจะเอาทั้งหมด ของ message

```
for message_block in words:
        message_schedule = []
        for t in range(0, 64):
            if t <= 15:
                message_schedule.append(bytes(message_block[t*4:(t*4)+4]))
            else:
                term1 = _sigma1(int.from_bytes(message_schedule[t-2], 'big'))
                term2 = int.from_bytes(message_schedule[t-7], 'big')
                term3 = _sigma0(int.from_bytes(message_schedule[t-15], 'big'))
                term4 = int.from_bytes(message_schedule[t-16], 'big')

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

loop ใน 16 รอบแรก จะใช้ words 4 ตัว ซึ่ง words มีทั้งหมด 64 byte จะหยิบมาทีละ 4 byte ไปลงที่ words ตัวแรกถึงตัวที่ 16 ไม่ซ้ำกันไปเรื่อย ๆ จนครบ เหมือนกับ SHA-1 เพียงแต่ code นี้ จะใช้ for each วนใน words ก่อน ค่อย ทำวนข้างในอีก 64 รอบ

การทำงานในเงื่อนไข if คือ การสร้าง words 16 words แรกจาก input ส่วนการทำงานใน else คือ การสร้าง words ที่เหลือ จาก words ที่สร้างมาก่อนหน้านี้ โดยใช้การคำนวณตามที่ algorithm กำหนด

```
a = h0
b = h1
c = h2
d = h3
e = h4
f = h5
g = h6
h = h7
```

กำหนดชื่อ ค่าเริ่มต้น เพื่อให้ง่ายต่อการคำนวณ

```
for t in range(64):
            t1 = ((h + _capsigma1(e) + _ch(e, f, g) + K[t] +
int.from_bytes(message_schedule[t], 'big')) % 2**32)
            t2 = (_capsigma0(a) + _maj(a, b, c)) % 2**32
            h = g
            g = f
            f = e
            e = (d + t1) % 2**32
            d = c
            c = b
            b = a
            a = (t1 + t2) % 2**32
```

Loop การคำนวณ 64 รอบ และการอัพเดทค่า A,B,C,D,E,F,G,H ตาม algorithm

```
h0 = (h0 + a) % 2**32
h1 = (h1 + b) % 2**32
h2 = (h2 + c) % 2**32
h3 = (h3 + d) % 2**32
h4 = (h4 + e) % 2**32
h5 = (h5 + f) % 2**32
h6 = (h6 + g) % 2**32
h7 = (h7 + h) % 2**32
```

นำเอา ค่าเริ่มต้นมาบวก กับ ค่า A,B,C,D,E,F,G,H ที่อัพเดทไว้ เหตุผลที่ต้อง mod กับ 2 ** 32 ก็เพราะต้องการให้แน่ใจว่า ค่าที่ได้จะยังคงขนาดอยู่ในช่วง 32-bit

```
return '{:08x}{:08x}{:08x}{:08x}{:08x}{:08x}{:08x}{:08x}'.format(h0, h1, h2, h3, h4, h5, h6, h7)
```

สุดท้าย return โดยการนำค่า H_0 ต่อกันจน H_7 รูปแบบที่แสดง H แต่ละตัวในเลขฐาน 16

ReadMe

อย่างที่กล่าวไปใน introduction คือ การที่ทำเอกสารนี้ขึ้นมานั้นไม่ได้จะแอบอ้าง เอางานของผู้อื่นมาเป็นของตัวเองแต่อย่างใด ผมเพียงแค่ตั้งใจจะอธิบาย SHA Algorithm ในรูปแบบ ภาษาไทยแค่นั้นเอง ฉะนั้น Source code นำมาอธิบายในเอกสารฉบับนี้

ผมขอขอบคุณ คุณ Keane Nguyen (ชื่อใน Github) ที่แชร์ Source code SHA-256 ไว้ให้ ผมได้นำปรับเปลี่ยนเล็กน้อยในส่วนของ Padding the Message ที่ผมมี

Credit

Secure Hashing: SHA-1, SHA-2, and SHA-3

In book: Circuits and Systems for Security and Privacy (pp.382)Chapter: 4Publisher: CRC PressEditors: Farhana Sheikh, Leonel Sousa