

# The Evolution of Hashing

*What is four letters and is both a tasty breakfast item as well as a plant with pointy leaves?*

**If you guessed “hash,” then you are correct!** However, hash has another meaning that relates to cryptography, and that is the focus of this article.

A hash function is a cryptographic tool that facilitates secure authentication and ensures data message integrity across digital channels. Simply stated, a hash function is a process that takes plaintext data of any size and converts it into a unique ciphertext of a specific length. No two pieces of input will have the same hash digest, and if the input changes, the hash digest changes as well. This mathematical process allows passwords to be stored securely in a database.

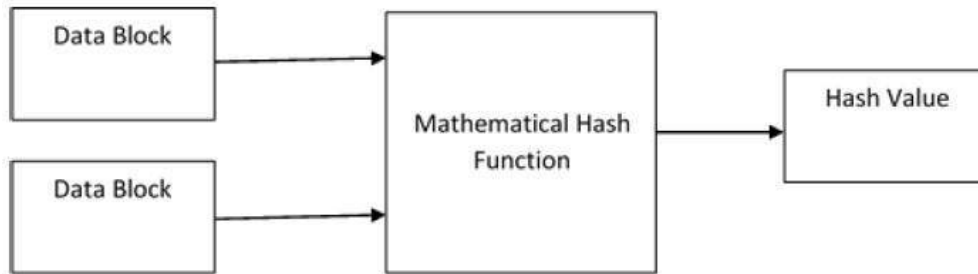
## Hashing



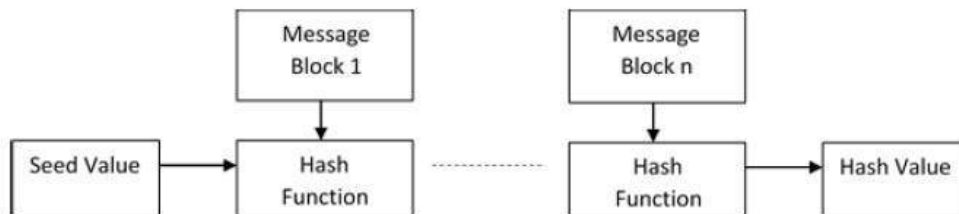
Hash functions are considered one-way because of the computing power, time, and cost it would take to brute force reverse the hash function. Attempting every possible combination leading to a hash value is completely impractical, meaning that, for all intents and purposes, hash functions are one-way.

Conversely, encryption is a two-way function. The purpose of encryption is to prevent unauthorized or unintended parties from accessing the data. Encryption relies on the use of a key, such that the data can only be decrypted by an individual with the key.

A hashing function operates on two fixed-size blocks of data to create a hash value, as depicted in the image below.. The size of each data block varies depending on the algorithm, typically ranging from 128 to 512 bits.



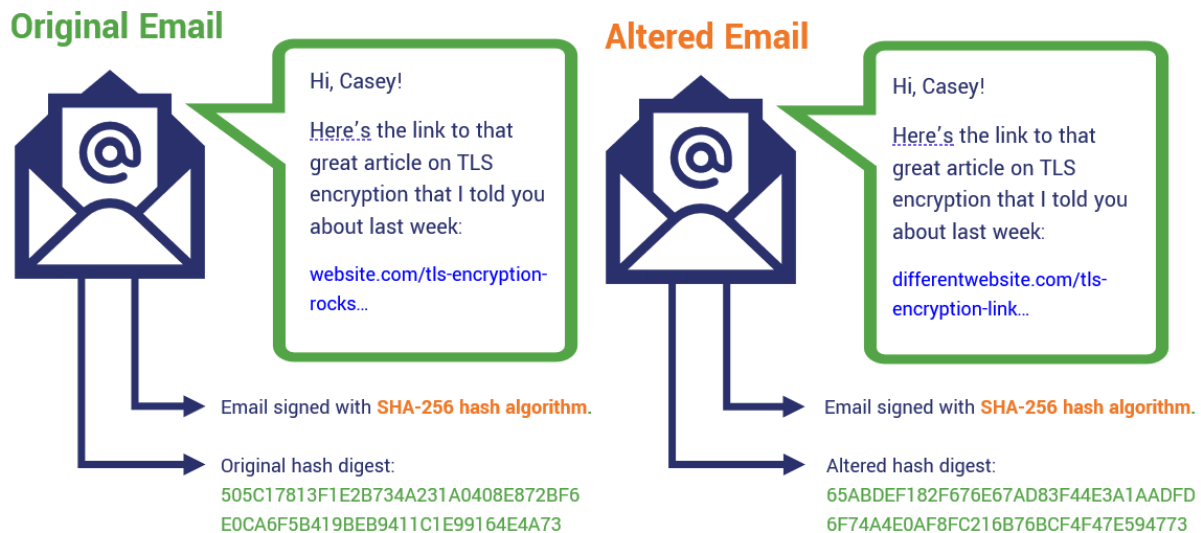
A hashing algorithm involves rounds of these hashing functions like a block cipher. Each round takes an input of a fixed size, typically a combination of the most recent message block and the output of the previous round. This process is repeated for as many rounds as required to hash the entire message, as seen in the following image.



The hash value of the first message block becomes an input to the second hash operation. This is known as an avalanche effect of hashing.

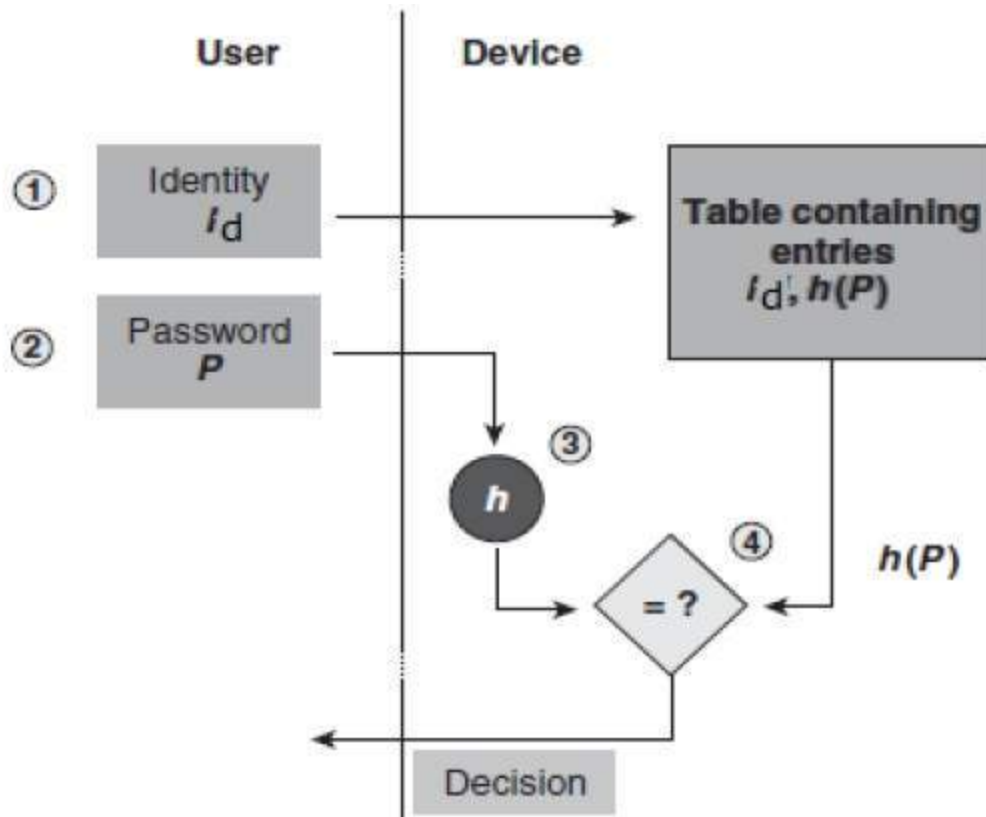
There are two direct applications of hash functions based on their cryptographic properties. Hash functions ensure data integrity in public key cryptography, meaning that they serve as a way to identify whether data has been tampered with after it has been signed.

## How Hashing Ensures Data Integrity



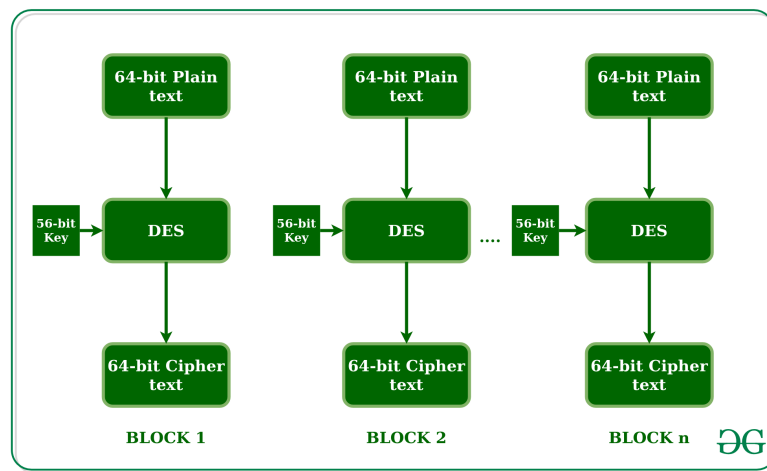
In the example depicted in the image above, a digitally signed email was manipulated in transit. The hash digest changes completely when any of the email content gets modified after being digitally signed, indicating that it cannot be trusted. This informs the recipient of the email that the message has been changed.

Hash functions also facilitate the verification and safe storage of passwords, which is discussed in much further detail in this article. Nowadays, many websites provide the option to store a password. But, storing plaintext passwords in a public-facing server would be dangerous, leaving the information vulnerable to cybercriminals. Instead, most websites hash passwords to generate hash values, which are stored instead of the plaintext passwords. This login process is depicted in the image below and simulated in the rest of the article.



## DES

DES(Data Encryption Standard) is based on the Feistel block cipher, and was developed in 1971 by IBM cryptography researcher Horst Feistel. DES became the approved federal encryption standard in November 1976 by the National Institute of Standards and Technology (NIST), and was reaffirmed as the standard in 1983, 1988, and 1999. DES is a block cipher and encrypts data in 64-bit blocks, which means that 64 bits of plain text goes as the input to DES, which produces 64 bits of ciphertext. The same algorithm and 56-bit key are used for encryption and decryption, with minor differences. The image below illustrates this process.

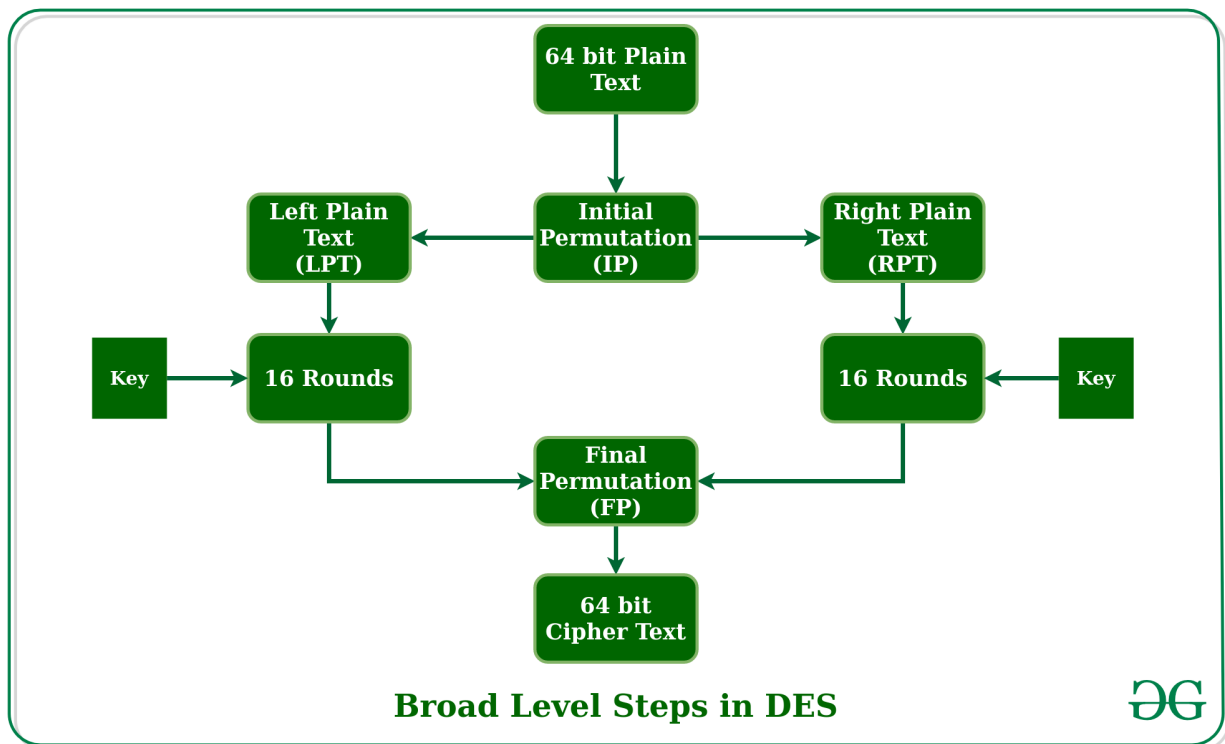


The initial key actually consists of 64 bits. Before the DES process even starts, every 8th bit of the key is discarded to create a 56-bit key, as seen in the image below.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

**Figure - discarding of every 8<sup>th</sup> bit of original key**

DES is based on the two fundamental attributes of cryptography, substitution and transposition. DES consists of 16 steps, or rounds. Each round performs the steps of substitution and transposition. This process occurs as described in the image below.



Before the first round, Initial Permutation (IP) occurs. In IP, the bit positions of the original plain text block are rearranged. As seen in the code below, IP replaces the first bit of the original plain text block with the 58th bit of the original plain text, the second bit with the 50th bit of the original plain text, and so on.

```
// Initial Permutation Table (Standard IP Table Values for DES found online)
int initial_perm[64] = {
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
};
```

*Can be viewed in DES.cpp, function DES*

After IP, the resulting 64-bit permuted text block is divided into two half blocks, the Right Plain Text and Left Plain Text, each consisting of 32 bits. From the 56-bit key, a different 48-bit Sub Key is generated during each round using key transformation. In key transformation, the 56-bit key is divided into two halves, each 28 bits. These halves are circularly shifted left by one or two positions, depending on the round. The number of key bits shifted per round is shown in the code below.

```
// Number of bit shifts
int shift_table[16] = {
    1, 1, 2, 2,
    2, 2, 2, 2,
    1, 2, 2, 2,
    2, 2, 2, 1
};
```

*Can be viewed in DES.cpp, function getKeys*

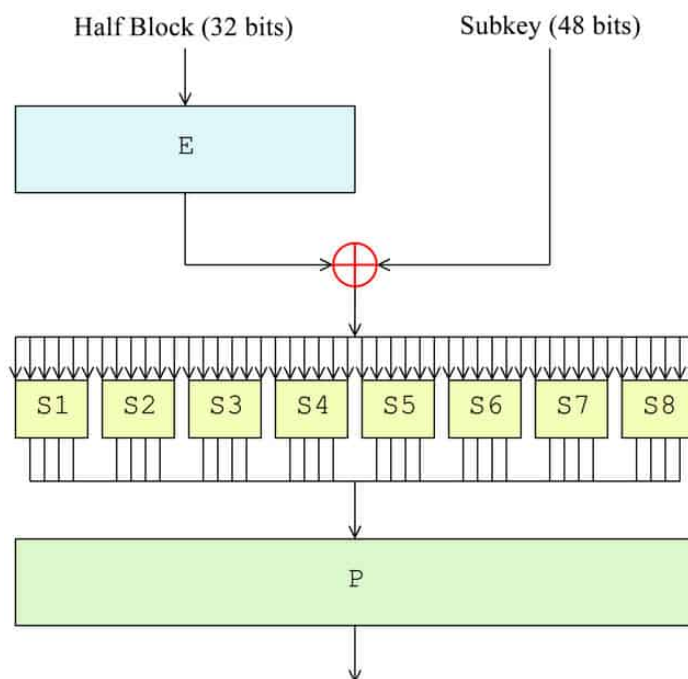
After an appropriate shift, 48 of the 56 bits are selected. The bits are selected as seen in the code below.

```
// Key- Compression Table
int key_comp[48] = {
    14, 17, 11, 24, 1, 5,
    3, 28, 15, 6, 21, 10,
    23, 19, 12, 4, 26, 8,
    16, 7, 27, 20, 13, 2,
    41, 52, 31, 37, 47, 55,
    30, 40, 51, 45, 33, 48,
    44, 49, 39, 56, 34, 53,
    46, 42, 50, 36, 29, 32
};
```

*Can be viewed in DES.cpp, function getKeys*

Because the key transformation process involves permutation as well as a selection of a 48-bit subset of the original 56-bit key, it is called Compression Permutation. This compression permutation technique produces a different subset of key bits in each round, which makes DES not easy to crack.

The following process is performed first on the Right Plain Text (RPT) and next on the Left Plain Text (LPT).



During the expansion permutation, the Right Plain Text (RPT) is expanded from 32 bits to 48 bits. The 32-bit RPT is divided into 8 blocks, each consisting of 4 bits. Then, each 4-bit block of the previous step is expanded to a corresponding 6-bit block. This process results in expansion as well as a permutation of the input bit while creating output. Next, the 48-bit key is XOR with the 48-bit RPT, called key mixing, as seen in the code below.

```
for (int i = 0; i < 16; i++) {
    // Expansion D-box
    right_expanded = permute(right, exp_d, 48);

    // XOR RoundKey[i] and right_expanded
    x = binxor(rkb[i], right_expanded);
```

*Can be viewed in DES.cpp, function DES*

The result of this is divided into 8 blocks, each 6 bits. After going through the eight substitution blocks, each block is reduced from 6 to 4 bits. The first and last bit of each block provides the row index, and the remaining bits provide the column index. These indices are used to look up values in a substitution box. This helps to further obscure the relationship between the ciphertext and the plaintext that it is linked to. A substitution box has 4 rows, 16 columns, and contains numbers from 0 to 15. Some of the substitution boxes are seen in the code below.

```
// Substitution Value Table (Standard S-box Values for DES found online)
int s[8][4][16] = {
    // Substitution box 1
    { 14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,
      0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
      4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
      15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13 },
    // Substitution box 2
    { 15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10,
      3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
      0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
      13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9 },
    // Substitution box 3
    { 10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,
      13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
      -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

*Can be viewed in DES.cpp, function DES*



The result is transposed in another permutation, in accordance with the following rule.

```
// Straight Permutation Table
int per[32] = {
    16, 7, 20, 21,
    29, 12, 28, 17,
    1, 15, 23, 26,
    5, 18, 31, 10,
    2, 8, 24, 14,
    32, 27, 3, 9,
    19, 13, 30, 6,
    22, 11, 4, 25
};
```

*Can be viewed in DES.cpp, function DES*

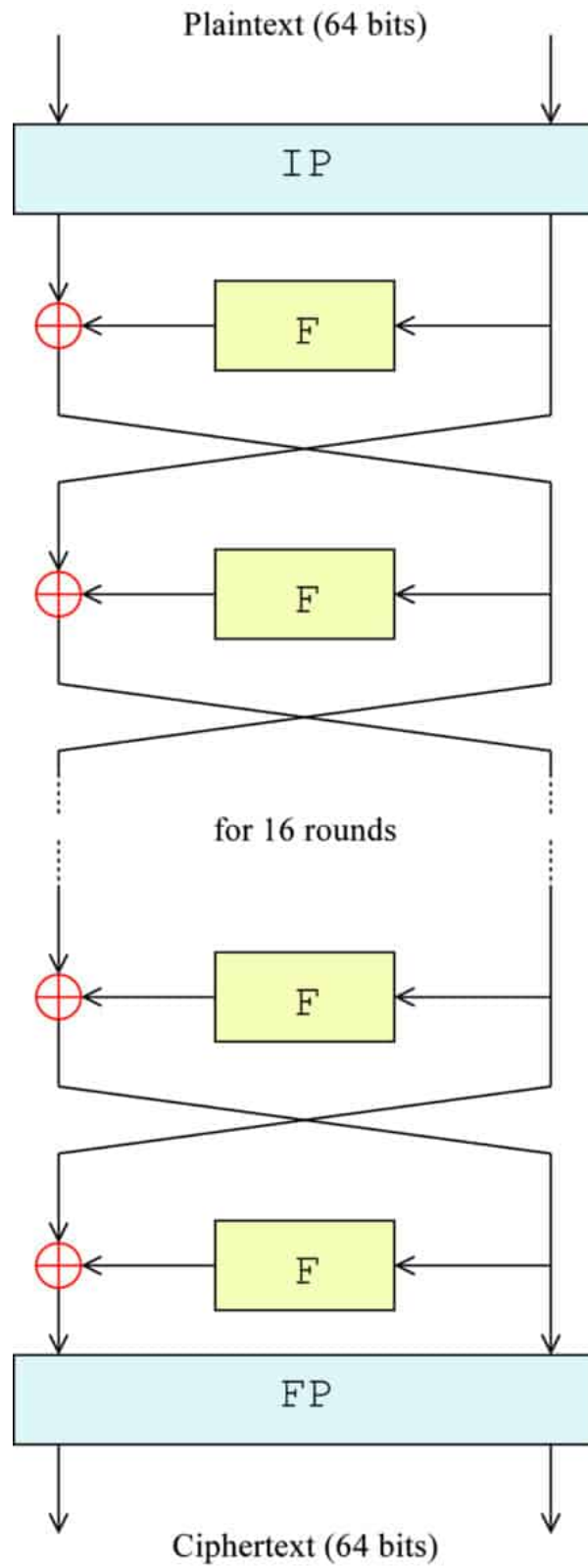
The result is XOR with the left half, and this is stored in the RPT. The data goes through these four steps (expansion permutation, key mixing, substitution, and permutation), called the F function, followed by the XOR, another 15 times, for a total of 16 rounds. In the second round, the original, untouched version of the right side of the block becomes the new left side. The result of the first round is sent through the F function. Everything remains the same, except the subkey for round two is used instead. This result is XOR with the new left side. Essentially, the old right side becomes the new left, while the result of the operation becomes the new right side, as seen in the code below.

```
// XOR left and op
x = binxor(op, left);
left = x;

// Swapper if not at the last key
if (i != 15)
    swap(left, right);
```

*Can be viewed in DES.cpp, function DES*

This process of switching between the left and right sides is illustrated in the following image.



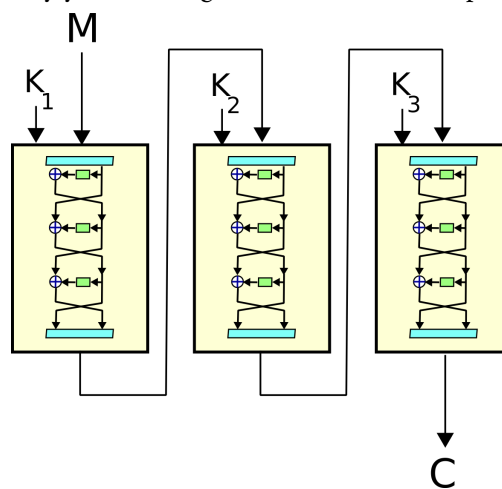
This process continues until the fifteenth round, with the blocks switching over and the next subkey being used in each round. In the final round, the blocks are not switched over. This allows the algorithm to be used for both encryption and decryption. Instead, the blocks are combined to form a 64-bit block.

The final step in the final permutation. This permutation is the inverse of the initial permutation, and it adds no extra security value. It rearranges the data according to the following table, and the result is the final ciphertext.

```
// Straight Permutation Table
int per[32] = {
    16, 7, 20, 21,
    29, 12, 28, 17,
    1, 15, 23, 26,
    5, 18, 31, 10,
    2, 8, 24, 14,
    32, 27, 3, 9,
    19, 13, 30, 6,
    22, 11, 4, 25
};
```

*Can be viewed in DES.cpp, function DES*

Starting in 2002, DES was no longer deemed suitable as the creations of AES (Advanced Encryption Standard) replaced it as the acceptable standard. Despite this claim, DES will remain prevalent in government and banking for many years through a new version, “Triple-DES”.



*Image representing Triple DES*

Triple DES(3DES) is a symmetric key-block cipher. It utilizes the DES cipher three times. In this process, it encrypts with the first key,  $k_1$ , then decrypts with the second key,  $k_2$ , and finally encrypts with the third key,  $k_3$ . In 3DES, there are 2 keys variations, and  $k_1$  and  $k_3$  are the same, where  $k_2$  is different.

In May 2005, the NIST officially withdrew the 1999 reaffirmation of DES declaring it. However, 3DES (3DES), is approved for sensitive government information through 2030. Triple DES is still used today, but considered a legacy encryption algorithm. Some examples of its implementations included Microsoft Office, Firefox and EMV payment systems. Nowadays, many no longer use 3DES as there are better, more secure alternatives. While Triple DES is in use now, the NIST plans to deprecate it in 2023 and disallow all forms of Triple-DES from 2024 onward. This signifies the end of an era in these algorithms.

## **MD5**

MD5 is a cryptographic hashing algorithm of the message digest family designed in 1991 as an improvement upon the earlier MD4 hashing algorithm. The algorithm outputs a unique 128-bit digest from an input string of arbitrary size. MD5 is no longer considered cryptographically secure; in 2008, the Carnegie Mellon Software Engineering Institute declared MD5 was “cryptographically broken and unsuitable for further use.” Despite this, a 2019 report estimated that 25% of all content management systems use MD5 for password hashing. Due to its susceptibility to collision attacks, most US government entities require the SHA family of hashing functions be used in place of MD5, especially in applications that utilize digital signatures.

The first step in implementing the MD5 algorithm is padding the input string. Bits are padded so that the length of the original string (in bits) is congruent to 448 modulo 512. The first bit added is 1 with trailing 0's until the length is 64 bits less than a multiple of 512. If the string is already congruent to 448 modulo 512, this padding is still done.

Next, 64 bits are allocated to the digest, making its bitlength congruent to 512. The message digest buffer, consisting of 4 words contained in 32-bit registers, is then initialized with fixed hexadecimal values. The fixed values are as shown:

Word A	01	23	45	67
Word B	89	Ab	Cd	Ef
Word C	Fe	Dc	Ba	98
Word D	76	54	32	10

This initialization is implemented in the code as shown:

```
// initial buffer values
MD5h0 = 0x67452301;
MD5h1 = 0xefcdab89;
MD5h2 = 0x98badcfe;
MD5h3 = 0x10325476;

// Initialize hash value for this chunk:
uint32_t a = MD5h0;
uint32_t b = MD5h1;
uint32_t c = MD5h2;
uint32_t d = MD5h3;
```

*Can be viewed in MD5.cpp*

A series of auxiliary functions are then applied to each 512-bit chunk of the input message digest. Each chunk is first broken into 16 sub-blocks  $M[0], M[1], \dots, M[15]$ . Each sub-block is initialized with the above fixed hexadecimal values; from there, a series of 64 iterations operate over the sub-blocks, with the following functions being applied:

$$\begin{aligned}
 F(X, Y, Z) &= (X \wedge Y) \vee (\neg X \wedge Z) \\
 G(X, Y, Z) &= (X \wedge Y) \vee (Y \wedge \neg Z) \\
 H(X, Y, Z) &= X \oplus Y \oplus Z \\
 I(X, Y, Z) &= Y \oplus (X \vee \neg Z)
 \end{aligned}$$

The function  $F(X, Y, Z)$  is applied to the first 16 passes of the loop,  $G(X, Y, Z)$  is applied to passes 17-32,  $H(X, Y, Z)$  is applied to passes 33-48, and  $I(X, Y, Z)$  is applied to passes 49-64. An integer  $g$  is computed on each pass to correspond to the specific element—a 32-bit word—in the 512-bit chunk that the rotation will be applied to. The degree of rotation is specified in a predefined array  $r$  of 64 integer values, as shown below:

```
uint32_t r[] = {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};
```

*Can be viewed in MD5.cpp*

Using the computed value of  $g$ ; the number of rotations to be applied from  $r$ ; and values  $a$ ,  $b$ ,  $c$ , and  $d$ —32-bit integer hash values initialized from each  $MD5h0$ ,  $MD5h1$ ,  $MD5h2$ , and  $MD5h3$ —the algorithm applies a left rotation as shown below:

```
uint32_t temp = d;
d = c;
c = b;
// apply left rotation given parameters specified by a, f, k[i], w[g], and r[i]
printf("\t\trotate left(%x + %x + %x + %x, %d)\n", a, f, k[i], w[g], r[i]);
b = b + LEFTROTATE((a + f + k[i] + w[g]), r[i]);
a = temp;
```

*Can be viewed in MD5.cpp*

Upon each rotation, the values contained in the variables  $a$ ,  $b$ ,  $c$ , and  $d$  are updated. After 64 rotations are performed upon the 512-bit chunk, the values contained in each variable are added to the buffer variables as shown below:

```
// update buffer values hash to result so far
MD5h0 += a;
MD5h1 += b;
MD5h2 += c;
MD5h3 += d;
```

*Can be viewed in MD5.cpp*

After each 512-bit chunk is passed through the auxiliary functions, the resulting buffer values are no longer changed. The address of each buffer then contains 8 of the 32 characters in the final MD5 hash. The hexadecimal values contained at each of these values is then concatenated to display the MD5 hash, as shown below:

```
/* *****
Step 5 - Concatenate final buffer values to achieve final MD5 hash
***** */
printf("\n\033[0;31m\t\t***** Step 5 *****\n\033[0m");
printf("Concatenate final buffer blocks to achieve final MD5 hash\n\n");
printf("32 bit block 1: ");
p=(uint8_t *)&MD5h0;
printf("%.2x%.2x%.2x%.2x\n", p[0], p[1], p[2], p[3]);

printf("32 bit block 2: ");
p=(uint8_t *)&MD5h1;
printf("%.2x%.2x%.2x%.2x\n", p[0], p[1], p[2], p[3]);

printf("32 bit block 3: ");
p=(uint8_t *)&MD5h2;
printf("%.2x%.2x%.2x%.2x\n", p[0], p[1], p[2], p[3]);

printf("32 bit block 4: ");
p=(uint8_t *)&MD5h3;
printf("%.2x%.2x%.2x%.2x\n", p[0], p[1], p[2], p[3]);
```

*Can be viewed in MD5.cpp*

Upon running the MD5 function, the final step displayed to the terminal shows the hex values contained in each of the final 4 blocks, and their subsequent concatenation—which is the complete MD5 hash.

```

***** Step 5 *****
Concatenate final buffer blocks to achieve final MD5 hash

32 bit block 1: bcbcdcf
32 bit block 2: 4c9b8542
32 bit block 3: 075fc6b7
32 bit block 4: ac25a83d
Final MD5 hash: bcbcdcf4c9b8542075fc6b7ac25a83d

```

*Can be viewed upon running the MD5 hash function*

## SHA-1

SHA stands for secure hashing algorithm. SHA-1 is a cryptographic hashing function, which is a modified version of MD5. It is used for hashing data, certificate files, and other cryptographic purposes including cryptocurrencies such as bitcoin. SHA-1 which was developed in 1993 by the U.S. government's standards agency National Institute of Standards and Technology (NIST). SHA-1 takes an input string of length between  $1 < 2^{64}$  and produces a 160-bit hash value, commonly known as a message digest. SHA-1 produces a message digest based on principles similar to MD5, but generates a larger hash value (160 bits vs. 128 bits).

Breakingdown SHA-1, the first step is the initialization of the following buffers. These initial values are set by the SHA-1 algorithm are shown below in hex.

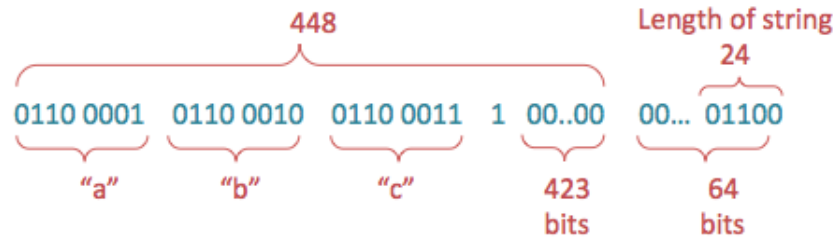
```

hash[0] = 0x67452301;
hash[1] = 0xefcdab89;
hash[2] = 0x98badcfe;
hash[3] = 0x10325476;
hash[4] = 0xc3d2e1f0;

```

*Can be viewed in SHA1.cpp, function initialize*

After this initialization, SHA-1 pads the inputted string, similar to the padding which occurs in MD5. The message is first padded by appending a 1 to the end of the string. Then 0's are appended to the end of the message until the length is 448 bits. Finally, 64 more bits are appended to the end, which represent the binary format of 64 bits indicating the length of the original message. This creates a block of 512 bits.



To visualize this concept, the image above depicts the breakdown of the 512 bit chunk when the string “abc” is entered. The string, “abc” is first converted into binary, then padded with a 1 and remaining 0’s. In this example, 423 0’s were appended to the message to make it 448 bits long. Then, the last 64 bits contain the length of the message in binary, as illustrated above being 24 bits long for the message “abc”.

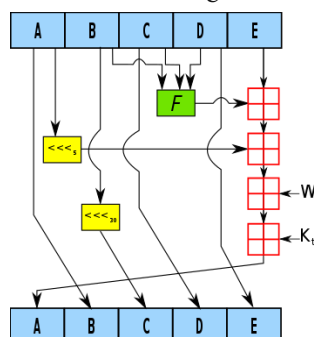
Once the message block of 512 bits is determined, it is broken down into 16 parts, each 32 bits. The 16 parts are often labeled as  $W(t)$  and they are passed to what is known as the compression functions,  $f(t)$ . SHA-1 requires 80 iterations of the processing functions defined by the following, where  $t$  is the number of rotations:

$$\begin{aligned}
 f(t;B,C,D) &= (B \text{ AND } C) \text{ OR } ((\text{NOT } B) \text{ AND } D) & (0 \leq t \leq 19) \\
 f(t;B,C,D) &= B \text{ XOR } C \text{ XOR } D & (20 \leq t \leq 39) \\
 f(t;B,C,D) &= (B \text{ AND } C) \text{ OR } (B \text{ AND } D) \text{ OR } (C \text{ AND } D) & (40 \leq t \leq 59) \\
 f(t;B,C,D) &= B \text{ XOR } C \text{ XOR } D & (60 \leq t \leq 79)
 \end{aligned}$$


SHA-1 also requires a processing constant for each of the 80 rotations. They are defined as:

$$\begin{aligned}
 K(t) &= 0x5A827999 & (0 \leq t \leq 19) \\
 K(t) &= 0x6ED9EBA1 & (20 \leq t \leq 39) \\
 K(t) &= 0x8F1BBCDC & (40 \leq t \leq 59) \\
 K(t) &= 0xCA62C1D6 & (60 \leq t \leq 79)
 \end{aligned}$$

Visualization one iteration of a compression function, gives the following graph:





A, B, C, D and E are 32-bit words of the state. F varies based on which iteration,  $t$ , and corresponds to one of the  $f(t)$  functions provided above. The yellow boxes denote a left bit rotation by  $n$  places, as  $n$  varies for each operation.  $W(t)$ , as mentioned beforehand, is the message block expanded into separate 32 bit pieces.  $K(t)$  is the round constant for iteration,  $t$ . The red blocks,  illustrate the addition of modulo  $2^{32}$ .

An implementation of the functions and constants combined in C++ code is as follows:

```
/* Help macros */
//these are defined by the SHA1 algorithm
#define SHA1_ROL(value, bits) (((value) << (bits)) | (((value) & 0xffffffff) >> (32 - (bits))))
#define SHA1_BLK(i) (block[i&15] = SHA1_ROL(block[(i+13)&15] ^ block[(i+8)&15] ^ block[(i+2)&15] ^ block[i&15],1))

/* (R0+R1), R2, R3, R4 are the different operations used in SHA1 */
//these are defined by the SHA1 algorithm
#define SHA1_ROUND_0(v,w,x,y,z,i) z += ((w&(x^y))^y) + block[i] + 0x5a827999 + SHA1_ROL(v,5); w=SHA1_ROL(w,30);
#define SHA1_ROUND_1(v,w,x,y,z,i) z += ((w&(x^y))^y) + SHA1_BLK(i) + 0x5a827999 + SHA1_ROL(v,5); w=SHA1_ROL(w,30);
#define SHA1_ROUND_2(v,w,x,y,z,i) z += (w^x^y) + SHA1_BLK(i) + 0x6ed9eba1 + SHA1_ROL(v,5); w=SHA1_ROL(w,30);
#define SHA1_ROUND_3(v,w,x,y,z,i) z += (((w|x)&y)|(w&x)) + SHA1_BLK(i) + 0x8f1bbcdc + SHA1_ROL(v,5); w=SHA1_ROL(w,30);
#define SHA1_ROUND_4(v,w,x,y,z,i) z += (w^x^y) + SHA1_BLK(i) + 0xca62c1d6 + SHA1_ROL(v,5); w=SHA1_ROL(w,30);
```

*Can be viewed in SHA1.cpp*

This code block defines each compression function as `SHA1_ROUND`, the number following corresponds to which round of rotations it occurs based on the iterations. These functions also include the constants,  $K(t)$ , for each of its corresponding iterations. `SHA_ROL` and `SHA_BLK` help the function perform the necessary shifts.

Then, the function transform executes all 80 iterations for the above functions. First, this function initializes a, b, c, d, e to correspond to the buffers that were presented above in 'hash[]' in order to split up what will become the full 160 bit hash.

```
/* Copy hash[] to working vars */
unsigned long int a = hash[0];
unsigned long int b = hash[1];
unsigned long int c = hash[2];
unsigned long int d = hash[3];
unsigned long int e = hash[4];
```

*Can be viewed in SHA1.cpp, function transform*

Afterwards, the function transform performs all 80 rotations on a, b, c, d, e.

```

SHA1_ROUND_0( a, b, c, d, e,  0 );
SHA1_ROUND_0( e, a, b, c, d,  1 );
SHA1_ROUND_0( d, e, a, b, c,  2 );
    .
    .
    .
SHA1_ROUND_0( a, b, c, d, e, 15 );
SHA1_ROUND_1( e, a, b, c, d, 16 );
SHA1_ROUND_1( d, e, a, b, c, 17 );
    .
    .
    .
SHA1_ROUND_1( b, c, d, e, a, 19 );
SHA1_ROUND_2( a, b, c, d, e, 20 );
SHA1_ROUND_2( e, a, b, c, d, 21 );
    .
    .
    .
SHA1_ROUND_2( b, c, d, e, a, 39 );
SHA1_ROUND_3( a, b, c, d, e, 40 );
SHA1_ROUND_3( e, a, b, c, d, 41 );
    .
    .
    .
SHA1_ROUND_3( b, c, d, e, a, 59 );
SHA1_ROUND_4( a, b, c, d, e, 60 );
SHA1_ROUND_4( e, a, b, c, d, 61 );
    .
    .
    .
SHA1_ROUND_4( d, e, a, b, c, 77 );
SHA1_ROUND_4( c, d, e, a, b, 78 );
SHA1_ROUND_4( b, c, d, e, a, 79 );

```

*Can be viewed in SHA1.cpp, function transform*

The values of a, b, c, d, and e are also displayed after each 20 iterations to illustrate the process of the hash. After the 20 rounds of each function, the values of a, b, c, d, and e are added back to the original buffer values stored in “hash[]”.

```

hash[0] += a;
hash[1] += b;
hash[2] += c;
hash[3] += d;
hash[4] += e;

```

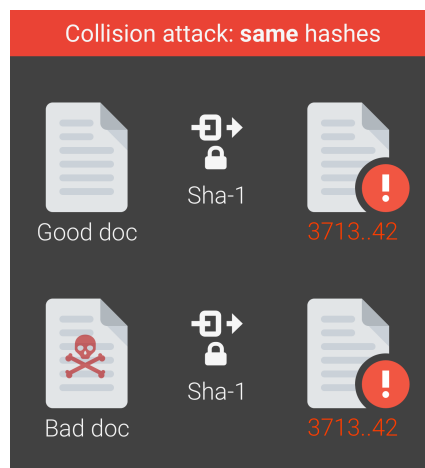
*Can be viewed in SHA1.cpp, function transform*

The final step is to concatenate these 5 values together to get the final hash value. This is the complete process of SHA-1. This implementation contains the steps which allows the user to visually see how the hash occurs.

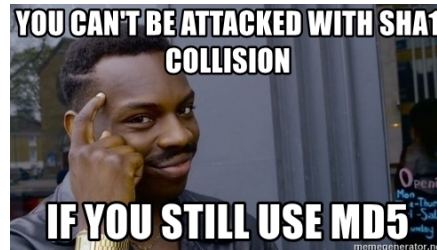
While SHA-1 is an improvement from MD5, which has been declared cryptographically broken, it is not as secure as needed. Since 2005, SHA-1 has not been considered secure and many organizations say it should be replaced. The National Institute of Standards and Technology formally deprecated use of SHA-1 in 2011 and disallowed its use for digital signatures in 2013. Therefore, replacing SHA-1 in digital signatures is urgent. Due to this, it is recommended to remove SHA-1 from products as soon as possible and instead use SHA-2.



On February 27th, 2017, Google announced SHAttered, the first-ever crafted collision for SHA-1. Google was able to create a PDF file that had the same SHA-1 hash as another PDF file, despite having different content. This official publication of a practical attack on SHA-1 was an effort to increase awareness and convince the industry to quickly move to other algorithms, such as SHA-256. The SHAttered attack is 100,000 faster than the brute force attack that relies on the birthday paradox. For reference, the birthday paradox is that if there are 23 people in a room, the probability of a shared birthday exceeds 50%. The brute force attack on SHA-1 requires 12,000,000 GPU years to complete, which is impractical to complete.



From the SHAttered attack, Google was able to reinforce that relying on SHA-1 for digital signatures, file integrity, or file identification is potentially vulnerable. The vulnerabilities found within SHA-1 illustrate why there is a need to switch to SHA-256, due to it being more secure.



### SHA-256

SHA-2 was developed shortly after the discovery of cost-effective brute force attacks against SHA-1. It is a family of two similar hash functions, with different block sizes, known as SHA-256 and SHA-512. SHA-2 can be cracked like SHA-1. Because of the short length of the hash digest, SHA-1 is more easily cracked by brute force than SHA-2, but SHA-2 can still be cracked by brute force. Another drawback of SHA-1 is that two different values can have the same hash digest, because there are not enough combinations that can be created with 160 bits. Collisions occur when two values have the same hash digest. SHA-1 can easily create collisions, which makes it easier for attackers to get two matching digests and recreate the original plaintext. Conversely, in SHA-2, every digest has a unique value, because SHA-2 can produce a variety of bit-lengths, from 256 to 512 bit. Compared to SHA-1, SHA-2 is much more secure and, as of 2016, is required in all digital signatures and certificates. Brute force attacks can take years or even decades to crack the hash digest, so SHA-2 is considered the most secure hash algorithm.

The most widely used hashing algorithm today, SHA-256 was published in 2001. In SHA-256, messages up to  $2^{64}$  bits are transformed into digests of size 256 bits. SHA-256 begins with appending bits to the original message such that after the addition of these bits the length of the message should be 64 bits less than a multiple of 512, exactly the same as in SHA-1. The first bit appended is '1' and the rest are '0'. Next, the length bits, which are equivalent to 64 bits, are appended to the overall message to make it an exact multiple of 512 bits, as seen in the image below. The length bits are calculated by taking the modulo of the original message without the padding with  $2^{32}$ .



Next, the buffers are initialized. These buffers are initialized so that the first round of hashing does not start with zeroes, which would make this hashing broken.

```
// Initial Hash Values, first thirty-two bits of the fractional parts of
// the square roots of the first eight prime numbers.
unsigned long static H0 = 0x6a09e667;
unsigned long static H1 = 0xbb67ae85;
unsigned long static H2 = 0x3c6ef372;
unsigned long static H3 = 0xa54ff53a;
unsigned long static H4 = 0x510e527f;
unsigned long static H5 = 0x9b05688c;
unsigned long static H6 = 0x1f83d9ab;
unsigned long static H7 = 0x5be0cd19;

unsigned long W[64];
```

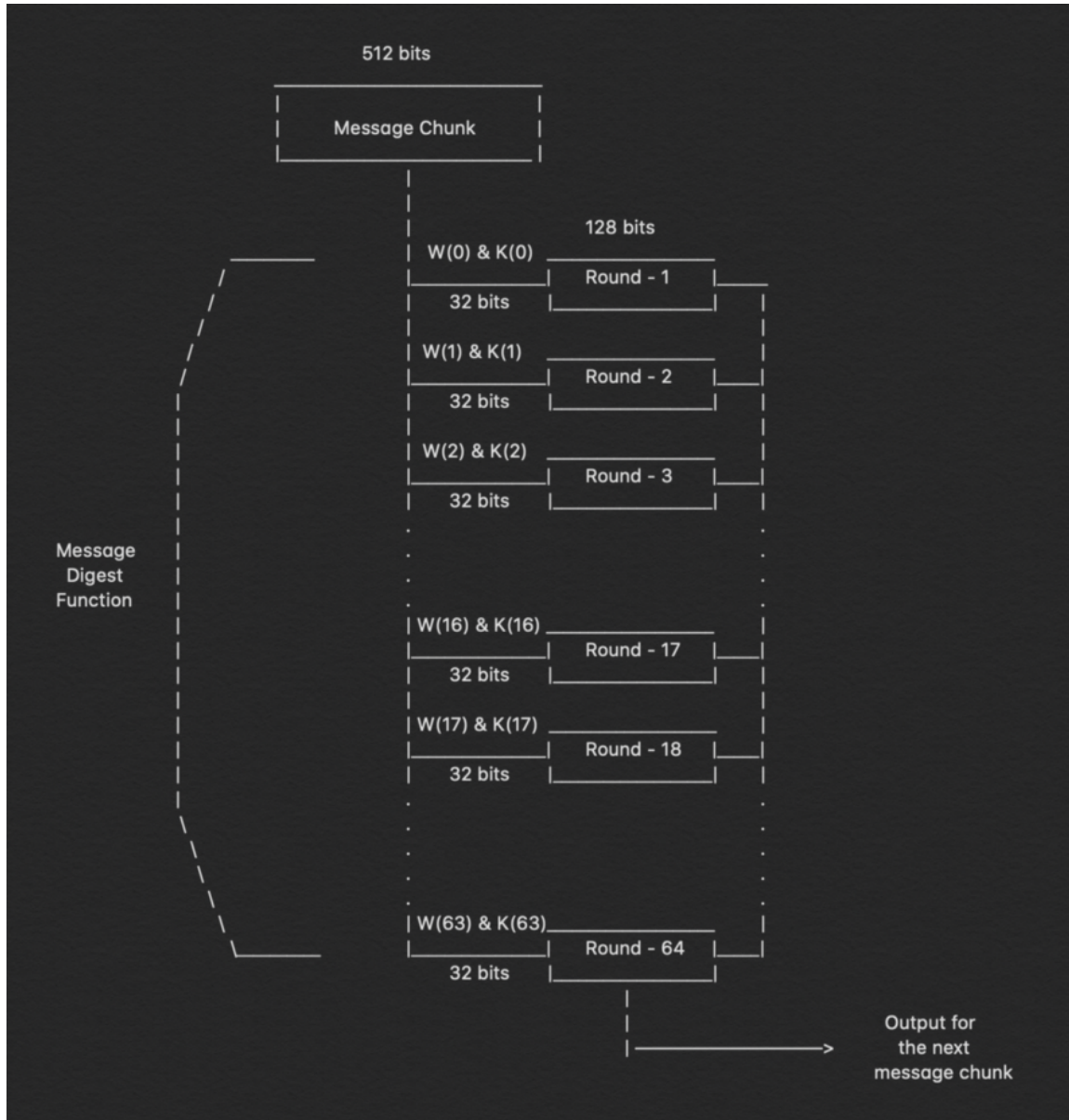
*Can be viewed in SHA256.cpp, function compute\_hash*

There are also 64 keys that are initialized.

```
// These words represent the first 32
// bits of the fractional parts of the cube roots of the first sixty-
// four prime numbers.
unsigned long k[64] = {
    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
};
```

*Can be viewed in SHA256.cpp, function compute\_hash*

The entire message block that is ' $n \times 512$ ' bits long is divided into ' $n$ ' chunks of 512 bits and each of these 512 bits is put through 64 rounds of operations. The output obtained is sent as input for the next round of operation, as depicted in the image below.



The two inputs sent in are  $W(i)$  and  $K(i)$ . As seen in the code, for the first 16 rounds, the 512-bit message is further broken down into 16 parts each of 32 bits, but after that the value of  $W(i)$  needs to be calculated at each step, using right rotations and XOR.



```

for(int t = 0; t <= 15; t++)
{
    W[t] = block[t] & 0xFFFFFFFF;

    if (show_Wt)
        std::cout << "W[" << t << "]: 0x" << show_as_hex(W[t]) << std::endl;
}

for(int t = 16; t <= 63; t++)
{
    W[t] = SSIG1(W[t-2]) + W[t-7] + SSIG0(W[t-15]) + W[t-16];

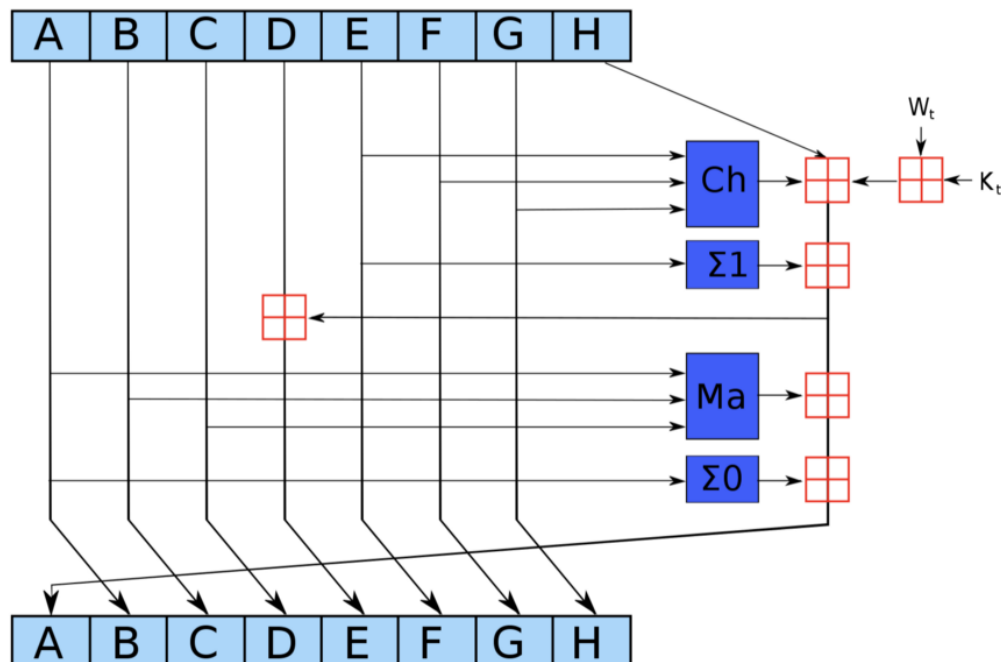
    // Have to make sure numbers are still 32 bits
    W[t] = W[t] & 0xFFFFFFFF;

    if (show_Wt)
        std::cout << "W[" << t << "]: " << W[t] << std::endl;
}

```

*Can be viewed in SHA256.cpp, function compute\_hash*

The following image depicts the 64 rounds of calculations performed on  $W(i)$ .



These calculations are depicted in the code below.

```
for( int t = 0; t < 64; t++)
{
    temp1 = h + EP1(e) + CH(e,f,g) + k[t] + W[t];
    if ((t == 20) & show_T1_calculation)
    {
        std::cout << "\nh: 0x" << std::hex << h << " dec:" << std::dec << h
            << " sign:" << std::dec << (int)h << "\n" << std::endl;
        std::cout << "EP1(e): 0x" << std::hex << EP1(e) << " dec:"
            << std::dec << EP1(e) << " sign:" << std::dec << (int)EP1(e)
            << "\n" << std::endl;
        std::cout << "CH(e,f,g): 0x" << std::hex << CH(e,f,g) << " dec:"
            << std::dec << CH(e,f,g) << " sign:" << std::dec
            << (int)CH(e,f,g) << "\n" << std::endl;
        std::cout << "k[t]: 0x" << std::hex << k[t] << " dec:" << std::dec
            << k[t] << " sign:" << std::dec << (int)k[t] << "\n" << std::endl;
        std::cout << "W[t]: 0x" << std::hex << W[t] << " dec:" << std::dec
            << W[t] << " sign:" << std::dec << (int)W[t] << "\n" << std::endl;
        std::cout << "temp1: 0x" << std::hex << temp1 << " dec:" << std::dec
            << temp1 << " sign:" << std::dec << (int)temp1 << "\n" << std::endl;
    }

    temp2 = EP0(a) + MAJ(a,b,c);

    // Shows the variables and operations for getting T2.
    if ((t == 20) & show_T2_calculation)
    {
        std::cout << "a: 0x" << std::hex << a << " dec:" << std::dec << a
            << " sign:" << std::dec << (int)a << "\n" << std::endl;
        std::cout << "b: 0x" << std::hex << b << " dec:" << std::dec << b
            << " sign:" << std::dec << (int)b << "\n" << std::endl;
        std::cout << "c: 0x" << std::hex << c << " dec:" << std::dec << c
            << " sign:" << std::dec << (int)c << "\n" << std::endl;
        std::cout << "EP0(a): 0x" << std::hex << EP0(a) << " dec:"
            << std::dec << EP0(a) << " sign:" << std::dec << (int)EP0(a)
            << "\n" << std::endl;
        std::cout << "MAJ(a,b,c): 0x" << std::hex << MAJ(a,b,c) << " dec:"
            << std::dec << MAJ(a,b,c) << " sign:" << std::dec
            << (int)MAJ(a,b,c) << "\n" << std::endl;
        std::cout << "temp2: 0x" << std::hex << temp2 << " dec:" << std::dec
            << temp2 << " sign:" << std::dec << (int)temp2 << "\n" << std::endl;
    }

    h = g;
    g = f;
    f = e;
    e = (d + temp1) & 0xFFFFFFFF; // Makes sure that we are still using 32 bits.
    d = c;
    c = b;
    b = a;
    a = (temp1 + temp2) & 0xFFFFFFFF; // Makes sure that we are still using 32 bits.
}
```

*Can be viewed in SHA256.cpp, function compute\_hash*



Finally, all of the hash values are added with their respective variables, a-h, and masked with 0xFFFFFFFF to ensure that the hash values are still 32 bits.

```
// Add up all the working variables to each hash and make sure we are still
// working with solely 32 bit variables.
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;
```

*Can be viewed in SHA256.cpp, function compute\_hash*

Finally, the hash values are concatenated to create the final hash value for the input.

```
// Append the hash segments together one after the other to get the full
// 256 bit hash.
return show_as_hex(H0) + show_as_hex(H1) + show_as_hex(H2) +
       show_as_hex(H3) + show_as_hex(H4) + show_as_hex(H5) +
       show_as_hex(H6) + show_as_hex(H7);
```

*Can be viewed in SHA256.cpp, function compute\_hash*

SHA-256 is used in some of the most popular authentication and encryption protocols, including SSL, TLS, IPsec, SSH, and PGP. Linux and Unix use SHA-256 for secure password hashing. Cryptocurrencies such as Bitcoin use SHA-256 for verifying transactions.

Currently, SHA-2 is the industry standard for hashing algorithms, but SHA-3 may replace it in the future. The NIST released SHA-3 in 2015, but it was not made the industry standard. During the release of SHA-3, most companies were in the middle of transitioning from SHA-1 to SHA-2, so switching to SHA-3 did not make sense. In addition, SHA-3 was considered slower than SHA-2, although this is not entirely true. SHA-3 is slower on the software side, but it is much faster than SHA-1 and SHA-2 on the hardware side, and it is still getting faster. For this reason, SHA-3 will likely replace SHA-2 in the future, once SHA-2 becomes unsafe or deprecated.

## Works Consulted

- Anand, Aditya. "Breaking Down: SHA-1 Algorithm." *Infosec Writeups*, 6 Nov. 2019, <https://infosecwriteups.com/breaking-down-sha-1-algorithm-c152ed353de2>.
- Crane, Casey. "What Is a Hash Function in Cryptography? A Beginner's Guide." *Hashed Out by The SSL Store*, 25 Jan. 2021, <https://www.thesslstore.com/blog/what-is-a-hash-function-in-cryptography-a-beginners-guide/>.
- "Cryptography Hash Functions." *TutorialsPoint*, [www.tutorialspoint.com/cryptography/cryptography\\_hash\\_functions.htm](http://www.tutorialspoint.com/cryptography/cryptography_hash_functions.htm).
- "Data Encryption Standard (DES) | Set 1." *Geeks for Geeks*, 8 Nov. 2021, <https://www.geeksforgeeks.org/data-encryption-standard-des-set-1/>.
- "Data Encryption Standard." *Wikipedia*, 21 Oct. 2021, [https://en.wikipedia.org/wiki/Data\\_Encryption\\_Standard#Brute-force\\_attack](https://en.wikipedia.org/wiki/Data_Encryption_Standard#Brute-force_attack).
- Eastlake, Jones, et al. "US Secure Hash Algorithm 1 (SHA1)." *Cisco Systems*, September 2001, <https://www.ipa.go.jp/security/rfc/RFC3174EN.html>.
- Lake, Josh. "What is 3DES encryption and how does DES work?" *Comparitech*, 20 Feb. 2019, <https://www.comparitech.com/blog/information-security/3des-encryption/>.
- "MD5 Algorithm: Know Working and Uses of MD5 Algorithm." *EDUCBA*, 6 Mar. 2021, [www.educba.com/md5-algorithm/](http://www.educba.com/md5-algorithm/).
- "MD5." *Wikipedia*, 30 Nov. 2021, <https://en.wikipedia.org/wiki/MD5>.
- "Message-Digest Algorithm 5." *ScienceDirect Topics*, <https://www.sciencedirect.com/topics/computer-science/message-digest-algorithm-5>.
- Puneet. "What Is SHA, and What is SHA used for?" *Encryption Consulting*, 18 Nov. 2021, [www.encryptionconsulting.com/education-center/what-is-sha/](http://www.encryptionconsulting.com/education-center/what-is-sha/).
- "Secure Hash Algorithms." *Brilliant.org*, 4 December 2021, <https://brilliant.org/wiki/secure-hashing-algorithms/>.
- Shacklett, Mary E., and Peter Loshin. "What Is MD5 (MD5 Message-Digest Algorithm)?" *SearchSecurity*, 23 Aug. 2021, <https://www.techtarget.com/searchsecurity/definition/MD5>.

“SHA1.” *Wikipedia*, 27 Nov. 2021, <https://en.wikipedia.org/wiki/SHA-1#>.

Simmons, Gustavus. “Data Encryption Standard.” *Britannica*,  
<https://www.britannica.com/topic/Data-Encryption-Standard>.

“Triple DES.” *Wikipedia*, 1 Jan. 2021, [https://simple.wikipedia.org/wiki/Triple\\_DES](https://simple.wikipedia.org/wiki/Triple_DES).

“What is the DES Algorithm?” *Educative*,  
<https://www.educative.io/edpresso/what-is-the-des-algorithm>.

“What is DES (Data Encryption Standard): DES Algorithm and Operation.” *Simplilearn*, 26 Oct. 2021, <https://www.simplilearn.com/what-is-des-article>.

Yang, Herong. “SHA1 Message Digest Algorithm Overview.” *Herong’s Tutorial Examples*, 2021,  
<http://www.herongyang.com/Cryptography/SHA1-Message-Digest-Algorithm-Overview.html>.

“You can’t be attacked with SHA1 collisions if you use MD5.” *Meme Generator*,  
<https://memegenerator.net/instance/75808711/thinking-black-guy-you-cant-be-attacked-with-sha1-collision-if-you-still-use-md5>.