Trường Đại học Bách Khoa
TPHCM - ĐHQG TPHCM

Cryptography

Assignments

MD5 & SHA1

Đinh Thanh Phong - 2270243

Introduction

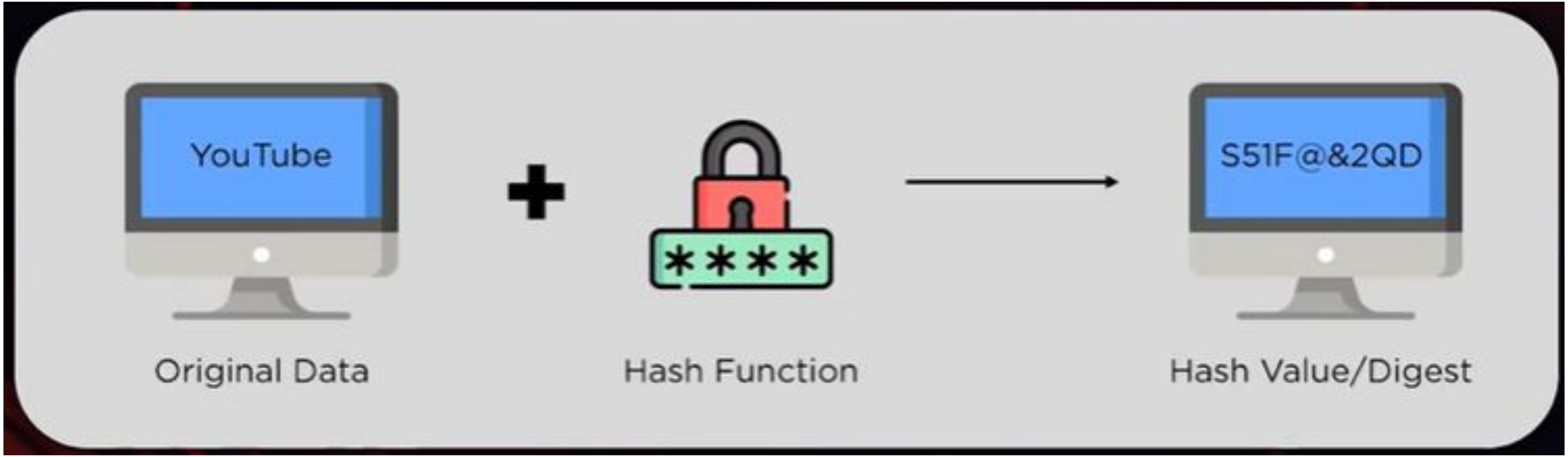
<table>
<tr><th colspan="2">Hash</th></tr>
<tr><td>MD5</td><td>SHA1</td></tr>
<tr><td>MD5 Hash Algorithm in Cryptography: Here's Everything You Should Know (simplilearn.com)
The MD5 algorithm (with examples) | Comparitech</td><td>https://www.youtube.com/watch?v=kmHojGMUn0Q</td></tr>
<tr><td>

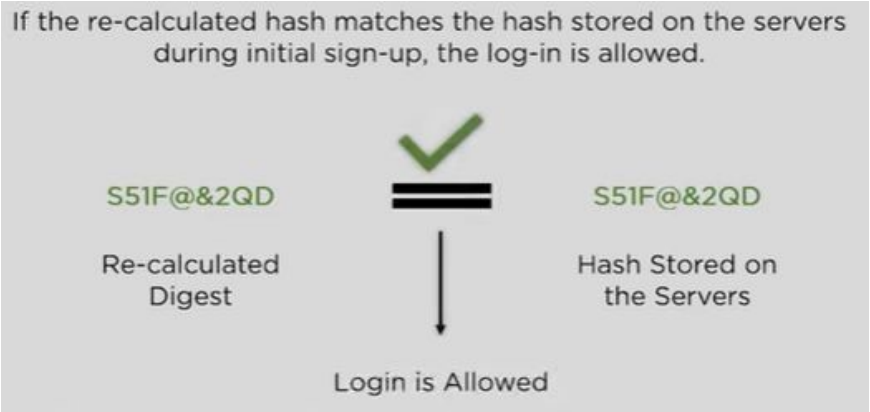

</td><td>

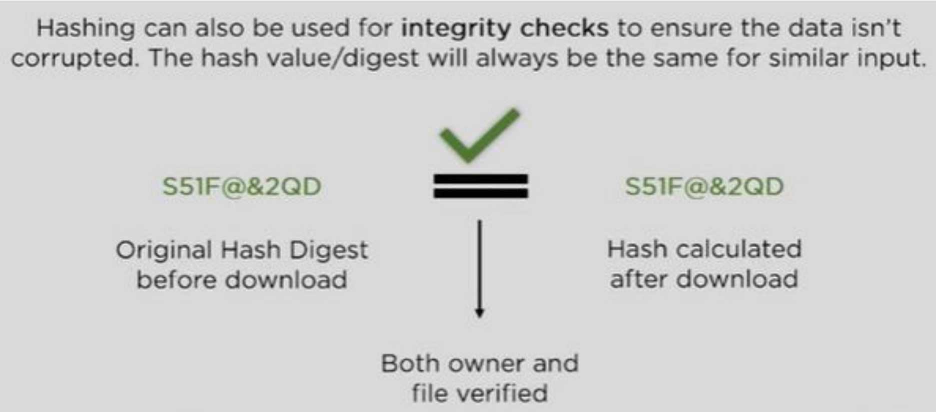

</td></tr>
</table>

Introduction

Hash Function







Hash Function

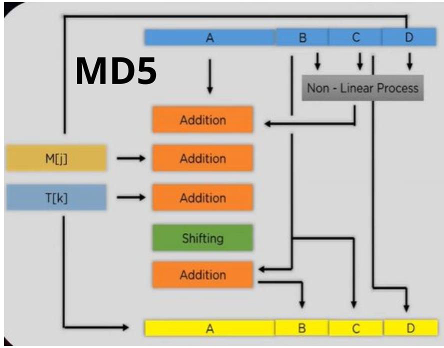
Tính chất của Hash
Deterministic: đối với chuỗi ký tự thì luôn đưa ra 1 kết quả sau hash
Irreversible: Không thể đảo ngược, từ Hash không thể tính ngược lại Value. Chỉ encrypt mà không thể decrypt
Utilize the "avalanche effect": Khi đưa 2 chuỗi gần giống nhau vào hash, thì sẽ ra kết quả hoàn toàn khác hẳn nhau.
Collision-resistant: Gần như không thể tìm 2 chuỗi khác nhau mà có cùng hash giống nhau

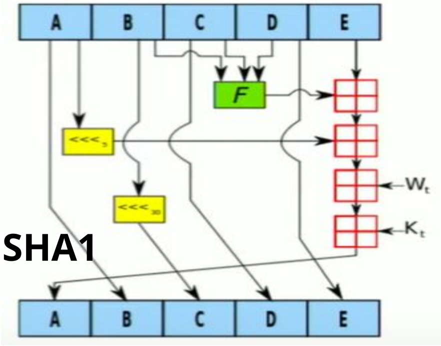
Tóm lược
Pre-image resistance: Không thể đảo ngược dù biết cả Value và Hash
Second pre-image resistance: Không thể tìm chuỗi ký tự khác có cùng hash
Unbreakable without brute force: chỉ có thể tấn công bằng vét cạn
One-way: không thể decrypt

Introduction

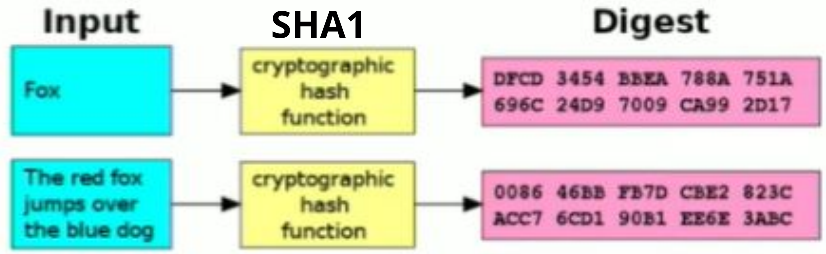
MD5 & SHA1

Comparison	MD5	SHA1
Security	Less Secure than SHA	High Secure than MD5
Message Digest Length	128 Bits	160 Bits
Attacks required to find out original Message	2^{128} bit operations required to break	2^{160} bit operations required to break
Attacks to try and find two messages producing the same MD	2^{64} bit operations required to break	2^{80} bit operations required to break
Speed	Faster, only 64 iterations	Slower than MD5, Required 80 iterations
Successful attacks so far	Attacks reported to some extents	No such attach report yet

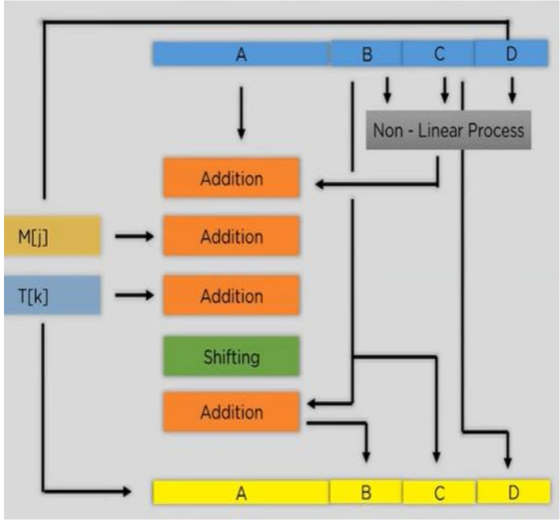




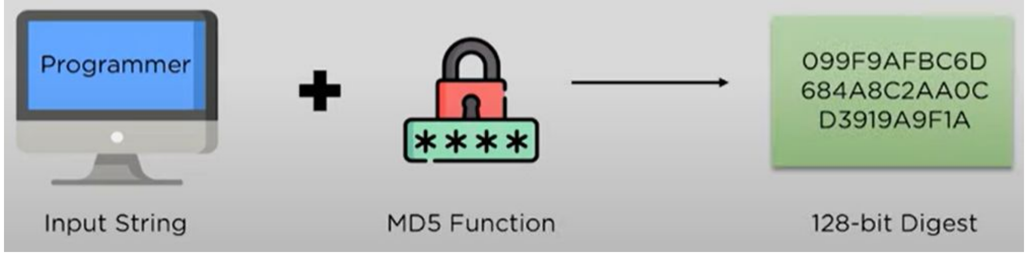
Input into Hash Function	MD5 Hash Value/Digest
Cryptography	d2fc0657a64a3291826136c7712abbe7
Cryptographyabc123	c56db83ab5482b4e94536f4a29b21de0
Cryptographyxyz456	783b10b483435e05f3f2705bdd5a825c



MD5 How it works?

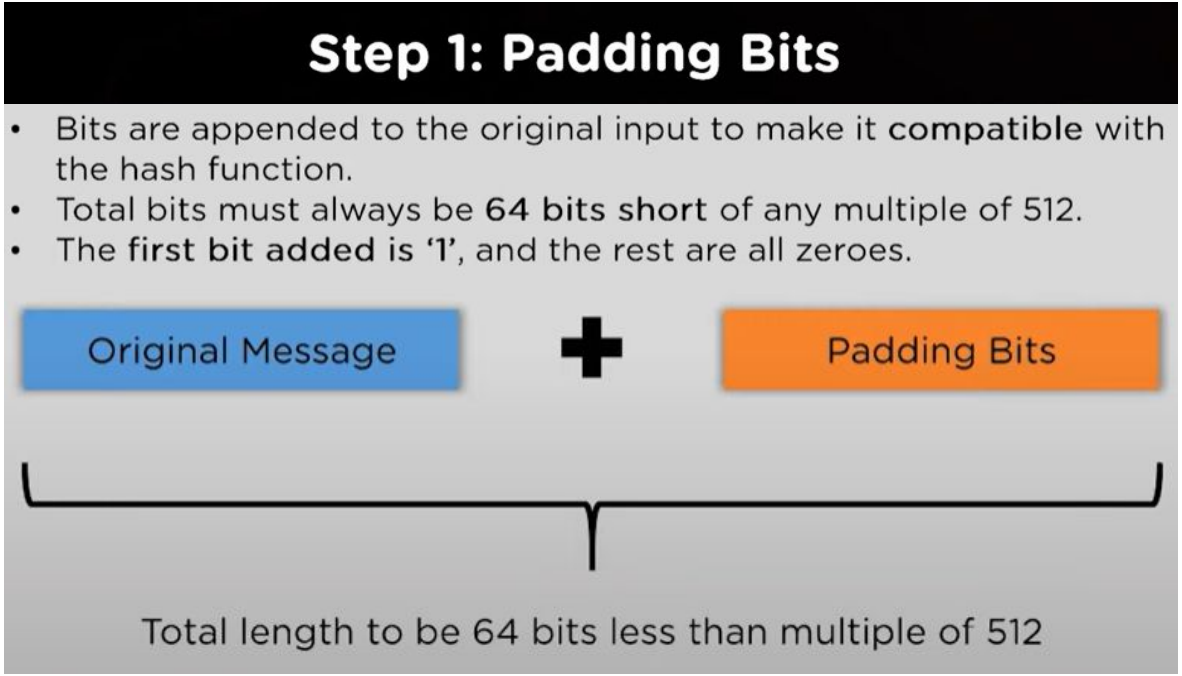


- One-way cryptographic hash function
- 128-bit digest size for every single input
- Initially designed for **digital signatures**
- Designed in **1991** by Ronald Rivest



Input into Hash Function	MD5 Hash Value/Digest
Cryptography	d2fc0657a64a3291826136c7712abbe7
Cryptographyabc123	c56db83ab5482b4e94536f4a29b21de0
Cryptographyxyz456	783b10b483435e05f3f2705bdd5a825c

MD5 How it works?



Using an ASCII table, we see that a capital letter "T" is written as "01010100" in binary. A lowercase "h" is "01101000", a lowercase "e" is "01100101", and a lowercase "y" is "01111001". The binary code for a space (SP) is "00100000". You can see it in the table at the top of the second column, in line with the decimal number 32.

If we continue on in this fashion, we see that our input, "They are deterministic" is written in binary as:

01010100 01101000 01100101 01111001 00100000 01100001
01110010 01100101 00100000 01100100 01100101 01110100
01100101 01110010 01101101 01101001 01101110 01101001
01110011 01110100 01101001 01100011

448 – 1 – 176 = 271

Therefore **the padding for this block will include a one, then an extra 271 zeros**. The reason we only need to pad it up to 448 bits (instead of 512) is because the final 64 bits (512 – 64 = 448) are reserved to display the message's length in binary. In this case, the number 176 is 10110000 in binary. This forms the very end of the padding scheme, while the preceding 56 bits (64 minus the eight bits that make up 10110000) are all filled up with zeros.

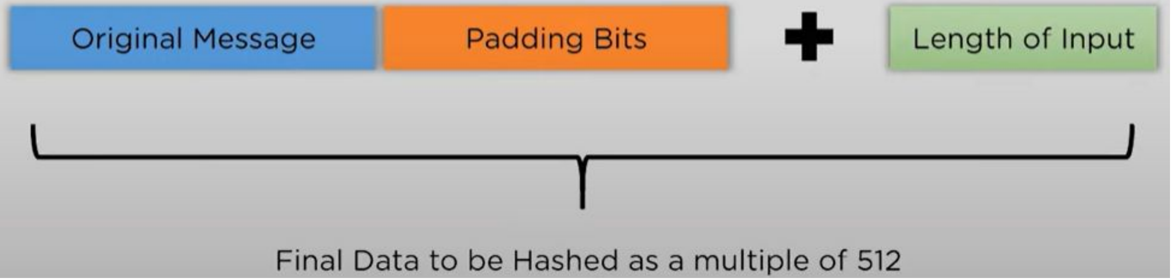
Once the padding scheme is complete, we end up with the following 512-bit string:

01010100 01101000 01100101 01111001 00100000 01100001
01110010 01100101 00100000 01100100 01100101 01110100
01100101 01110010 01101101 01101001 01101110 01101001
01110011 01110100 01101001 01100011 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 10110000

MD5 How it works?

Step 2: Padding Length

- **Length** of the original message is padded to the result from step 1.
- Length is expressed in the form of **64 bits.**
- Resultant string will now be a **multiple of 512**.
- Used to increase **complexity** of the function.



Using an ASCII table, we see that a capital letter "T" is written as "01010100" in binary. A lowercase "h" is "01101000", a lowercase "e" is "01100101", and a lowercase "y" is "01111001". The binary code for a space (SP) is "00100000". You can see it in the table at the top of the second column, in line with the decimal number 32.

If we continue on in this fashion, we see that our input, "They are deterministic" is written in binary as:

01010100 01101000 01100101 01111001 00100000 01100001
01110010 01100101 00100000 01100100 01100101 01110100
01100101 01110010 01101101 01101001 01101110 01101001
01110011 01110100 01101001 01100011

448 – 1 – 176 = 271

Therefore **the padding for this block will include a one, then an extra 271 zeros**. The reason we only need to pad it up to 448 bits (instead of 512) is because the final 64 bits (512 – 64 = 448) are reserved to display the message's length in binary. In this case, the number 176 is 10110000 in binary. This forms the very end of the padding scheme, while the preceding 56 bits (64 minus the eight bits that make up 10110000) are all filled up with zeros.

Once the padding scheme is complete, we end up with the following 512-bit string:

01010100 01101000 01100101 01111001 00100000 01100001
01110010 01100101 00100000 01100100 01100101 01110100
01100101 01110010 01101101 01101001 01101110 01101001
01110011 01110100 01101001 01100011 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 10110000

MD5 How it works?

Step 3: Initialize MD Buffer

- The entire message is broken down into blocks of **512 bits** each.
- 4 buffers are used of **32 bits** each.
- They are **4 words** named A, B, C and D.
- The first iteration has **fixed** hexadecimal values.

A = 01 23 45 67	B = 89 ab cd ef
C = fe dc ba 98	D = 76 54 32 10

The MD5 algorithm's initialization vectors

At the beginning, the initialization vectors are four separate numbers, specified in the RFC that outlines the MD5 standard. These are:

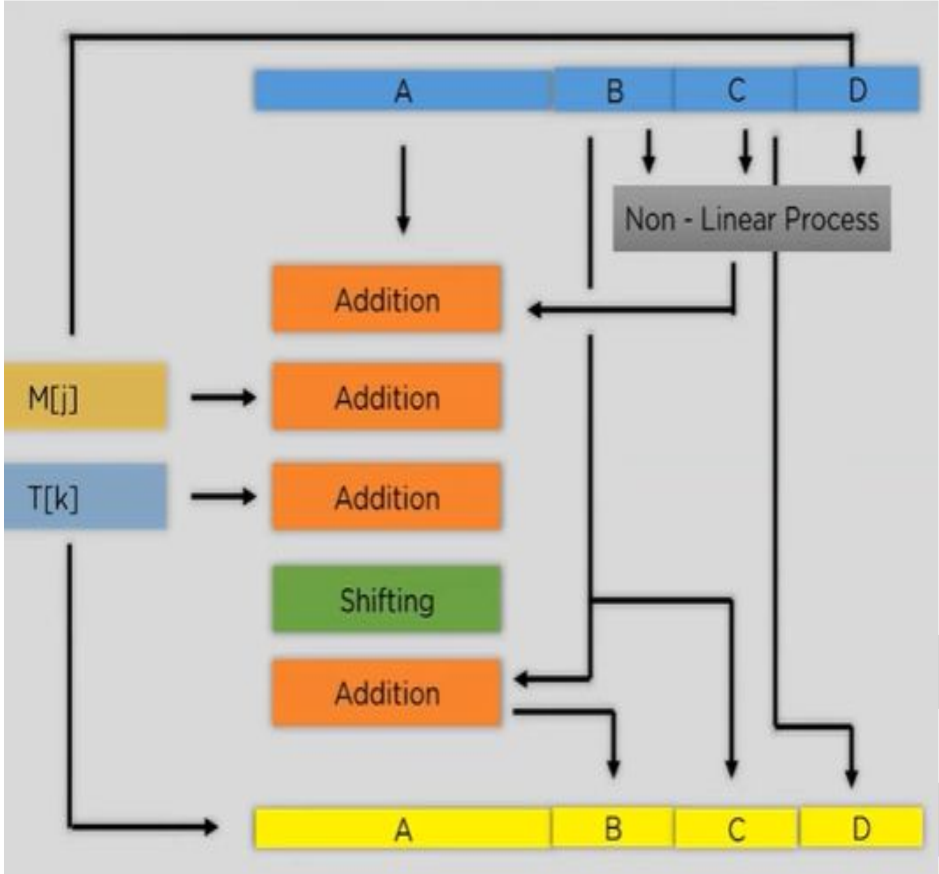
- A – 01234567
- B – 89abcdef
- C – fedcba98
- D – 76543210

MD5 How it works?

Step 4: Process Each Block

- Each block is broken down to **16 sub blocks** of 32 bit each.
- There are **4 rounds** of operations, each of them utilizing all 16 sub blocks, the 4 buffers and other constants.
- The constant value is an **array of 64 elements**, with 16 elements being used every round.
- Sub blocks : M[0], M[1], ... M[15]
- Constant array : T[1], T[2], ... T[64]

16 x 32 = 512 bits



MD5 How it works?

Non-Linear Process Function

- Different for each round.
- Used to increase randomness of the hash as an upgrade over MD4.

Round 1: (b AND c) OR ((NOT b) AND (d))

Round 2: (b AND d) OR (c AND (NOT d))

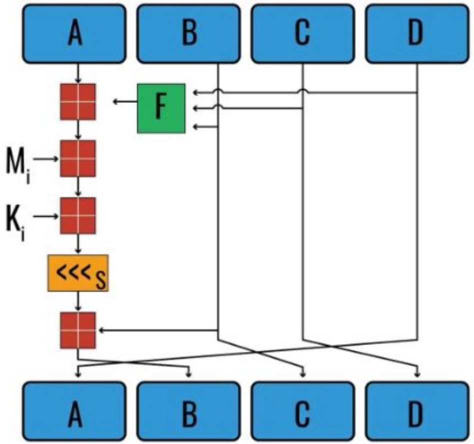
Round 3: b XOR c XOR d

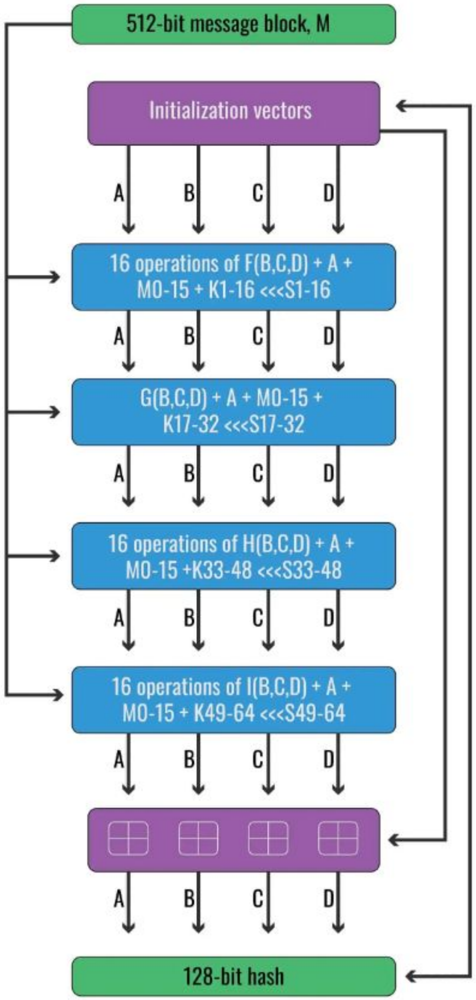
Round 4: c XOR (b OR (NOT d))

One of these K values is used in each of the 64 operations for a 512-bit block. K1 to K16 are used in the first round, K17 to K32 are used in the second round, K33 to K48 are used in the third round, and K49 to K64 are used in the fourth round.

After the K value has been added, the next step is to shift the number of bits to the left

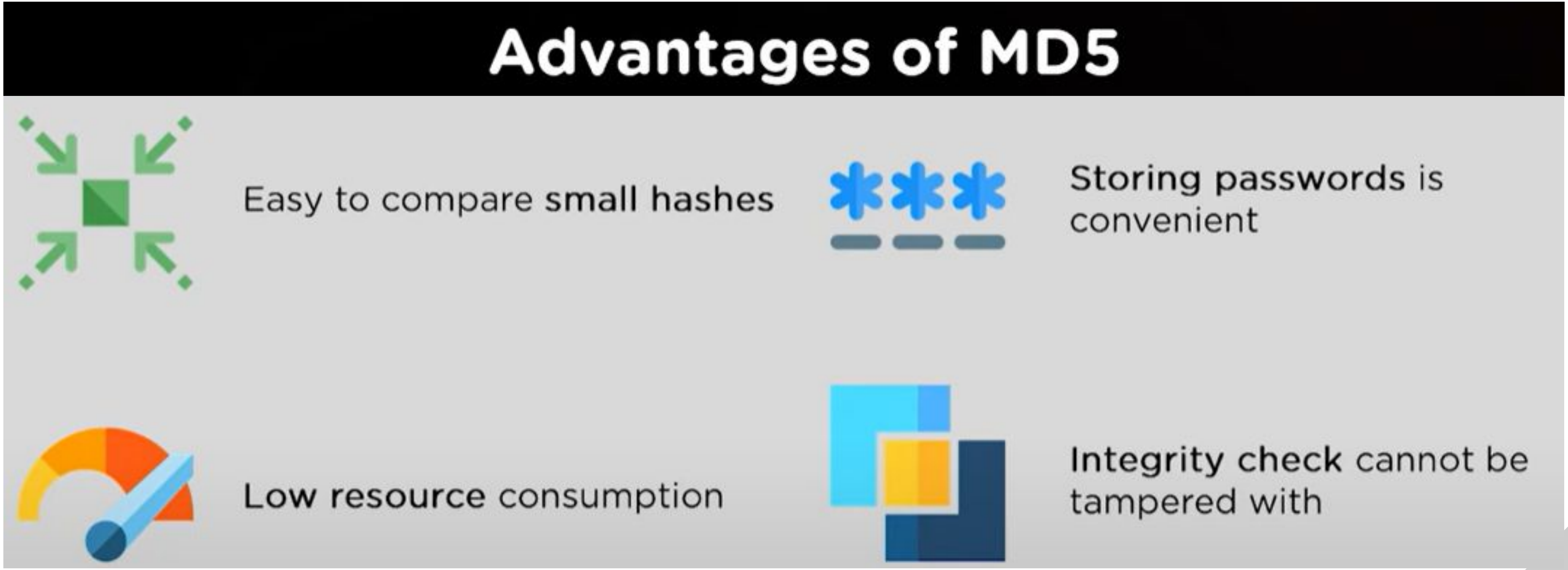
Hash = ABCD = **61f1141806fbee528a1a2bf59437d949**





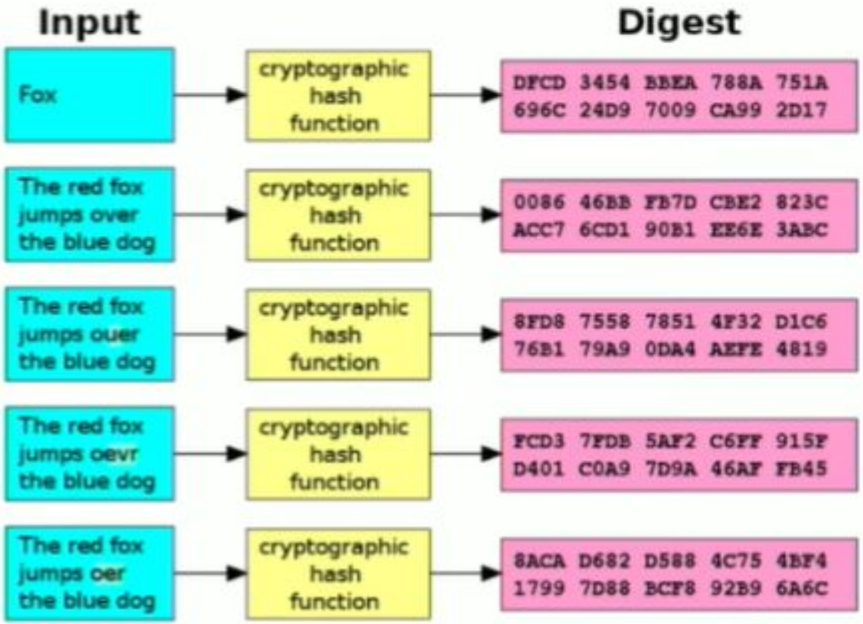
MD5

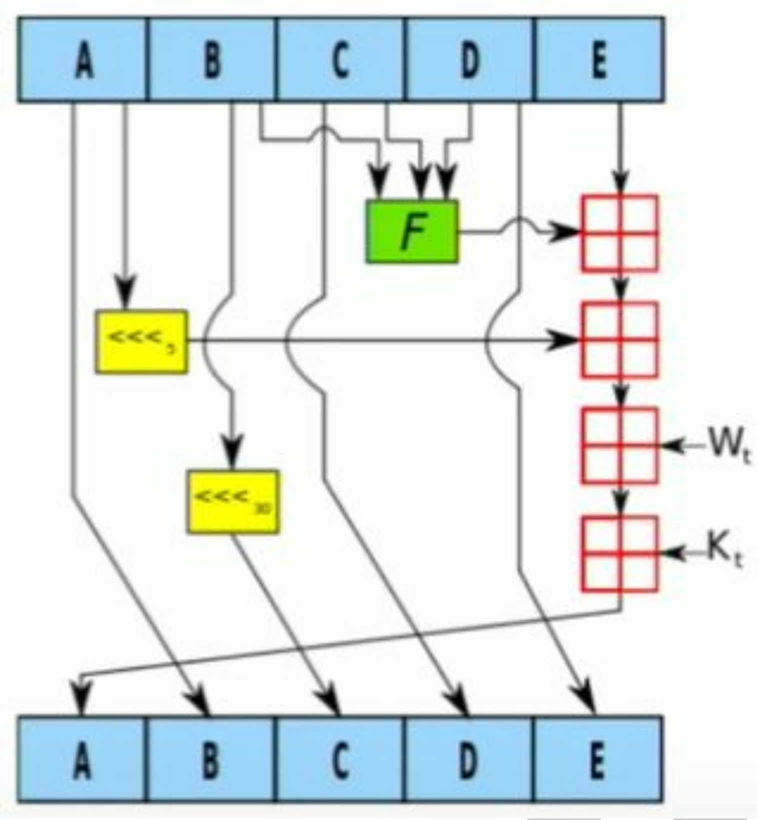
How it works?



SHA1

- Designed and published by the NSA
- Commonly used through the mid 2000s
- No longer considered secure
- Text → 40-digit hexadecimal





SHA1 How it works?

1. Take input text and split it into an array of the characters' ASCII codes

'A Test'

[A, , T, e, s, t]

```
function sha1(text) {
  const asciiText = text.split('')
    .map((letter) => utils.charToASCII(letter));
```

[65, 32, 84, 101, 115, 116]

2. Convert ASCII codes to binary
3. Pad zeros to the front of each until they are 8 bits long

[65, 32, 84, 101, 115, 116]

```
let binary8bit = asciiText
                .map((num) => utils.asciiToBinary(num))
                .map((num) => utils.padZero(num, 8));
```

[01000001, 00100000, 01010100, 01100101, 01110011, 01110100]

SHA1 How it works?

3. Pad zeros to the front of each until they are 8 bits long

[65, 32, 84, 101, 115, 116]

```
let binary8bit = asciiText
                .map((num) => utils.asciiToBinary(num))
                .map((num) => utils.padZero(num, 8));
```

[01000001, 00100000,
01010100, 01100101,
01110011, 01110100]

4. join and append a 1

```
let numString = binary8bit.join('') + '1';
```

0100000100100000010101000110
0101011100110111010001

SHA1 How it works?

4. join and append a 1

```
let numString = binary8bit.join('') + '1';
```

0100000100100000010101000110
0101011100110111101001

5. pad the binary message with zeros until its length is 512 mod 448

Đếm độ dài của chuỗi rồi chia cho 512; xem có dư 448 không? Nếu không, thì thêm số 0 vào cho đến khi “độ dài chia 512 dư 448”

```
while (numString.length % 512 !== 448) {
    numString += '0';
  }
```

01000001001000000101010001
1001010111001101110100 1

0100000100100000010101000110010101110011011101001000000000000000000-
000000000000000000000000000000000000000000000000000000000000000000000000-
000000000000000000000000000000000000000000000000000000000000000000000000-
000000000000000000000000000000000000000000000000000000000000000000000000-
000000000000000000000000000000000000000000000000000000000000000000000000-
000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000

SHA1 How it works?

3. Pad zeros to the front of each until they are 8 bits long

[65, 32, 84, 101, 115, 116]

```
let binary8bit = asciiText
                .map((num) => utils.asciiToBinary(num))
                .map((num) => utils.padZero(num, 8));
```

[01000001, 00100000,
01010100, 01100101,
01110011, 01110100]

5. pad the binary message with zeros until its length is 512 mod 448

```
while (numString.length % 512 !== 448) {
    numString += '0';
  }
```

010000010010000001010100011001
1001010111001101110100 1

0100000100100000010101000110010101110011011101001000000000000000000-
00000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000-
0000000000000000000000000000000000000

6. take binary 8-bit ASCII code array from step 3, get its length in binary

```
const length = binary8bit.join('').length;
const binaryLength = utils.asciiToBinary(length);
```

Đếm độ dài của chuỗi ký tự ban đầu, rồi ghi nhận "độ dài" đó theo hệ nhị phân

48

110000

7. pad with zeros until it is 64 characters

```
const paddedBinLength = utils.padZero(binaryLength, 64);
numString += paddedBinLength;
```

Thêm số 0 vào trước "độ dài" dạng nhị phân đó cho đến khi đạt 64 bit (512 -448 = 64 bit)

00000000000000000000000-
00000000000000000000000-
000000000000000110000

SHA1 How it works?

5. pad the binary message with zeros until its length is 512 mod 448

```
while (numString.length % 512 !== 448) {
    numString += '0';
  }
```

010000010010000000101010001
10010101110011011101001

0100000100100000010101000110010101110011011101001000000000000000000-
0000000000000000000000000000000000000000000000000000000000000000-
0000000000000000000000000000000000000000000000000000000000000000-
0000000000000000000000000000000000000000000000000000000000000000-
0000000000000000000000000000000000000000000000000000000000000000-
0000000000000000000000000000000000000000000000000000000000000000-
0000000000000000000000000000000000000

512-448 = 64

7. pad with zeros until it is 64 characters

```
const paddedBinLength = utils.padZero(binaryLength, 64);
numString += paddedBinLength;
```

Thêm số 0 vào trước “độ dài” dạng nhị phân đó cho đến khi đạt 64 bit

0000000000000000000000-
0000000000000000000000-
000000000000000110000

8. append to your previously created binary message from step 5

0100000100100000010101000110010101110011011101001000000000000000000000000000000000000000000000000000-
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000-
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000-
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000-
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000000000000000110000

SHA1 How it works?

8. append to your previously created binary message from step 5

```
010000010010000001010100011001010111001101110100100000000000000000000000000000000000000-
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000-
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000-
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000-
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000000000110000
```

9. break the message into an array of 'chunks' of 512 characters

```
const chunks = utils.stringSplit(numString, 512);
```

```
[ 01000001001000000101010001100101011100110111010010000000000000000-
00000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000-
00000000000000000000000000000000000000000000000000000000000000000-
000000000000000000000000000000000000000000000000000110000 ]
```

10. break each chunk into a subarray of sixteen 32-bit 'words'

512/32 = 16 “word”

```
const chunkWords = chunks
            .map((chunk) =>
            utils.stringSplit(chunk, 32));
```

```
[ [ 01000001001000000101010001100101, 01110011011101001000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000110000 ] ]
```

SHA1 How it works?

11. loop through each 'chunk' array of sixteen 32-bit 'words' and extend each array to 80 'words' using bitwise operations

Hiện tại 16 “word” => thành 80 “word”

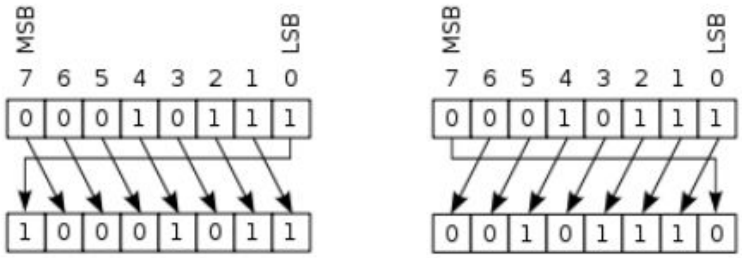
```
    //chunkWords is an array of 'chunk' subarrays
const words80 = chunkWords.map((chunk) => {
//we start with a 'chunk' array of 16 32-bit 'words'
    for (let i = 16; i <= 79; i++) {
      //take four words from that chunk using
      //your current i in the loop
      const wordA = chunk[i - 3];
      const wordB = chunk[i - 8];
      const wordC = chunk[i - 14];
      const wordD = chunk[i - 16];

      //perform consecutive XOR bitwise
      //operations going through each word
      const xorA = utils.xOR(wordA, wordB);
      const xorB = utils.xOR(xorA, wordC);
      const xorC = utils.xOR(xorB, wordD);

      //left rotate by one
      const newWord = utils.leftRotate(xorC, 1);
      //append to the array and continue the loop
      chunk.push(newWord);
    }
    return array;
  });
```

```
[[01000001001000000101010001100101, 01110011011101001000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000000000,
00000000000000000000000000000000, 00000000000000000000000000110000]]
```

SHA1 How it works?



Right Rotate Left Rotate

DEFINITION

Circular Shift Operation

Now, the circular shift operation $S^n(X)$ on the word X by n bits, n being an integer between 0 and 32, is defined by

$$S^n(X) = (X << n) \quad \textbf{OR} \quad (X >> 32 - n),$$

where $X \ << n$ is the **left-shift** operation, obtained by discarding the leftmost n bits of X and padding the result with n zeroes on the right.

$X \ >> 32 - n$ is the **right-shift** operation obtained by discarding the rightmost n bits of X and padding the result with n zeroes on the left. Thus $S^n(X)$ is equivalent to a circular shift of X by n positions, and in this case the circular left-shift is used. [3]

DEFINITION

Functions used in the algorithm

A sequence of logical functions are used in SHA-1, depending on the value of i, where $0 \leq i \leq 79$, and on three 32-bit words B, C, and D, in order to produce a 32-bit output. The following equations describe the logical functions, where $\neg$ is the logical *NOT*, $\vee$ is the logical *OR*, $\wedge$ is the logical *AND*, and $\oplus$ is the logical XOR:

$$f(i; B, C, D) = (B \wedge C) \vee ((\neg B) \wedge D) \qquad \text{for } 0 \geq i \geq 19$$

$$f(i; B, C, D) = B \oplus C \oplus D \qquad \text{for } 20 \geq i \geq 39$$

$$f(i; B, C, D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) \qquad \text{for } 40 \geq i \geq 59$$

$$f(i; B, C, D) = B \oplus C \oplus D \qquad \text{for } 60 \geq i \geq 79.$$

SHA1 How it works?

12. initialize some variables

```
let h0 = '01100111010001010010001100000001';
let h1 = '11101111110011011010101110001001';
let h2 = '10011000101110101101110011111110';
let h3 = '00010000001100100101010001110110';
let h4 = '11000011110100101110000111110000';
let a = h0;
let b = h1;
let c = h2;
let d = h3;
let e = h4;
```

13. looping through each chunk: bitwise operations and variable reassignment

```
for (let i = 0; i < words80.length; i++) {
  for (let j = 0; j < 80; j++) {
    let f;
    let k;
    if (j < 20) {
      const BandC = utils.and(b, c);
      const notB = utils.and(utils.not(b), d);
      f = utils.or(BandC, notB);
      k = '01011010100000100111100110011001';
    } else if (j < 40) {
      const BxorC = utils.xOR(b, c);
      f = utils.xOR(BxorC, d);
      k = '01101110110110011110101110100001';
    } else if (j < 60) {
      const BandC = utils.and(b, c);
      const BandD = utils.and(b, d);
      const CandD = utils.and(c, d);
      const BandCorBandD = utils.or(BandC, BandD);
      f = utils.or(BandCorBandD, CandD);
      k = '10001111000110111011110011011100';
    } else {
      const BxorC = utils.xOR(b, c);
      f = utils.xOR(BxorC, d);
      k = '11001010011000101100000111010110';
    }
    const word = words80[i][j];
    const tempA = utils.binaryAddition(utils.leftRotate(a, 5), f);
    const tempB = utils.binaryAddition(tempA, e);
    const tempC = utils.binaryAddition(tempB, k);
    let temp = utils.binaryAddition(tempC, word);

    temp = utils.truncate(temp, 32);
    e = d;
    d = c;
    c = utils.leftRotate(b, 30);
    b = a;
    a = temp;
  }
  h0 = utils.truncate(utils.binaryAddition(h0, a), 32);
  h1 = utils.truncate(utils.binaryAddition(h1, b), 32);
  h2 = utils.truncate(utils.binaryAddition(h2, c), 32);
  h3 = utils.truncate(utils.binaryAddition(h3, d), 32);
  h4 = utils.truncate(utils.binaryAddition(h4, e), 32);
}
```

h0,h1,h2,h3,h4: là các ký tự mong muốn cuối cùng

h0 = 01100111010001010010001100000001
h1 = 11101111110011011010101110001001
h2 = 10011000101110101101110011111110
h3 = 00010000001100100101010001110110
h4 = 11000011110100101110000111110000

h0 = 100011110000110000000100001010101
h1 = 10010001010101100011001111100100
h2 = 101001111101111000001100101000110
h3 = 100010110011100000111010011001000
h4 = 10010000000111011111000000100011

SHA1 How it works?

1. Take input text and split it into an array of the characters' ASCII codes

'A Test'

[A, , T, e, s, t]

[65, 32, 84, 101, 115, 116]

14. convert each of the five resulting variables to hexadecimal
15. join them together and return it!

```
return [h0, h1, h2, h3, h4]
       .map((string) => utils.binaryToHex(string))
       .join('');
```

h0 = 10001111000011000000100001010101
h1 = 10010001010101100011001111100100
h2 = 10100111110111100001100101000110
h3 = 10001011001110000111010011001000
h4 = 10010000000111011111000001000011

h0 = 8f0c0855
h1 = 915633e4
h2 = a7de1946
h3 = 8b3874c8
h4 = 901df043

Your hash value! 8f0c0855915633e4a7de19468b3874c8901df043

Reference: