Study and Implementation of the BLAKE Hash function

Ajay Choudhury
Roll no.: 18018, EECS Department, IISERB
BS Project Report (Under Dr Shashank Singh) - ECS412
Submitted on: 21st April 2022

Abstract

This project studies the working principles and methods of the hash functions MD5 and BLAKE. MD5 hash function is not cryptographically secure now, it has already been broken and is used only for local integrity checks and other simple tasks, but it is a great example to understand the concept of hash functions; it introduces message compression using a naive method and then with the help of some non-linear functions on the message, the hash for the message is generated. The BLAKE hash function is similar to the MD5 hash function for a few initial steps. MD5 produces a 128-bit hash value with the compression of the message by adding some extra bits and applying some non-linear functions. But in BLAKE, a core function uses 16 different states to process the content of the message and generates a 256-bit hash value. The working method of both the hash functions is explained, and the hash functions are implemented in the C programming language for simple string values.

Introduction

Hash functions take strings or files of arbitrary length and compress them into shorter lines unique to the particular message or file processed. Hashing is a process of scrambling a piece of information or data beyond recognition to produce an output called the hash. The hash functions are computationally feasible when run in the forward direction, but they become computationally expensive when run backwards to derive the message or input from the hash value; this makes the hashes (or hash digests) practically irreversible.

Mathematically, it can be represented as follows:

$$h : \{0,1\}^{\infty} \rightarrow \{0,1\}^{n}$$

The hash functions take any combination of zeros and ones, i.e. binary input, and calculate a hash value of fixed length (say n) for the file or message given as input. The hash value is practically different and unique for every individual file or message passed as input to the hash functions but, theoretically, as we can derive from the mathematical representation of the hash functions, the input set is of infinite elements, and the output set consists of a finite number of elements i.e. the hashes; so, there must be infinitely many messages for which the hash value will be same. A visual interpretation is shown below:

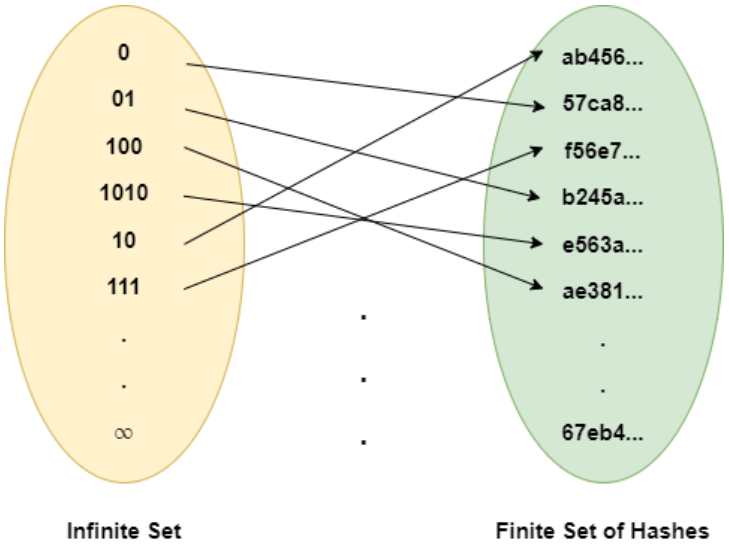


Fig. 1: Mapping between an infinite set of inputs and a finite set of hashes

Any cryptographic hash function is not completely unbreakable but the hash functions are designed in such a way that calculating the messages from their hash values is so computationally expensive that it will not be feasible for the computational power available today or the hash values should be practically irreversible. On the other hand, the same hash function should run in polynomial time and efficiently in the forward direction.

There are three main features of a cryptographic hash function, and they are as follows:

a. **Collision resistance:** A collision in a function *H* is a pair of distinct inputs *x* and *x'* such that *H(x) = H(x')*. In this case, we also say that *x* and *x'* collide under *H*. A function *H* is collision-resistant if it is infeasible for any polynomial-time algorithm to find a collision in *H*.

b. **Preimage resistance:** Informally, a hash function is preimage resistant if given a message *x* and some hash digest *y*. It is hard for a polynomial-time algorithm to find a value *x'* such that *H(x') = y*.

c. **Second preimage resistance:** In simple words, a hash function is said to be second preimage resistant if given a message and its hash value. It is computationally impractical to find another message such that they have the same hash value. In other words, a hash function is second preimage resistant if given *x; it* is hard for a polynomial-time algorithm to find *x'* such that *H(x) = H(x')*.

There are many fields of application for cryptographic hash functions like digital signatures, public-key encryption, integrity verification, message authentication, password protection, key agreement protocols, and many other cryptographic protocols. Currently, encryption of an email, sending a message on your mobile phone, connecting to an HTTPS website, or connecting to a remote machine through IPSec or SSH all use a hash function somewhere under the hood. Git revision control system uses a hash function to identify files and changes in a repository, host-based intrusion detection systems (HIDS) use them to detect modified files, network-based intrusion detection systems (NIDS) use hashes to detect known-malicious data going through a network, forensic analysts use hash values to prove that digital artefacts have not been modified, Bitcoin uses a hash function in its proof-of-work systems, and there are many more such use-cases of hash functions.

Some names of hash functions are MD5, SHA-1, SHA-256, BLAKE, Keccak, etc. The MD5 and SHA-1 hash functions are cryptographically broken but are still used for non-cryptographic uses like integrity checks on local servers, etc. SHA-2 and SHA-3 are cryptographically secure and used widely for various purposes.

MD5 Hash Algorithm and Implementation

MD5 hash function was designed as a successor of MD4 by Ronald Rivest in 1991. It is a one-way hash function, initially designed for digital signatures. It has been broken and is not used for cryptographic-purpose anymore but still can be used for integrity check of files against unintentional corruption. It generates a 128-bit hash digest for a given input message or file.

The algorithm of MD5

The algorithm for the MD5 hash function can be elaborated in six steps:

1. **Appending padding bits:** Some extra bits are added to the original message such that the total length of the output becomes 64-bit less than a multiple of 512-bit. Padding bits have the first bit like one, and the rest are 0's.

2. **Appending length bits:** To make the total length of the output a multiple of *512*, we need to add a 64-bit length section to the message, which is determined by the size of the message. For example, if the message is a string is *"ball",* then the length bit will be *(8 x 4) bits = 32 bits =* 00100000000000000000000000000000000000000000000000000000000000000000 in binary form and little-endian representation.

3. **Initializing MD buffers and constants:** Four *32-bit* buffers that contain some hexadecimal values are initialised, as shown below:

    - `A = 0x67452301`
    - `B = 0xefcdab89`
    - `C = 0x98badcfe`
    - `D = 0x10325476`

4. **Dividing the message into 512-bit blocks and further into 32-bit sub-blocks:** The whole output generated after appending the padding and length bits is a multiple of 512 in length. So the message is now divided into n blocks of 512-bit each. Each block of 512-bit is now divided into 16 sub-blocks of 32-bit each and four rounds of operations are performed on each sub-block. 16 operations (one on each sub-block) are performed in each round i.e 64 operations in each block of 512-bit.

5. **Operation on each sub-block:** Here, the figure below represents a single MD5 operation on a sub-block in each 512-bits block. Here,
    - **a,b,c,d:** Chaining variables
    - **P:** Non-linear function
    - **M[i]:** i^{th} part of the message
    - **t[k]:** k^{th} constant
    - **CLS_s:** Circular Left Shift by s-bits

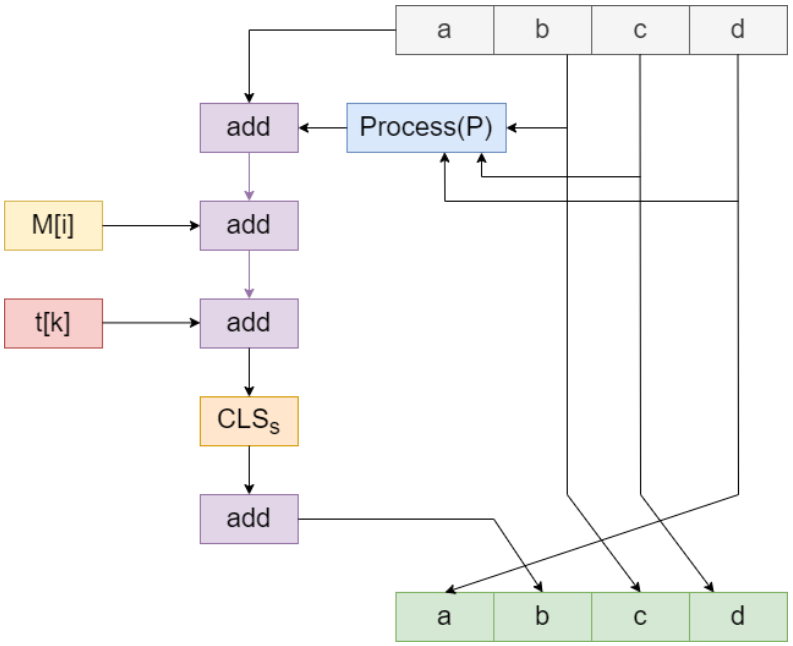


Fig.2: Single operation on a 32-bit sub-block

Here Single MD5 operation can also be represented in the equation as

$$a = b + ((a + P(b, c, d) + M[i] + t[k]) <<< s)$$

In each round, 16 operations are performed and in each round different auxiliary function or process (P) is used:

$$F(B, C, D) = (B \wedge C) \vee (\neg B \wedge D)$$
$$G(B, C, D) = (B \wedge D) \vee (C \wedge \neg D)$$
$$H(B, C, D) = B \oplus C \oplus D$$
$$I(B, C, D) = C \oplus (B \vee \neg D)$$

6. **Generating the message digest:** After all the rounds have been performed, the buffers *a, b, c* and *d* are added with the initial buffers *A, B, C* and *D* and is re-assigned to *a, b, c* and *d*. Now, *a, b, c* and *d* contains the MD5 output of the message, starting from *a* and ending with *d*, but the representation should be little-endian. The resultant buffer is passed on as the starting buffer for the next 512-bit block (if current block is not the last *512-bit* block). Similar cycle goes on until the last block from which we get the *128-bit* MD5 digest or hash.

Implementation of MD5 in C [Repository]

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

// initial MD buffers
uint32_t A, B, C, D;
```

```c
#define LEFTROTATE(x, c) (((x) << (c)) | ((x) >> (32 - (c))))

// function for circular left shift

void md5(uint8_t *in_msg, size_t in_len)
{
	// define the constants
	uint32_t t[] = {0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee, 0xf57c0faf,
0x4787c62a, 0xa8304613, 0xfd469501, 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be,
0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821, 0xf61e2562, 0xc040b340, 0x265e5a51,
0xe9b6c7aa, 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8, 0x21e1cde6, 0xc33707d6,
0xf4d50d87, 0x455a14ed, 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a, 0xfffa3942,
0x8771f681, 0x6d9d6122, 0xfde5380c, 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70,
0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05, 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8,
0xc4ac5665, 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039, 0x655b59c3, 0x8f0ccc92,
0xffeff47d, 0x85845dd1, 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1, 0xf7537e82,
0xbd3af235, 0x2ad7d2bb, 0xeb86d391};

	uint32_t s[] = {7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5,
9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 4, 11, 16, 23, 4, 11, 16, 23, 4,
11, 16, 23, 4, 11, 16, 23, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15,
21};

	// initialize MD buffers
	A = 0x67452301;
	B = 0xefcdab89;
	C = 0x98badcfe;
	D = 0x10325476;

	// copy of MD buffers
	uint32_t a = A;
	uint32_t b = B;
	uint32_t c = C;
	uint32_t d = D;

	// calculate length of the message after padding --------- to be derived
	// here it should be 448 bits = 448/8 = 56 bytes
	int pad_len = 56;

	// for length bit we need to add 64 bits = 64/8 = 8 bytes
	// 56 + 8 = 64 bytes
	uint8_t new_msg[64] = {0};

	// copy the messages ascii value to the new message
	for (int i = 0; i < in_len; i++)
	{
		new_msg[i] = in_msg[i];
	}

	// add a 1 just after the message for the beginning of padding bits
```

```
// ascii of 128 = 10000000
new_msg[in_len] = 128;

// length bit: here a string is taken hence each character is of 1 byte
// thus, 8*in_len = length of message
uint8_t len_bit = 8 * in_len;

// add the length bit to the new message
memcpy(new_msg + pad_len, &len_bit, 1);

// a loop for accessing 512-bit blocks
// a loop for accessing all 16 32-bit blocks in the 512-bit block
// here only one 512-bit block is there → skipping the outer loop
uint32_t i;
uint32_t *lit_end_msg = (uint32_t *)(new_msg);

// print the new message in
for (i = 0; i < 64; i++)
{
      uint32_t f, g;
      if (i < 16)
      {
            f = (b & c) | ((~b) & d);
            g = i;
      }
      else if (i < 32)
      {
            f = (d & b) | ((~d) & c);
            g = (5 * i + 1) % 16;
      }
      else if (i < 48)
      {
            f = b ^ c ^ d;
            g = (3 * i + 5) % 16;
      }
      else
      {
            f = c ^ (b | (~d));
            g = (7 * i) % 16;
      }

      // swap buffers
      f = f + a + t[i] + lit_end_msg[g];
      a = d;
      d = c;
      c = b;
      b = b + LEFTROTATE(f, s[i]);
}

// Add this chunk's hash to result so far:
```

```
        A += a;
        B += b;
        C += c;
        D += d;

        // print the new message
        for (int i = 0; i < 64; i++)
        {
                printf("%d: %d\n", i + 1, new_msg[i]);
        }
        // print the little endian message
        // 97 106 97 121 = 01100001 01101010 01100001 01111001 (ajay)
        // in lit endian = 01111001 01100001 01101010 01100001 (121 97 106 97 - yaja)
        printf("\nIn Little-Endian Form\n");
        for (int i = 0; i < 64; i++)
        {
                printf("%d: %d\n", i + 1, lit_end_msg[i]);
        }
}

int main()
{
        char *msg = "ajay";
        size_t msg_len = strlen(msg);

        // invoke the md5 function
        md5(msg, msg_len);

        // hash
        printf("Hash: %x%x%x%x\n", A, B, C, D);
        // need to reverse the little endianness
        uint8_t *temp = (uint8_t *)&A;
        printf("MD5 Hash: %x%x%x%x", temp[0], temp[1], temp[2], temp[3]);
        temp = (uint8_t *)&B;
        printf("%x%x%x%x", temp[0], temp[1], temp[2], temp[3]);
        temp = (uint8_t *)&C;
        printf("%x%x%x%x", temp[0], temp[1], temp[2], temp[3]);
        temp = (uint8_t *)&D;
        printf("%x%x%x%x", temp[0], temp[1], temp[2], temp[3]);
}
```

Output:

```
MD5 Hash: 29e45782db729fa1059d4294ede399
```

The BLAKE Hash function

The BLAKE hash function was one of the five entries for SHA-3 standards in the NIST (National Institute of Standards and Technology) hash function competition that made it to the final round. Other finalists include

Grøstl, JH, Keccak and Skein. The competition was held between 2007 and 2012 and the Keccak hash function was announced as the winner and the new standard for the SHA-3 hash algorithm.

The BLAKE hash functions follow the *HAIFA* iteration mode: the compression function depends on the salt and the number of bits hashed so far (counter), to compress each message block with a distinct function. BLAKE repeatedly combines an 8-word hash value with 16 message words, truncating the *ChaCha* result to obtain the next hash value. BLAKE-256 and BLAKE-224 use *32-bit* words and produce digest sizes of 256 bits and 224 bits, respectively, while BLAKE-512 and BLAKE-384 use *64-bit* words and produce digest sizes of 512 bits and 384 bits, respectively.

An implementation of BLAKE is claimed to require low resources and is fast in both software and hardware environments. The successor of the BLAKE hash function is the BLAKE2 hash function, which was announced in 2012 and the BLAKE3 hash function, based on BLAKE2, was announced in 2020.

The algorithm for the BLAKE-256 hash function

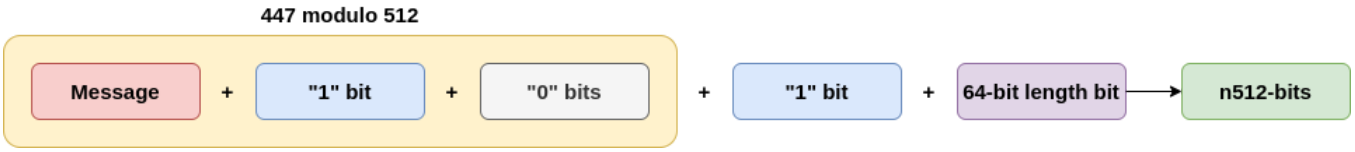
BLAKE hash function has mainly two variants, one which uses *32-bit* words and another one that uses *64-bit* words, they produce a hash digest of length *256-bit* and *512-bit* respectively. BLAKE-256 works on a file or message of a maximum length of (2^{64} - *1*) and produces a hexadecimal output of length 64 i.e. *256-bit* hash.

Major steps involved in the BLAKE hash function are:

a. **Padding bits:** The data to be hashed (at most *(2^{64} −1) bits*) is first padded such that its length becomes congruent to *447 modulo 512*. It is then split into *512-bit* blocks. It involves two steps:

   1. Append to the data a bit *"1"* followed by the minimal (possibly zero) number of bits *"0"* so that the total length is congruent to *447 modulo 512*. Thus, at least one bit and at most 512 are appended.
   2. Append a bit *"1"* to the data, followed by a *64-bit* unsigned big-endian representation of the original length of data. Then, the 512-bit blocks are divided into 16 *32-bit sub-blocks*.

   This process makes sure that the bit length of the padded data is a multiple of 512.



b. **Initialization:** A 512-bit (16 words) initial state is initialized and represented as a *4 x 4 array*. It is initialized with *h, s, t* and word constants as shown below

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix}$$

$$\begin{aligned} c_0 &= 243F6A88 & c_1 &= 85A308D3 \\ c_2 &= 13198A2E & c_3 &= 03707344 \\ c_4 &= A4093822 & c_5 &= 299F31D0 \\ c_6 &= 082EFA98 & c_7 &= EC4E6C89 \\ c_8 &= 452821E6 & c_9 &= 38D01377 \\ c_{10} &= BE5466CF & c_{11} &= 34E90C6C \\ c_{12} &= C0AC29B7 & c_{13} &= C97C50DD \\ c_{14} &= 3F84D5B5 & c_{15} &= B5470917 \end{aligned}$$

Here, *h* represents the chain value, *s* is the salt, and *c* refers to 16 constant values (the leading 512 or 1024 bits of the fractional part of π), and *t* is the counter. It also uses a table of 10 *16-element permutations*. The salt *s* is chosen by the user, and set to the null value when no salt is required (i.e., $s_0 = s_1 = s_2 = s_3 = 0$). *BLAKE-256(m, s)* = h^N, where *m* is the (non-padded) message, and *s* is the salt. The notation *BLAKE-256(m)* denotes the hash of *m* when no salt is used (i.e., *s = 0*).

σ[0]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ[1]	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
σ[2]	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
σ[3]	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
σ[4]	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
σ[5]	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
σ[6]	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
σ[7]	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
σ[8]	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
σ[9]	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	0

Table 1: The table of 16-element permutations

c. **Compression using the round function:** A round function iterates 14 times on the state *v* and uses a single core function *G* eight times in a round, i.e. 112 times. The core operation *G*, equivalent to ChaCha's quarter round, operates on a *4-word column* or diagonal *a, b, c, d*, combined with two words of message *m[ ]* and two constant words *c[ ]*. It is performed eight times per full round. The operation is depicted visually as

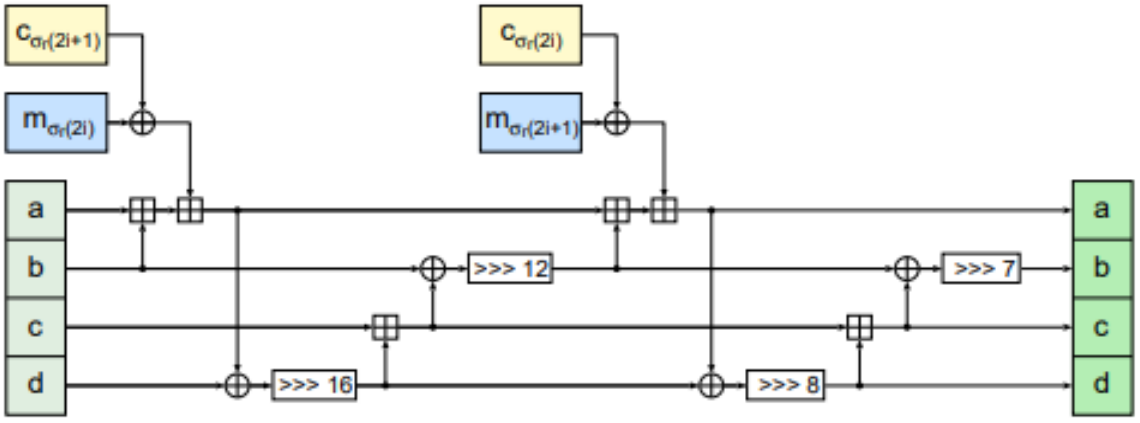


Fig.3: Core operation G

The operation can also be depicted in form of equations as:

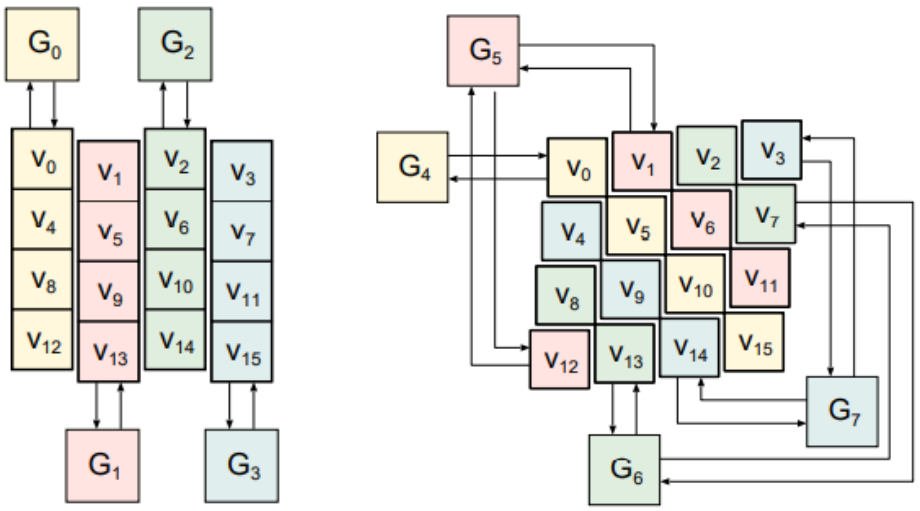
$$a \leftarrow a + b + (m_{\sigma r(2i)} \oplus c_{\sigma r(2i+1)})$$
$$d \leftarrow (d \oplus a) \gg 16$$

```
c ← c + d
b ← (b ⊕ c) ≫ 12
a ← a + b + (m_σr(2i+1) ⊕ c_σr(2i) )
d ← (d ⊕ a) ≫ 8
c ← c + d
b ← (b ⊕ c) ≫ 7
```

Above, r is the round number (0–13), at round $r \geqslant 10$, the permutation used is $\sigma_{r \bmod 10}$, for example, at the last round $(r = 15)$, the permutation $\sigma_{15 \bmod 10} = \sigma_5$ is used and i varies from 0 to 7. Also, the addition here is addition modulo 2^{32} and $\oplus$ refers to the XOR operation. $(a >>> s)$ means right shift operation by s-bit on a.

Here, G is applied to the word matrix in two ways: *column-wise* and *diagonally*. Firstly, G is applied to each column with four state variables (say v_0, v_4, v_8 and v_{12}) and the values of those state variables are updated, next, the core function G is applied diagonally, as depicted with different colours in figure 4. The order of the round functions is depicted by the number given in the figure below. Similarly, each of the 14 rounds has 4 column steps and 4 diagonal steps respectively.

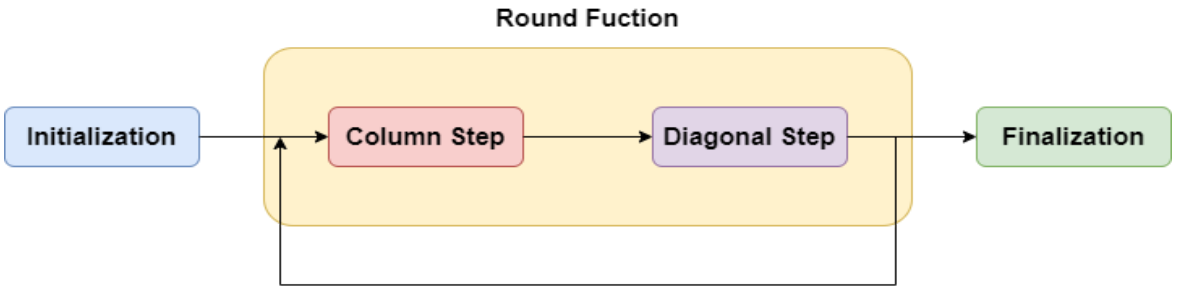
$$G_0(v_0, v_4, v_8, v_{12}) \quad G_1(v_1, v_5, v_9, v_{13}) \quad G_2(v_2, v_6, v_{10}, v_{14}) \quad G_3(v_3, v_7, v_{11}, v_{15})$$
$$G_4(v_0, v_5, v_{10}, v_{15}) \quad G_5(v_1, v_6, v_{11}, v_{12}) \quad G_6(v_2, v_7, v_8, v_{13}) \quad G_7(v_3, v_4, v_9, v_{14})$$



(a) (b)

Fig.4: Applying the function G (a) column-wise and (b) diagonally.

In each round function, the process goes on as depicted below:



d. **Finalisation:** The new chain values are extracted from the modified state variables. XOR operations are performed on them along with salts and the initial chain values to get the final hash value. It is depicted as

$$
\begin{aligned}
h'_0 &\leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8 \\
h'_1 &\leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9 \\
h'_2 &\leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v_{10} \\
h'_3 &\leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v_{11} \\
h'_4 &\leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v_{12} \\
h'_5 &\leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v_{13} \\
h'_6 &\leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v_{14} \\
h'_7 &\leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v_{15}
\end{aligned}
$$

The resultant values $h'_0 \ldots h'_7$ make the hash value for the message and is represented in big-endian representation.

Implementation of BLAKE-256 Hash Function in C [Repository]

```c
#include <string.h>
#include <stdio.h>
#include <stdint.h>

typedef struct
{
  uint32_t h[8], s[4], t[2];
  int buflen, nullt;
  uint8_t buf[64];
} state256;

#define U8TO32_BIG(p)                                          \
  (((uint32_t)((p)[0]) << 24) | ((uint32_t)((p)[1]) << 16) | \
   ((uint32_t)((p)[2]) << 8) | ((uint32_t)((p)[3])))

#define U32TO8_BIG(p, v)         \
  (p)[0] = (uint8_t)((v) >> 24); \
  (p)[1] = (uint8_t)((v) >> 16); \
  (p)[2] = (uint8_t)((v) >> 8);  \
  (p)[3] = (uint8_t)((v));

const uint8_t sigma[][16] =
    {
        {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15},
        {14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3},
        {11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4},
        {7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8},
```

```c
        {9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13},
        {2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9},
        {12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11},
        {13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10},
        {6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5},
        {10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0},
        {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15},
        {14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3},
        {11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4},
        {7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8},
        {9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13},
        {2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9}};

const uint32_t constant[16] =
    {
        0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344,
        0xa4093822, 0x299f31d0, 0x082efa98, 0xec4e6c89,
        0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,
        0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917};

static const uint8_t padding[129] =
    {
        0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

// initialization of states
void initialize(state256 *S)
{
  S->h[0] = 0x6a09e667;
  S->h[1] = 0xbb67ae85;
  S->h[2] = 0x3c6ef372;
  S->h[3] = 0xa54ff53a;
  S->h[4] = 0x510e527f;
  S->h[5] = 0x9b05688c;
  S->h[6] = 0x1f83d9ab;
  S->h[7] = 0x5be0cd19;
  S->t[0] = S->t[1] = S->buflen = S->nullt = 0;
  S->s[0] = S->s[1] = S->s[2] = S->s[3] = 0;
}

// core function
void core_function(state256 *S, const uint8_t *block)
{
  // states and message block - 32-bit each
```

```
  uint32_t v[16], m[16], i;
  // Shift
#define ROT(x, n) (((x) << (32 - n)) | ((x) >> (n)))
// core function
#define G(a, b, c, d, e)                                      \
  v[a] += (m[sigma[i][e]] ^ constant[sigma[i][e + 1]]) + v[b]; \
  v[d] = ROT(v[d] ^ v[a], 16);                                \
  v[c] += v[d];                                               \
  v[b] = ROT(v[b] ^ v[c], 12);                                \
  v[a] += (m[sigma[i][e + 1]] ^ constant[sigma[i][e]]) + v[b]; \
  v[d] = ROT(v[d] ^ v[a], 8);                                 \
  v[c] += v[d];                                               \
  v[b] = ROT(v[b] ^ v[c], 7);

  // convert take 8-bit blocks into 32-bit and big-endian format
  for (i = 0; i < 16; ++i)
    m[i] = U8TO32_BIG(block + i * 4);

  // initial states
  for (i = 0; i < 8; ++i)
    v[i] = S->h[i];

  // rest states
  v[8] = S->s[0] ^ constant[0];
  v[9] = S->s[1] ^ constant[1];
  v[10] = S->s[2] ^ constant[2];
  v[11] = S->s[3] ^ constant[3];
  v[12] = constant[4];
  v[13] = constant[5];
  v[14] = constant[6];
  v[15] = constant[7];

  // XOR with t is not required when the block has padding-bits
  if (!S->nullt)
  {
    v[12] ^= S->t[0];
    v[13] ^= S->t[0];
    v[14] ^= S->t[1];
    v[15] ^= S->t[1];
  }

  // run the core function 14 times for blake 256-hash
  for (i = 0; i < 14; ++i)
  {
    // column step
    G(0, 4, 8, 12, 0);
    G(1, 5, 9, 13, 2);
    G(2, 6, 10, 14, 4);
    G(3, 7, 11, 15, 6);
    // diagonal step
```

```
    G(0, 5, 10, 15, 8);
    G(1, 6, 11, 12, 10);
    G(2, 7, 8, 13, 12);
    G(3, 4, 9, 14, 14);
  }

  // generating the hash with all updated states
  for (i = 0; i < 16; ++i)
    S->h[i % 8] ^= v[i];

  for (i = 0; i < 8; ++i)
    S->h[i] ^= S->s[i % 4];
}

// update the length of the block left and to fill
void add_padding(state256 *S, const uint8_t *in, uint64_t inlen)
{
  // space left
  int left = S->buflen;
  // printf("\nleft = %d\n", left);

  // space left
  int fill = 64 - left;
  // printf("fill = %d\n", fill);

  // data left is not null and length to be left is greater than available
  if (left && (inlen >= fill))
  {
    // printf("1st condition\n");
    memcpy((void *)(S->buf + left), (void *)in, fill);
    S->t[0] += 512;

    // printf("S->t[0] = %d\n", S->t[0]);

    if (S->t[0] == 0)
      S->t[1]++;

    // for (int j = 0; j < 64; j++)
    // {
    //   printf("S->buf[%d] = %d\n", j, S->buf[j]);
    // }
    // printf("\n");

    core_function(S, S->buf);
    in += fill;
    inlen -= fill;

    // printf("inlen = %d\n", inlen);
    left = 0;
  }
```

```
  // if meesage is greater than length of 64
  while (inlen ⩾ 64)
  {
    // printf("Condition 2\n");
    S->t[0] += 512;

    if (S->t[0] == 0)
      S->t[1]++;

    core_function(S, in);
    in += 64;
    inlen -= 64;
  }

  // if the message when block is empty
  if (inlen > 0)
  {
    // printf("Condition 3\n");
    memcpy((void *)(S->buf + left), (void *)in, (size_t)inlen);
    S->buflen = left + (int)inlen;
    // printf("inlen = %d\n", inlen);
    // printf("buflen = %d\n", S->buflen);
    // for (int j = 0; j < 64; j++)
    // {
    //   printf("S->buf[%d] = %d\n", j, S->buf[j]);
    // }
  }
  else
    S->buflen = 0;
}

// finalize blake 256
void final(state256 *S, uint8_t *out)
{
  uint8_t msglen[8], zo = 0x01, oo = 0x81;
  uint32_t lo = S->t[0] + (S->buflen << 3), hi = S->t[1];
  // printf("lo = %d\n", lo);
  // printf("hi = %d\n", hi);

  // space fill is less than greater than 2^32 bits
  if (lo < (S->buflen << 3))
    hi++;

  // get the message in 8-bit big-endian format
  U32TO8_BIG(msglen + 0, hi);
  U32TO8_BIG(msglen + 4, lo);

  // print the message
  // for (int i = 0; i < 8; i++)
```

```
// {
//   printf("msglen[%d] = %d\n", i, msglen[i]);
// }

// only one byte for padding is fill
if (S->buflen == 55)
{
  S->t[0] -= 8;
  add_padding(S, &oo, 1);
}
else
{
  // atleast 2 bytes are available for padding
  if (S->buflen < 55)
  {
    // if buflen is 0
    if (!S->buflen)
      S->nullt = 1;

    S->t[0] -= 440 - (S->buflen << 3);
    // printf("S[t[0]] = %d\n", S->t[0]);
    // printf("buflen = %d\n", S->buflen);
    add_padding(S, padding, 55 - S->buflen);
  }
  else
  {
    S->t[0] -= 512 - (S->buflen << 3);
    add_padding(S, padding, 64 - S->buflen);
    S->t[0] -= 440;
    add_padding(S, padding + 1, 55);
    S->nullt = 1;
  }

  // add one after padding 0 bits
  add_padding(S, &zo, 1);
  S->t[0] -= 8;
  // printf("S->t[0] = %d\n", S->t[0]);
}

S->t[0] -= 64;
// printf("S->t[0] = %d\n", S->t[0]);
// for (int j = 0; j < 64; j++)
// {
//   printf("S->buf[%d] = %d\n", j, S->buf[j]);
// }

add_padding(S, msglen, 8);

// converting the 32-bit blocks into 8-bit hash output in big-endian
U32TO8_BIG(out + 0, S->h[0]);
```

```
  U32TO8_BIG(out + 4, S->h[1]);
  U32TO8_BIG(out + 8, S->h[2]);
  U32TO8_BIG(out + 12, S->h[3]);
  U32TO8_BIG(out + 16, S->h[4]);
  U32TO8_BIG(out + 20, S->h[5]);
  U32TO8_BIG(out + 24, S->h[6]);
  U32TO8_BIG(out + 28, S->h[7]);
}

void blake_256(uint8_t *out, const uint8_t *in, uint64_t inlen)
{
  state256 S;
  initialize(&S);
  add_padding(&S, in, inlen);
  final(&S, out);
}

int main(int argc, char **argv)
{
  // take the message
  char *in = "";

  // length of message
  size_t msg_len = strlen(in);

  // output array - this will contain the final hash
  uint8_t out[32];

  // invoke the hashing function
  blake_256(out, in, msg_len);

  // print the hash in hexadecimal form
  printf("BLAKE256 HASH for \"\": ");
  for (int i = 0; i < 32; ++i)
  {
    printf("%x", out[i]);
  }
  return 0;
}
```

Output

```
BLAKE256 HASH for "": 716f6e863f744b9ac22c97ec7b76ea5f598bc5b2f67c61510bfc4751384ea7a
```

Notations Used

Symbol	Meaning
←	variable assignment

+	addition modulo 2 32 or (modulo 2 64)
⊕	Boolean exclusive OR (XOR)
n >>> k	rotation of n by k bits towards less significant bits or right shift by k-bits i.e. $n / 2^k$
n <<< k	rotation of n by k bits towards more significant bits or left shift by k-bits i.e. $n \times 2^k$

References

1. SHA-3 proposal BLAKE, version 1.3. [Source]- SHA-3 proposal BLAKE (aumasson.jp).
2. The Hash Function BLAKE- Jean-Philippe Aumasson, Willi Meier Raphael C.-W. Phan, Luca Henzen.