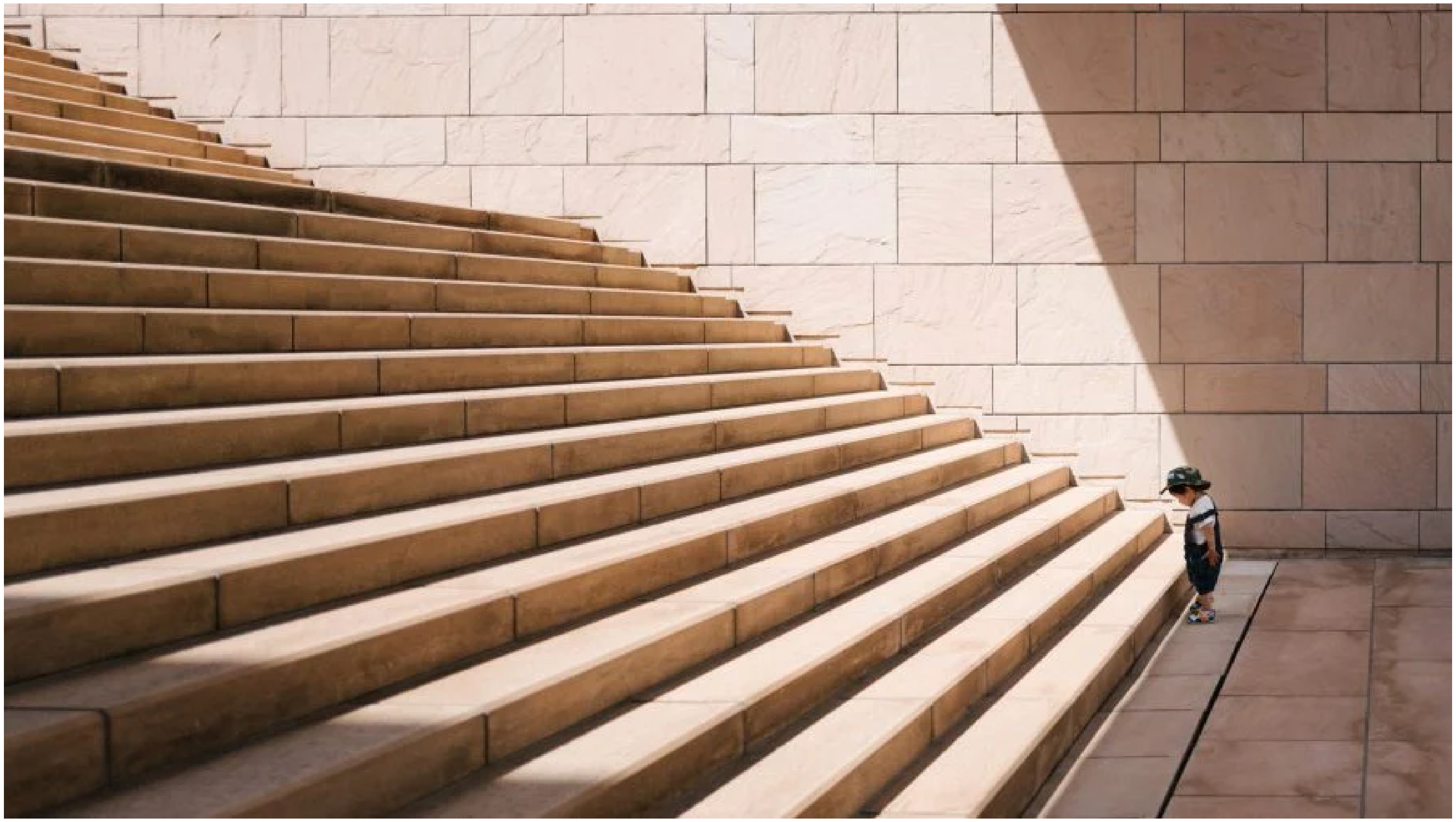
Qvault.io – Coding courses to launch your tech career



Last Updated: June 10, 2021 - Published on: July 8, 2020 by Lane Wagner

SHA-2 (Secure Hash Algorithm 2), of which SHA-256 is a part, is one of the most popular hashing algorithms out there. In this article, we're going to break down each step of the algorithm and work through a real-life example by hand. SHA-2 is known for its security (it hasn't broken down like SHA-1), and its speed. In cases where keys are not being generated, such as mining Bitcoin, a fast hash algorithm like SHA-2 often reigns supreme.

What Is a Hash Function?

Three of the main purposes of a hash function are:

- To scramble data deterministically
- To accept input of any length and output a fixed-length result

SHA-2 is a very famous and strong family of hash functions, as as you would expect, it fulfills all of the above purposes. Take a look at our article on hash functions if you need to brush up on their properties.

SHA-2 Family vs SHA-256

SHA-2 is an algorithm, a generalized idea of how to hash data. SHA-2 has several variants that all use the same algorithm but use different constants. SHA-256, for example. sets additional constants that define the SHA-2 algorithm's behavior, one such constant being the output size, `256`. The `256` and `512` in `SHA-256` and `SHA-512` refer to their respective digest sizes in bits.

SHA-2 Family vs SHA-1

SHA-2 is a successor to the SHA-1 hash and remains one of the strongest hash functions in use today. SHA-256, as opposed to SHA-1, hasn't been compromised. For this reason, there's really no reason to use SHA-1 these days, it isn't safe. The flexibility of output size (224, 256, 512, etc) also allows SHA-2 to pair well with popular KDFs and ciphers like AES-256.

Formal Acceptance by NIST

SHA-256 is formally defined in the National Institute of Standards and Technology's FIPS 180-4. Along with standardization and formalization comes a list of test vectors that allow developers to ensure they've implemented the algorithm properly.

Step 1 – Pre-Processing

- Convert “hello world” to binary:

```
01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111
01110010 01101100 01100100
```

- Append a single 1:

```
01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111
01110010 01101100 01100100 1
```

- Pad with 0’s until data is a multiple of 512, less 64 bits (448 bits in our case):

```
01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111
01110010 01101100 01100100 10000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

binary, “1011000”.

```
01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111
01110010 01101100 01100100 10000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 01011000
```

Now we have our input, which will always be evenly divisible by 512.

Step 2 – Initialize Hash Values (h)

Now we create 8 hash values. These are hard-coded constants that represent the first 32 bits of the fractional parts of the square roots of the first 8 primes: 2, 3, 5, 7, 11, 13, 17, 19

```
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
```

Step 3 – Initialize Round Constants (k)

Similar to step 2, we are creating some constants (Learn more about constants and when to use them here). This time, there are 64 of them. Each value (0-63) is the first 32 bits of the fractional parts of the cube roots of the first 64 primes (2 – 311).

```
0x428a2f98 0x71374491 0xb5c0fbcf 0xe9b5dba5 0x3956c25b 0x59f111f1 0x923f8
0xd807aa98 0x12835b01 0x243185be 0x550c7dc3 0x72be5d74 0x80deb1fe 0x9bdc0
0xe49b69c1 0xefbe4786 0x0fc19dc6 0x240ca1cc 0x2de92c6f 0x4a7484aa 0x5cb0a
0x983e5152 0xa831c66d 0xb00327c8 0xbf597fc7 0xc6e00bf3 0xd5a79147 0x06ca6
0x27b70a85 0x2e1b2138 0x4d2c6dfc 0x53380d13 0x650a7354 0x766a0abb 0x81c2c
0xa2bfe8a1 0xa81a664b 0xc24b8b70 0xc76c51a3 0xd192e819 0xd6990624 0xf40e3
0x19a4c116 0x1e376c08 0x2748774c 0x34b0bcb5 0x391c0cb3 0x4ed8aa4a 0x5b9cc
0x748f82ee 0x78a5636f 0x84c87814 0x8cc70208 0x90befffa 0xa4506ceb 0xbef9a
```

Step 4 – Chunk Loop

The following steps will happen for each 512-bit “chunk” of data from our input. In our case, because “hello world” is so short, we only have one chunk. At each iteration of the loop, we will be mutating the hash values h0-h7, which will be the final output.

Step 5 – Create Message Schedule (w)

```
01101000011001010110110001101100 01101111001000000111011101101111
01110010011011000110010010000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000001011000
```

- Add 48 more words initialized to zero, such that we have an array **w[0…63]**

```
01101000011001010110110001101100 01101111001000000111011101101111
01110010011011000110010010000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000001011000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
...
...
00000000000000000000000000000000 00000000000000000000000000000000
```

- s0 = (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)
- s1 = (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (w[i- 2] rightshift 10)
- w[i] = w[i-16] + s0 + w[i-7] + s1

Let’s do w[16] so we can see how it works:

```
w[1] rightrotate 7:
  01101111001000000111011101101111 -> 11011110110111100100000011101110
w[1] rightrotate 18:
  01101111001000000111011101101111 -> 00011101110110111101101111001000
w[1] rightshift 3:
  01101111001000000111011101101111 -> 00001101111001000000111011101101

s0 = 11011110110111100100000011101110 XOR 00011101110110111101101111001000 XOR 00001101111001000000111011101101

s0 = 11001110111000011001010111001011

w[14] rightrotate 17:
  00000000000000000000000000000000 -> 00000000000000000000000000000000
w[14] rightrotate19:
  00000000000000000000000000000000 -> 00000000000000000000000000000000
w[14] rightshift 10:
  00000000000000000000000000000000 -> 00000000000000000000000000000000

s1 = 00000000000000000000000000000000 XOR 00000000000000000000000000000000 XOR 00000000000000000000000000000000

s1 = 00000000000000000000000000000000

w[16] = w[0] + s0 + w[9] + s1
```

```
// addition is calculated modulo 2^32

w[16] = 00110111010001110000001000110111
```

This leaves us with 64 words in our message schedule (w):

```
01101000011001010110110001101100 01101111001000000111011101101111
01110010011011000110010010000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000001011000
00110111010001110000001000110111 10000110110100001100000000110001
11010011101111010001000100001011 01111000001111110100011110000010
00101010100100000111110011101101 01001011001011110111110011001001
00110001111000011001010001011101 10001001001101100100100101100100
01111111011110100000011011011010 11000001011110011010100100111010
10111011111010001111011001010101 00001100000110101110001111100110
10110000111111100000110101111101 01011111011011100101010110010011
00000000100010011001101101010010 00000111111100011100101010010100
00111011010111111110010111010110 01101000011001010110001011100110
11001000010011100000101010011110 00000110101011111001101100100101
10010010111011110110010011010111 01100011111110010101111001011010
```

```
00010000100001000101001100011101 01100000100100111110000011001101
10000011000000110101111111101001 11010101101011100111100100111000
00111001001111110000010110101101 11111011010010110001101111101111
11101011011101011111111100101001 01101010001101101001010100110100
00100010111111001001110011011000 10101001011101000000110100101011
01100000110011110011100010000101 11000100101011001001100000111010
00010001010000101111110110101101 10110000101100000000110111011001
10011000111100001100001101101111 01110010000101111011100000011110
10100010110101000110011110011010 00000001000011111001100101111011
11111100000010111010011110001010 11000010110000101110101100010110
```

Step 6 – Compression

- Initialize variables **a, b, c, d, e, f, g, h** and set them equal to the current hash values respectively. **h0, h1, h2, h3, h4, h5, h6, h7**
- Run the compression loop. The compression loop will mutate the values of **a…h**. The compression loop is as follows:
- for i from 0 to 63
  - S1 = (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
  - ch = (e and f) xor ((not e) and g)
  - temp1 = h + S1 + ch + k[i] + w[i]
  - S0 = (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
  - maj = (a and b) xor (a and c) xor (b and c)
  - temp2 := S0 + maj
  - h = g
  - g = f

- d = c
- c = b
- b = a
- a = temp1 + temp2

Let's go through the first iteration, all addition is calculated modulo 2^32:

```
a = 0x6a09e667 = 01101010000010011110011001100111
b = 0xbb67ae85 = 10111011011001111010111010000101
c = 0x3c6ef372 = 00111100011011101111001101110010
d = 0xa54ff53a = 10100101010011111111010100111010
e = 0x510e527f = 01010001000011100101001001111111
f = 0x9b05688c = 10011011000001010110100010001100
g = 0x1f83d9ab = 00011111100000111101100110101011
h = 0x5be0cd19 = 01011011111000001100110100011001

e rightrotate 6:
  01010001000011100101001001111111 -> 11111101010001000011100101001001
e rightrotate 11:
  01010001000011100101001001111111 -> 01001111111010100010000111001010
e rightrotate 25:
  01010001000011100101001001111111 -> 10000111001010010011111110101000
S1 = 11111101010001000011100101001001 XOR 01001111111010100010000111001010
S1 = 00110101100001110010011100101011

e and f:
    01010001000011100101001001111111
  & 10011011000001010110100010001100 =
```

```
(not e) and g:
    10101110111100011010110110000000
  & 00011111100000111101100110101011 =
    00001110100000011000100110000000
ch = (e and f) xor ((not e) and g)
   = 00010001000001000100000000001100 xor 00001110100000011000100110000000
   = 00011111100001011100100110001100

// k[i] is the round constant
// w[i] is the batch
temp1 = h + S1 + ch + k[i] + w[i]
temp1 = 01011011110000011001101000110 01 + 00110101100001110010011100101 0
temp1 = 01011011110111010101100111010100

a rightrotate 2:
  01101010000010011110011001100111 -> 11011010100000100111100110011001
a rightrotate 13:
  01101010000010011110011001100111 -> 00110011001110110101000001001111
a rightrotate 22:
  01101010000010011110011001100111 -> 00100111100110011001110110101000
S0 = 11011010100000100111100110011001 XOR 00110011001110110101000001001111
S0 = 11001110001000001011010001111110

a and b:
    01101010000010011110011001100111
  & 10111011011001111010111010000101 =
    00101010000000011010011000000101
a and c:
    01101010000010011110011001100111
  & 00111100011011101111001101110010 =
```

```
  & 00111100011011101111001101110010 =
    00111000011001101010001000000000
maj = (a and b) xor (a and c) xor (b and c)
    = 00101010000000011010011000000101 xor 00101000000010001110001001100
    = 00111010011011111110011001100111

temp2 = S0 + maj
      = 11001110001000001011010001111110 + 00111010011011111110011001100111
      = 00001000100100001001101011100101

h = 00011111100000111101100110101011
g = 10011011000001010110100010001100
f = 01010001000011100101001001111111
e = 10100101010011111111010100111010 + 01011011110111010101100111010100
  = 00000001001011010100111100001110
d = 00111100011011101111001101110010
c = 10111011011001111010111010000101
b = 01101010000010011110011001100111
a = 01011011110111010101100111010100 + 00001000100100001001101011100101
  = 01100100011011011111010010111001
```

That entire calculation is done 63 more times, modifying the variables a-h throughout. We won't do it by hand but we would have ender with:

```
h0 = 6A09E667 = 01101010000010011110011001100111
h1 = BB67AE85 = 10111011011001111010111010000101
h2 = 3C6EF372 = 00111100011011101111001101110010
```

```
h6 = 1F83D9AB = 00011111100000111101100110101011
h7 = 5BE0CD19 = 01011011111000001100110100011001

a = 4F434152 = 01001111010000110100000101010010
b = D7E58F83 = 11010111111001011000111110000011
c = 68BF5F65 = 01101000101111110101111101100101
d = 352DB6C0 = 00110101001011011011011011000000
e = 73769D64 = 01110011011101101001110101100100
f = DF4E1862 = 11011111010011100001100001100010
g = 71051E01 = 01110001000001010001111000000001
h = 870F00D0 = 10000111000011110000000011010000
```

Step 7 – Modify Final Values

After the compression loop, but still, within the chunk loop, we modify the hash values by adding their respective variables to them, a-h. As usual, all addition is modulo 2^32.

```
h0 = h0 + a = 10111001010011010010011110111001
h1 = h1 + b = 10010011010011010011111000001000
h2 = h2 + c = 10100101001011100101001011010111
h3 = h3 + d = 11011010011111011010101111111010
h4 = h4 + e = 11000100100001001110111111100011
h5 = h5 + f = 01111010010100111000000011101110
h6 = h6 + g = 10010000100010001111011110101100
h7 = h7 + h = 11100010111011111100110111101001
```

Last but not least, slap them all together, a simple string concatenation will do.

```
digest = h0 append h1 append h2 append h3 append h4 append h5 append h6 a
       = B94D27B9934D3E08A52E52D7DA7DABFAC484EFE37A5380EE9088F7ACE2EFCDE9
```

Done! We've been through every step (sans some iterations) of SHA-256 in excruciating detail ☺

I'm glad you've made it this far! Going step-by-step through the SHA-256 algorithm isn't exactly a walk in the park. Learning the fundamentals that underpin web security can be a huge boon to your career as a computer scientist, however, so keep it up!

The Pseudocode

If you want to see all the steps we just did above in pseudocode form, then here is it is, straight from WikiPedia:

```
Note 1: All variables are 32 bit unsigned integers and addition is calcul
Note 2: For each round, there is one round constant k[i] and one entry in
Note 3: The compression function uses 8 working variables, a through h
Note 4: Big-endian convention is used when expressing the constants in th
    and when parsing message block data from bytes to words, for example,
    the first word of the input message "abc" after padding is 0x61626380
```

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

Initialize array of round constants:
(first 32 bits of the fractional parts of the cube roots of the first 64
k[0..63] :=
   0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1
   0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe
   0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa
   0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147
   0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb
   0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624
   0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a
   0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb

Pre-processing (Padding):
begin with the original message of length L bits
append a single '1' bit
append K '0' bits, where K is the minimum number >= 0 such that L + 1 + K
append L as a 64-bit big-endian integer, making the total post-processed

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    create a 64-entry message schedule array w[0..63] of 32-bit words
```

```
Extend the first 16 words into the remaining 48 words w[16..63] of th
for i from 16 to 63
    s0 := (w[i-15] rightrotate  7) xor (w[i-15] rightrotate 18) xor (
    s1 := (w[i- 2] rightrotate 17) xor (w[i- 2] rightrotate 19) xor (
    w[i] := w[i-16] + s0 + w[i-7] + s1

Initialize working variables to current hash value:
a := h0
b := h1
c := h2
d := h3
e := h4
f := h5
g := h6
h := h7

Compression function main loop:
for i from 0 to 63
    S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate
    ch := (e and f) xor ((not e) and g)
    temp1 := h + S1 + ch + k[i] + w[i]
    S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate
    maj := (a and b) xor (a and c) xor (b and c)
    temp2 := S0 + maj

    h := g
    g := f
    f := e
    e := d + temp1
    d := c
```

```
    Add the compressed chunk to the current hash value:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
    h4 := h4 + e
    h5 := h5 + f
    h6 := h6 + g
    h7 := h7 + h

Produce the final hash value (big-endian):
digest := hash := h0 append h1 append h2 append h3 append h4 append h5 ap
```

Other hash function explainers

If you’re looking for an explanation of a different hash function, we may have you covered

- (Very) Basic Intro to the Scrypt Hash
- Bcrypt Step by Step
- (Very) Basic Intro to Hash Functions

Have questions or feedback?

Follow and hit me up on Twitter @q_vault if you have any questions or comments. If I've made a mistake in the article be sure to let me know so I can get it corrected!

Cryptography, Bitcoin, Security

‹ How to Rerender a Vue Route When Path Parameters Change

› Your Manager Can't Code? They Shouldn't Be Your Manager

Leave a Comment

You must be logged in to post a comment.