MD5

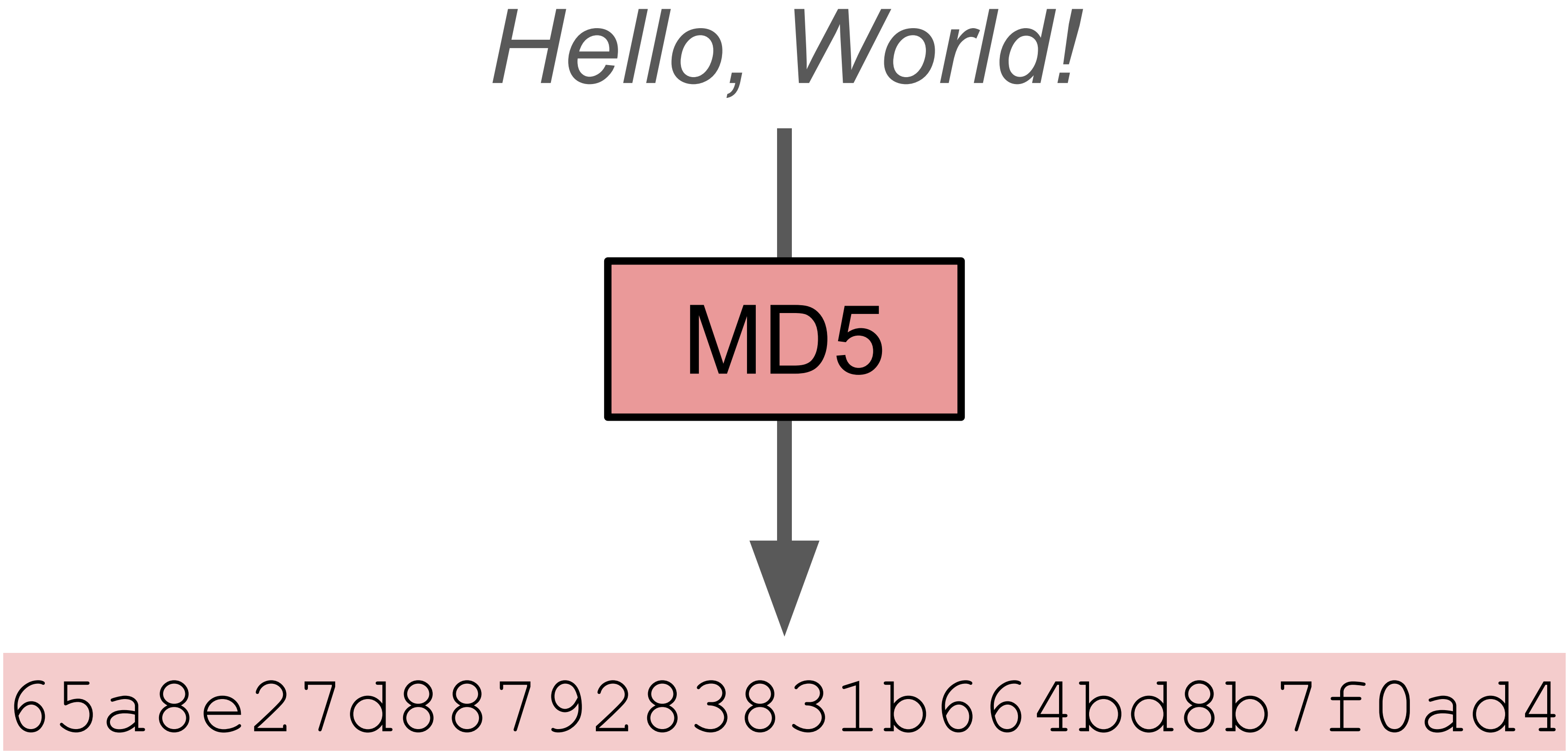
Raul Rodrigues, Josué Nascimento, João Nunes,
Pedro Pacheco, Victor Huander

What is a hash function

Merriam-Webster:
hash verb

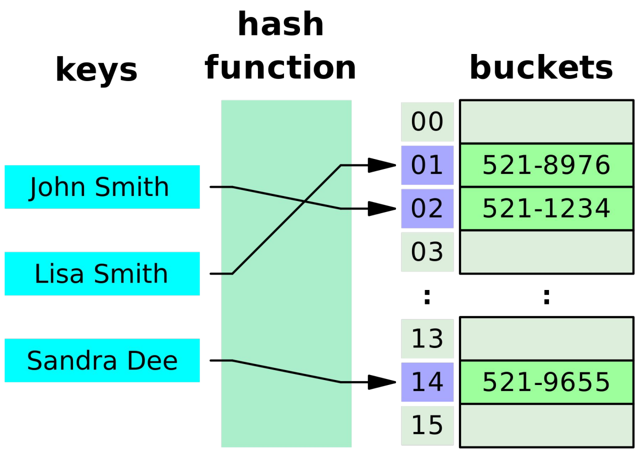
1. : to chop (food, such as meat and potatoes) into small pieces
2. : to confuse, to muddle

Wikipedia, Hashish, Public domain


Hello, World!
MD5
65a8e27d8879283831b664bd8b7f0ad4

Where hash functions are used

1. Abstract data structures
   a. Hash sets
   b. Hash maps/tables
2. Cryptography



Wikipedia, Hash table, Public domain

Hash tables

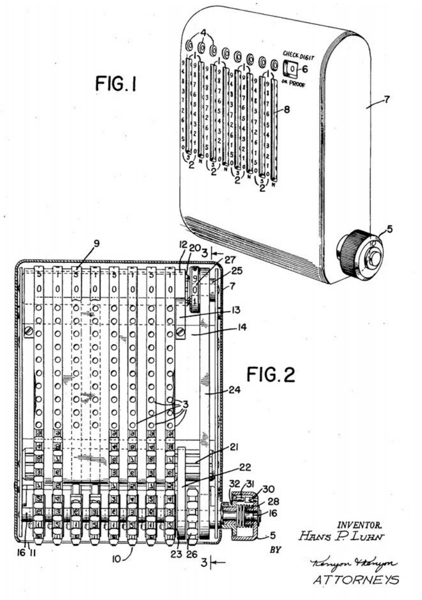
search	O(1)
insertion	O(1)
deletion	O(1)

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

https://www.bigocheatsheet.com/

History

Computer for verifying numbers



Hans Peter Luhn, 1958

Photo: IBM

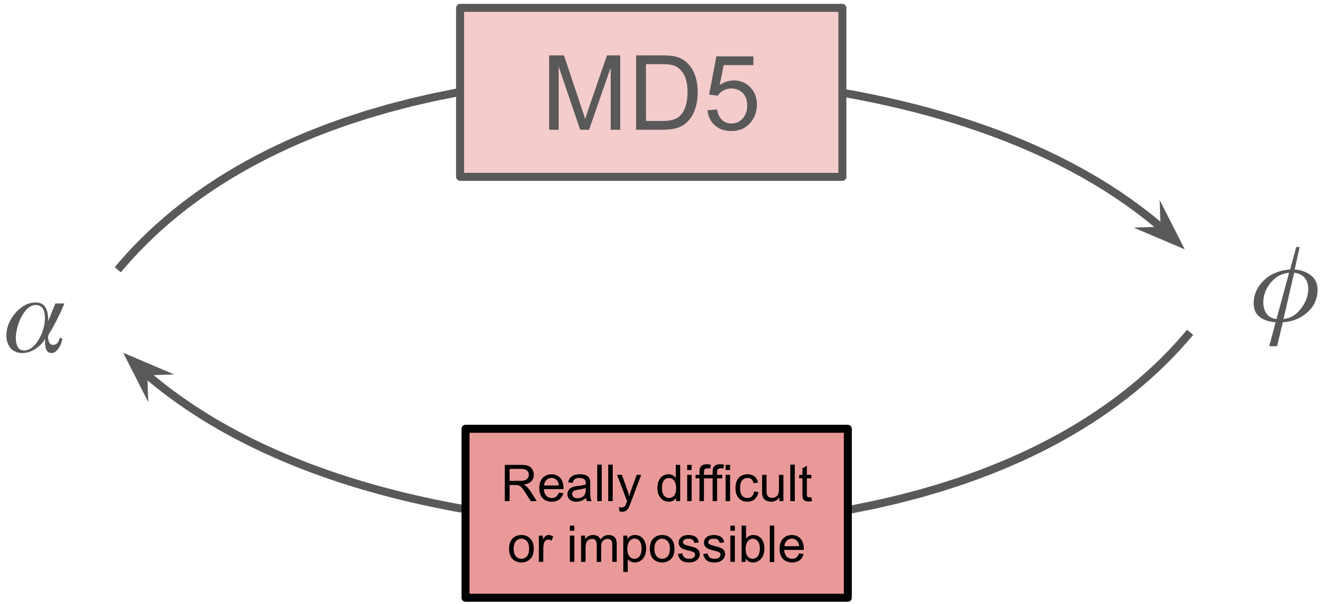
Hash functions properties

1. **Preimage Resistant**: it should be hard to find a message with a given hash value.

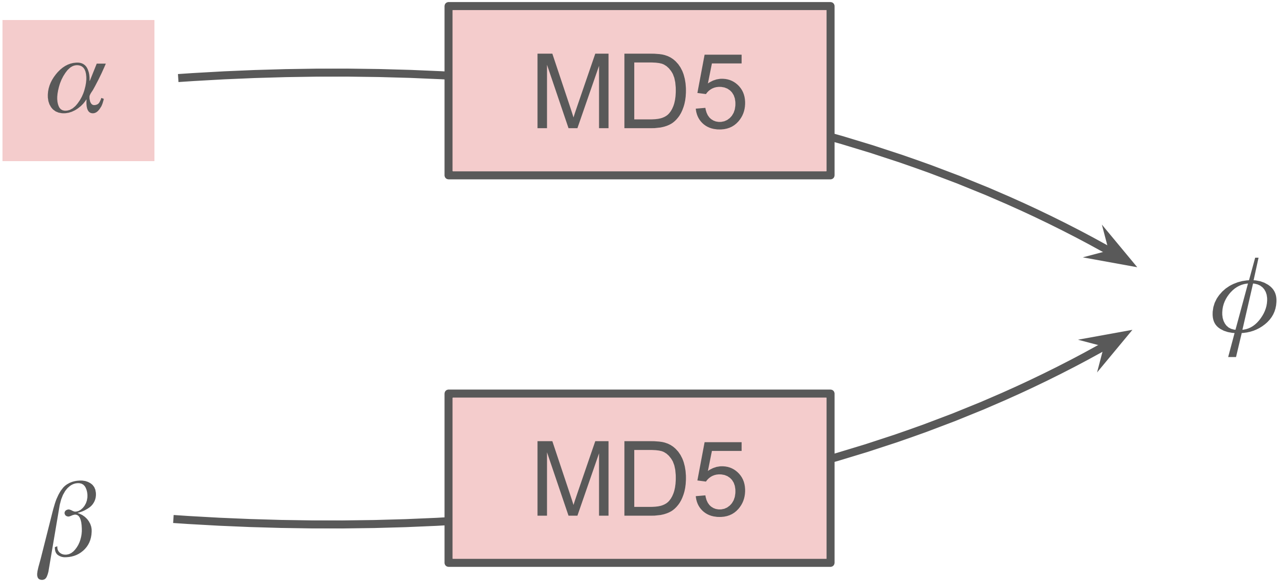
2. **Second Preimage Resistant**: given one message it should be hard to find another message with the same hash value.

3. **Collision Resistant**: it should be hard to find two messages with the same hash value.

Preimage resistant

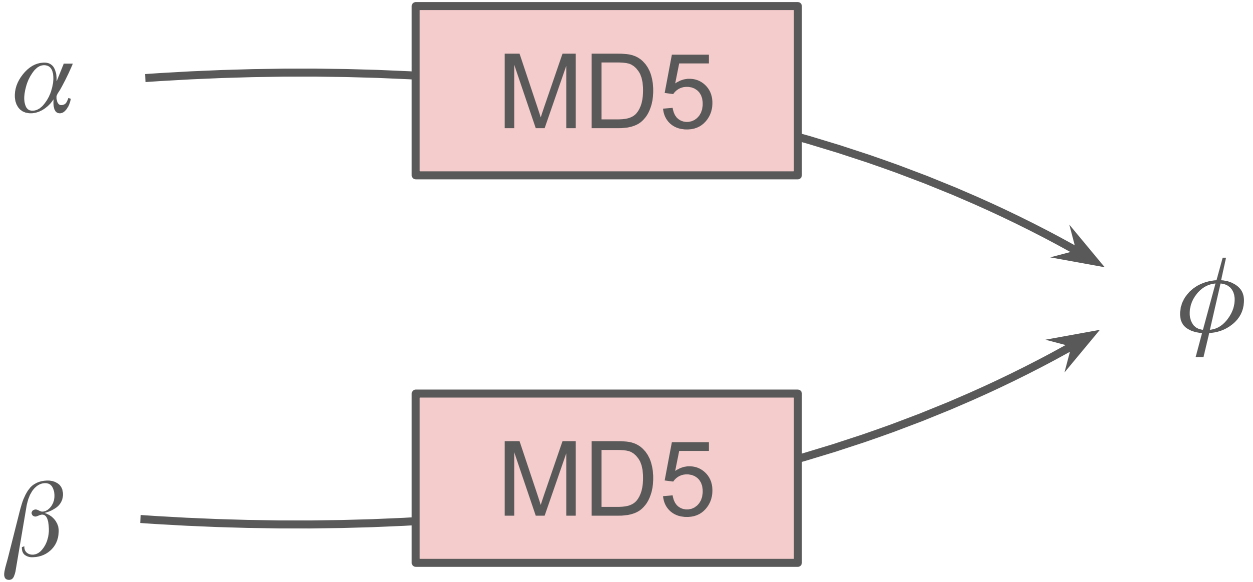


Second preimage resistant



given m it should be hard to find an m’ ≠ m with H(m) = H(m’)

Collision resistant



x ≠ x' ,   H(x) = H(x')

- ***Preimage Resistant****: it should be hard to find a message with a given hash value.*

- ***Second Preimage Resistant****: given one message it should be hard to find another message with the same hash value.*

- ***Collision Resistant****: it should be hard to find two messages with the same hash value.*

MD5

MD5 message-digest algorithm

- Professor Ronald Rivest
- 1992
- MD4 family

Message

1010 0000 1111

- Arbitrary size
- Fixed size output (128 bits)

4 registers / buffers

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

`digest = A + B + C + D`

Sine derived function / table

```
K(i) := floor(232 × abs (sin(i + 1)))
```

Pre computed table

```
K[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee
}
K[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501
}
K[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be
}
K[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821
}
K[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa
}
K[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8
}
K[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed
}
K[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a
}
K[32..35] := { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c
}
```

Shift amounts table

```
s[ 0..15] := { 7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22 }
s[16..31] := { 5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20 }
s[32..47] := { 4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23 }
s[48..63] := { 6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21 }
```

Padding

1. 101000001111
2. 1010000011111
3. 10100000111110000000000000000000000000000000000000000000000000000000000000000000000000
   000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   0000000000000
4. 10100000111110000000000000000000000000000000000000000000000000000000000000000000000000
   000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
   00000000000000000000000000000000000011100000000000000000000000000000000000000

Message size is now a multiple of 512

Original message plus ‘1’

```
10100000111111
```

Add n number of trailing zeros

Between 0 and 511 ‘zero’ bits are add the the end of the message following this approach:

the message size at the end of the padding needs to be congruent to 448, modulo 512

`10100000011111000000000000000...000000000`

Add size of the message

Now add the size of the original message in 64 bit format:

```
1.    000000000000000000000000000000000000000000000000000000000000 1100
2.    00000000000000000000000000000000 00000000000000000000000000001100
3.    00000000000000000000000000001100 00000000000000000000000000000000
4.    0000000000000000000000000000110000000000000000000000000000000000
```

```
1010000011111000000000000000...00000000000000000000000
0000000000000011000000000000000000000000000000000000
```

message


512 bits
512 bits
512 bits
...

512 bits

32 bit word	32 bit word	32 bit word	32 bit word
32 bit word	32 bit word	32 bit word	32 bit word
32 bit word	32 bit word	32 bit word	32 bit word
32 bit word	32 bit word	32 bit word	32 bit word

32 bit word

For each 32 bit word do

```
// Main loop:
for i from 0 to 63 do
       var int F, g

       if 0 ≤ i ≤ 15 then
              F := (B and C) or ((not B) and D)
              g := (i) mod 16
       else if 16 ≤ i ≤ 31 then
              F := (D and B) or ((not D) and C)
              g := (5×i + 1) mod 16
       else if 32 ≤ i ≤ 47 then
              F := B xor C xor D
              g := (3×i + 5) mod 16
       else if 48 ≤ i ≤ 63 then
              F := C xor (B or (not D))
              g := (7×i) mod 16

       F := F + A + K[i] + M[g]
       A := D
       D := C
       C := B
       B := B + leftrotate(F, s[i])
end for
```

For each 32 bit word do

```
// Main loop:
for i from 0 to 63 do
        var int F, g

        if 0 ≤ i ≤ 15 then
                F := (B and C) or ((not B) and D)
                g := (i) mod 16
        else if 16 ≤ i ≤ 31 then
                F := (D and B) or ((not D) and C)
                g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47 then
                F := B xor C xor D
                g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63 then
                F := C xor (B or (not D))
                g := (7×i) mod 16

        F := F + A + K[i] + M[g]
        A := D
        D := C
        C := B
        B := B + leftrotate(F, s[i])
end for
```

Sine table = `k[i]`

```
K(i) := floor(232 × abs (sin(i + 1)))
```

Pre computed table

```
K[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee
}
K[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501
}
K[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be
}
K[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821
}
K[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa
}
K[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8
}
K[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed
}
K[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a
}
K[32..35] := { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c
}
```

Shift amounts table = `s[i]`

```
s[ 0..15] := { 7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22 }
s[16..31] := { 5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20 }
s[32..47] := { 4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23 }
s[48..63] := { 6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21 }
```

F, g initialization

```
F := 0x00000000
g := 0
i := 13
```

```
// Main loop:
for i from 0 to 63 do
        var int F, g

        if 0 ≤ i ≤ 15 then
                F := (B and C) or ((not B) and D)
                g := (i) mod 16
        else if 16 ≤ i ≤ 31 then
                F := (D and B) or ((not D) and C)
                g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47 then
                F := B xor C xor D
                g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63 then
                F := C xor (B or (not D))
                g := (7×i) mod 16

        F := F + A + K[i] +  M[g]
        A := D
        D := C
        C := B
        B := B + leftrotate(F, s[i])
end for
```

Main bitwise operations block

i := 13

```
// Main loop:
for i from 0 to 63 do
        var int F, g

        if 0 ≤ i ≤ 15 then
                F := (B and C) or ((not B) and D)
                g := (i) mod 16
        else if 16 ≤ i ≤ 31 then
                F := (D and B) or ((not D) and C)
                g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47 then
                F := B xor C xor D
                g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63 then
                F := C xor (B or (not D))
                g := (7×i) mod 16

        F := F + A + K[i] +  M[g]
        A := D
        D := C
        C := B
        B := B + leftrotate(F, s[i])
end for
```

(B and C) or ((not B) and D)

```
A := 0x18AF03FF
B := 0x24893AF1
C := 0x556FBCCC
D := 0x916309F7
```

1. (0x24893AF1 and 0x556FBCCC) or ((not 0x24893AF1) and 0x916309F7)
2. (0x24893AF1 and 0x556FBCCC) or (0xDB76C50E and 0x916309F7)
3. 0x40938C0 or (0xDB76C50E and 0x916309F7)
4. 0x40938C0 or 0x6E9DFEFA
5. 0x6E9DFEFA

0x6E9DFEFA

```
// Main loop:
for i from 0 to 63 do
        var int F, g

        if 0 ≤ i ≤ 15 then
                F := (B and C) or ((not B) and D)
                g := (i) mod 16
        else if 16 ≤ i ≤ 31 then
                F := (D and B) or ((not D) and C)
                g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47 then
                F := B xor C xor D
                g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63 then
                F := C xor (B or (not D))
                g := (7×i) mod 16

        F := F + A + K[i] +  M[g]
        A := D
        D := C
        C := B
        B := B + leftrotate(F, s[i])
end for
```

g set

g will be used to select the word to be selected for the main sum part


```
// Main loop:
for i from 0 to 63 do
          var int F, g

          if 0 ≤ i ≤ 15 then
                    F := (B and C) or ((not B) and D)
                    g := (i) mod 16
          else if 16 ≤ i ≤ 31 then
                    F := (D and B) or ((not D) and C)
                    g := (5×i + 1) mod 16
          else if 32 ≤ i ≤ 47 then
                    F := B xor C xor D
                    g := (3×i + 5) mod 16
          else if 48 ≤ i ≤ 63 then
                    F := C xor (B or (not D))
                    g := (7×i) mod 16

          F := F + A + K[i] + M[g]
          A := D
          D := C
          C := B
          B := B + leftrotate(F, s[i])
end for
```

F + A + K[i] + M[g]

```
word := 0x658FBCAC
F    := 0x6E9DFEFA
A    := 0x18AF03FF
K[i] := 0xF6BB4B60
```

0x6E9DFEFA + 0x18AF03FF + 0xF6BB4B60 +0x658FBCAC

0x6E9DFEFA

```
// Main loop:
for i from 0 to 63 do
        var int F, g

        if 0 ≤ i ≤ 15 then
                F := (B and C) or ((not B) and D)
                g := (i) mod 16
        else if 16 ≤ i ≤ 31 then
                F := (D and B) or ((not D) and C)
                g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47 then
                F := B xor C xor D
                g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63 then
                F := C xor (B or (not D))
                g := (7×i) mod 16

        F := F + A + K[i] +  M[g]
        A := D
        D := C
        C := B
        B := B + leftrotate(F, s[i])
end for
```

register swapping

```
A := C
D := C
C := B
```

```
// Main loop:
for i from 0 to 63 do
        var int F, g

        if 0 ≤ i ≤ 15 then
                F := (B and C) or ((not B) and D)
                g := (i) mod 16
        else if 16 ≤ i ≤ 31 then
                F := (D and B) or ((not D) and C)
                g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47 then
                F := B xor C xor D
                g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63 then
                F := C xor (B or (not D))
                g := (7×i) mod 16

        F := F + A + K[i] +  M[g]
        A := D
        D := C
        C := B
        B := B + leftrotate(F, s[i])
end for
```

leftrotate(F, s[i])

```
B    := 0x24893AF1
F    := 0x6E9DFEFA
S[i] := 12
```

```
leftrotate (word, sa):
    return (word << sa) or (word >> (32 - sa))
```

1. 0x24893AF1 + (0x6E9DFEFA << 12) or (0x6E9DFEFA >> 20)
2. 0x24893AF1 + 0xDFEFA000 or 0xEFA00000
3. 0x24893AF1 + 0x304FA000
4. 0x54D8DAF1

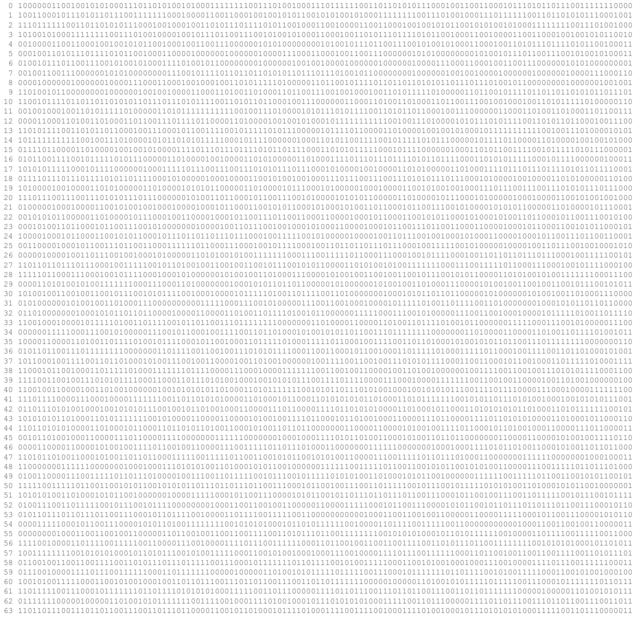
0x54D8DAF1

```
// Main loop:
for i from 0 to 63 do
        var int F, g

        if 0 ≤ i ≤ 15 then
                F := (B and C) or ((not B) and D)
                g := (i) mod 16
        else if 16 ≤ i ≤ 31 then
                F := (D and B) or ((not D) and C)
                g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47 then
                F := B xor C xor D
                g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63 then
                F := C xor (B or (not D))
                g := (7×i) mod 16

        F := F + A + K[i] +  M[g]
        A := D
        D := C
        C := B
        B := B + leftrotate(F, s[i])
end for
```

One full pass through the main loop (current digest)



.
.
.
```
26 0000010000100110111001001000101000001101010010100111111100011100111110110001110001001011110001001101101101110111000100111110 0101
27 1101101101110111000100111110010110100100110010011001011100101011000011010100101001111111000111001111101100011100010010111100 0100
28 1111101100011100010010111100010001010000001010010011010001110000101001001100100110010111001010110000110101001010011111110001 1100
29 0000110101001010011111110001110001101000000010001010110110110000010100000010100100110100011100001010010011001001100101110010 1011
30 1010010011001001100101110010101111001000100001011111010011011110011010000000100010101101101100000101000000101001001101000111 0000
31 0101000000101001001101000111000000000011111000111001010000011100110010001000010111111010011011110011010000000100010101101101 10000
32 0110100000000100010101101101100001000011000011010011011110100101100000011111000111001010000011100110010001000010111111010011 011110
33 1100100010000101111101001101111001011011001110111111111000000011010000110000110100110111101001011000000111110001110010100000 11100
34 0000011111000111001010000011100101100010011110011011010001010010101101100111011111111100000001101000011000011010011011110100 1011
35 1000011000011010011011110100101111000101100100011011111010001111101100010011110011011010001010010101101100111011111111100000 00110
```
.
.
.

26 00000100001001101110010010001010000011010100101001111111000111001111101100011100010010111100010011011011011101110001001111100101
27 11011011011101110001001111100101101001001100100110010111001010110000110101001010011111110001110011111011000111000100101111000100

Recent history

Why is MD5 no longer used widely in the industry?

Is it still used today?

Where did it fail?

- ***Preimage Resistant***: *it should be hard to find a message with a given hash value.*

- ***Second Preimage Resistant***: *given one message it should be hard to find another message with the same hash value.*

- ***Collision Resistant***: *it should be hard to find two messages with the same hash value.*

- In 1996, Dobbertin announced a collision of the compression function of MD5 (Dobbertin, 1996).

- Birthday attack
- On 17 August 2004, when collisions for the full MD5, Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu.
- On 1 March 2005, Arjen Lenstra, Xiaoyun Wang, and Benne de Weger demonstrated construction of two X.509 certificates with same hash

Dangers today

- As of 2019, one quarter of widely used content management systems were reported to still use MD5 for password hashing.[1]

- These hash and collision attacks have been demonstrated in the public in various situations, including colliding document files[2][3] and digital certificates.[4]

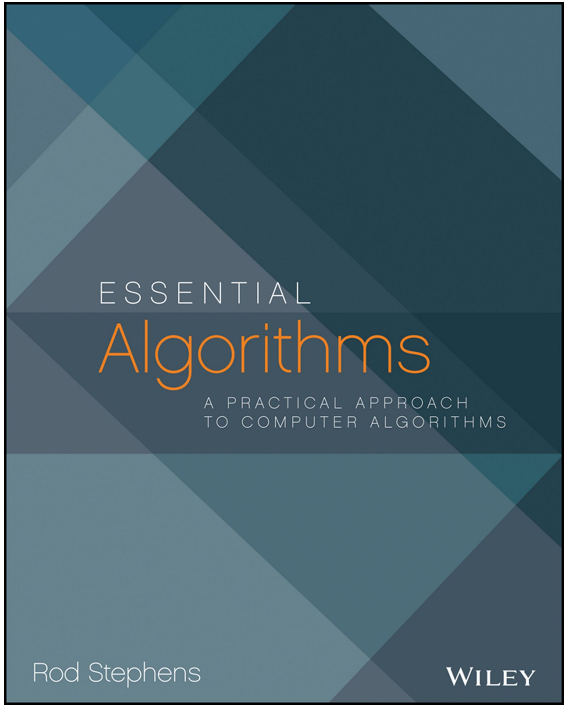
1. Cimpanu, Catalin. "A quarter of major CMSs use outdated MD5 as the default password hashing scheme". ZDNet;
2. Magnus Daum, Stefan Lucks. "Hash Collisions (The Poisoned Message Attack)". Eurocrypt 2005 rump session;
3. Max Gebhardt; Georg Illies; Werner Schindler (4 January 2017). "A Note on the Practical Value of Single Hash Collisions for Special File Formats";
4. Sotirov, Alexander; Marc Stevens; Jacob Appelbaum; Arjen Lenstra; David Molnar; Dag Arne Osvik; Benne de Weger (30 December 2008). "MD5 considered harmful today".

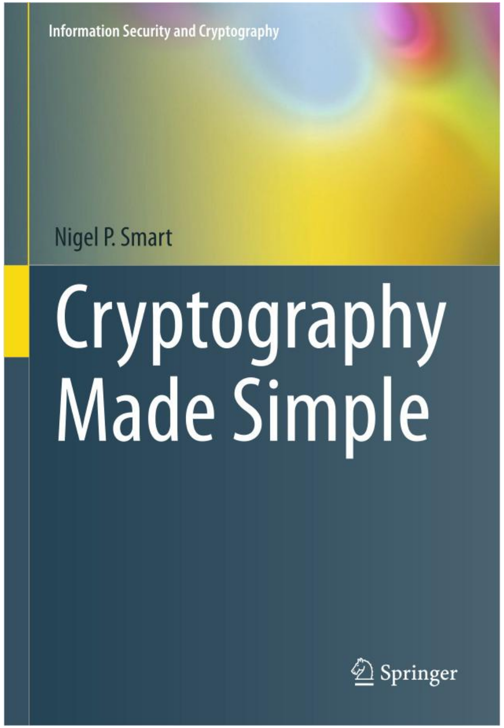
https://haveibeenpwned.com/

Hurb: In approximately March 2019, the online Brazilian travel agency Hurb (formerly Hotel Urbano) suffered a data breach. The data subsequently appeared online for download the following year and included over 20 million customer records with email and IP addresses, names, dates of birth, phone numbers and passwords stored as unsalted MD5 hashes. The data was provided to HIBP by dehashed.com.

Compromised data: Dates of birth, Email addresses, IP addresses, Names, Passwords, Phone numbers, Social media profiles

Stephens, Rod. Essential algorithms : a practical approach to computer algorithms. Indianapolis, IN: Wiley, 2013





Smart, Nigel P. Cryptography made simple. Cham, Switzerland: Springer, 2016

The MD5 Message-Digest Algorithm, MIT Laboratory for Computer Science and RSA Data Security, April 1992

https://datatracker.ietf.org/doc/html/rfc1321

MD5

7f138a09169b250e9dcb378140907378

Raul Rodrigues, Josué Nascimento, João Nunes,
Pedro Pacheco, Victor Huander