Chapter 1

Algoritmos de Hashing

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

Abstract

Este artigo ira descrever o funcionamento de funções de hashing e suas diversas aplicações dando foco à função de hash chamada "MD5 message-digest algorithm", conhecida somente como MD5. Será mostrada uma implementação e provas das propriedades de uma função de hash. Em seguida sera exposto os problemas que a função MD5 tem no contexto de segurança atualmente que efetivamente inviabilizam sua utilização na criptografia e segurança da informação.

1.1. O que é Hashing

Hashing refere-se ao processo de geração de uma saída (output) de tamanho fixo a partir de uma entrada (input) de tamanho variável. Isto é feito através do uso de fórmulas matemáticas conhecidas como funções hash (implementadas como algoritmos de hashing) [Cryptography Made Simple, 2016]. Estas fórmulas irão gerar uma saída, a qual é muito diferente do valor de entrada, tornando impossível reconhecer uma a partir da outra.

1.2. Por que Hashing

Hashing é uma técnica amplamente utilizada na computação em diversas aplicações, como por exemplo verificação e proteção de senhas, operações de compilação, na verificação da veracidade de arquivos, entre outros, e por isso é um tópico de muita atenção e constante evolução por parte da comunidade [Stephens, Rod. Essential algorithms]. Porém dependendo de qual operação se quer realizar, a metodologia de hashing irá se modificar, junto com algumas características específicas de cada tipo de função.

1.3. Casos de uso

Este processo é muito importante para a encriptação de senhas, pois caso algum invasor tenha contato com as senhas encriptadas, elas serão inúteis para o invasor. Casos de uso muito importantes incluem: a verificação de integridade de arquivos, pois cada arquivo

deverá possuir apenas um hash, que funciona como a identidade do arquivo, podendo assim provar que o arquivo recebido é o esperado; estrutura abstrata de dados conhecida como hash table que permite inserir e acessar dados em tempo O(1)[Stephens, Rod. Essential algorithms].

1.4. MD5

A MD5 será a função de hashing que será abordada no nosso estudo. A MD5 produz um valor de hash de 128 bits expresso em 32 caracteres hexadecimais dada uma entrada [The MD5 Message-Digest Algorithm, 1992]. Embora ela tenha sido projetada inicialmente para ser usada como uma função hash criptográfica, foi constatado que ela sofre de extensas vulnerabilidades. Ela pode, também, ser usada como uma soma de verificação para checar a integridade de dados de arquivos. Além das aplicações acima, ela é também adequada para outros fins não criptográficos, por exemplo, para determinar a partição para uma chave específica em um banco de dados particionado .

O algoritmo MD5 foi formalmente descrito em Abril de 1992 por Ronald L. Rivest. Criado para ser o sucessor do algoritmo de hashing MD4 para ser computacionalmente mais segura contra colisões. O algoritmo de MD5 aceita como entrada uma mensagem de tamanho arbitrário [The MD5 Message-Digest Algorithm, 1992]. Não é necessário que o tamanho da mensagem seja um múltiplo de 8 bits. O algoritmo de MD5 pode ser separado em 5 partes.

1.4.1. Passo 1: acrescentar bits de preenchimento

A mensagem é inicialmente "completada" para que seu tamanho seja igual ao congruente de 448 módulo 512. Ou seja que se for feito o módulo 512 da mensagem com os de preenchimento o resultado deve ser 448. O os bit de preenchimento são criados da seguinte forma. Um de "1" é adicionado ao final da mensagem e bits "0" são adicionados depois até que a mensagem esteja do tamanho correto.

1.4.2. Passo 2: acrescentar o tamanho da mensagem original

Uma representação em 64 bits do tamanho da mensagem original, sem os bits de preenchimento, é adicionada ao final da mensagem depois dos bits de preenchimento. No caso que a mensagem original não possa ser representada por 64 bits, apenas os bits de menor significância são adicionados.

Agora a mensagem a ser processada tem um tamanho divisível por 512.

1.4.3. Passo 3: inicialização do buffer MD

Um buffer de quatro palavras (32 bits cada palavra) será usado para computar o digest (hash) da mensagem. Normalmente esse buffer é representado por quatro inteiros de 32 bits. O buffer tem o valor inicial como descrito:

```
A: 01 23 45 67
B: 89 AB CD EF
C: FE DC BA 98
```

```
D: 76 54 32 10
```

1.4.4. Passo 4: processamento da mensagem em blocos de 16 palavras

Inicialmente são definidas 4 funções auxiliares. Cada uma tem como parâmetros 3 palavras de 32 bits e como resultado uma palavra de 32 bits.

```
F(X,Y,Z) = (X and Y) or (not(X) and Z)

G(X,Y,Z) = (X and Z) or (Y and not(Z))

H(X,Y,Z) = (X xor Y) xor Z

I(X,Y,Z) = Y xor (X or not(Z))
```

As quatro funções F, G, H e I funcionam de maneira "bitwise paralelo"

Essa etapa utiliza uma tabela T[1 ... 64] de 64 elementos construída a partir da função seno. Seja T[i] o elemento de posição i da tabela, equivalente à parte inteira de 4294967296 multiplicado pelo módulo do seno de i, onde i é representado em radianos.

Este pseudo código foi dado no memo de Rivest original, na seção `3.4 Step 4. Process Message in 16-Word Blocks`. Aqui se encontra em verbatim:

Algoritmo 1. Pseudo código de Rivest

```
/* Process each 16-word block. */
   For i = 0 to N/16-1 do

     /* Copy block i into X. */
     For j = 0 to 15 do
       Set X[j] to M[i*16+j].
     end /* of loop on j */

     /* Save A as AA, B as BB, C as CC, and D as DD. */
     AA = A
     BB = B
     CC = C
     DD = D

     /* Round 1. */
     /* Let [abcd k s i] denote the operation
          a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
     /* Do the following 16 operations. */
     [ABCD  0  7  1]  [DABC  1 12  2]  [CDAB  2 17  3]  [BCDA  3 22  4]
     [ABCD  4  7  5]  [DABC  5 12  6]  [CDAB  6 17  7]  [BCDA  7 22  8]
     [ABCD  8  7  9]  [DABC  9 12 10]  [CDAB 10 17 11]  [BCDA 11 22 12]
     [ABCD 12  7 13]  [DABC 13 12 14]  [CDAB 14 17 15]  [BCDA 15 22 16]

     /* Round 2. */
     /* Let [abcd k s i] denote the operation
          a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */
     /* Do the following 16 operations. */
```

```
28      [ABCD  1  5 17]  [DABC  6  9 18]  [CDAB 11 14 19]  [BCDA  0 20 20]
29      [ABCD  5  5 21]  [DABC 10  9 22]  [CDAB 15 14 23]  [BCDA  4 20 24]
30      [ABCD  9  5 25]  [DABC 14  9 26]  [CDAB  3 14 27]  [BCDA  8 20 28]
31      [ABCD 13  5 29]  [DABC  2  9 30]  [CDAB  7 14 31]  [BCDA 12 20 32]
32
33      /* Round 3. */
34      /* Let [abcd k s t] denote the operation
35           a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
36      /* Do the following 16 operations. */
37      [ABCD  5  4 33]  [DABC  8 11 34]  [CDAB 11 16 35]  [BCDA 14 23 36]
38      [ABCD  1  4 37]  [DABC  4 11 38]  [CDAB  7 16 39]  [BCDA 10 23 40]
39      [ABCD 13  4 41]  [DABC  0 11 42]  [CDAB  3 16 43]  [BCDA  6 23 44]
40      [ABCD  9  4 45]  [DABC 12 11 46]  [CDAB 15 16 47]  [BCDA  2 23 48]
41
42      /* Round 4. */
43      /* Let [abcd k s t] denote the operation
44           a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
45      /* Do the following 16 operations. */
46      [ABCD  0  6 49]  [DABC  7 10 50]  [CDAB 14 15 51]  [BCDA  5 21 52]
47      [ABCD 12  6 53]  [DABC  3 10 54]  [CDAB 10 15 55]  [BCDA  1 21 56]
48      [ABCD  8  6 57]  [DABC 15 10 58]  [CDAB  6 15 59]  [BCDA 13 21 60]
49      [ABCD  4  6 61]  [DABC 11 10 62]  [CDAB  2 15 63]  [BCDA  9 21 64]
50
51      /* Then perform the following additions. (That is increment each
52         of the four registers by the value it had before this block
53         was started.) */
54      A = A + AA
55      B = B + BB
56      C = C + CC
57      D = D + DD
58
59    end /* of loop on i */
```

1.4.5. Passo 5: resultado

O digest (hash) da mensagem é a concatenação de A, B, C e D nesta ordem. Uma implementação em Typescript será apresentada em seguida.

1.5. Implementação

Nesta seção será descrito o processo de implementação do algoritmo MD5 que o grupo realizou com base na descrição original de Rivest. Será apresentado todos os artefatos e métodos auxiliares para a construção do algoritmo juntamente com exemplos de cada um.

1.5.1. Typescript

Typescript é uma linguagem de programação criada e mantida pela Microsoft. É descrita como um superset estritamente sintático da linguagem Javascript que adiciona tipagem estática. Sendo um desenvolvimento em cima de Javascript, Typescript usa intepretadores de Javascript depois de ser "transcompilado" para Javascript.

1.5.2. Definição

Nossa implementação foi feita pensando estritamente em um implementação acadêmica. Para o calculo do digest os dados são interpretados como strings de caracteres ’1’ e ’0’. Essa não é um solução viável para uma construção de ferramenta porém é suficiente para o estudo das propriedades do algoritmo.

1.5.3. Implementação

A implementação inicia com a criação de uma classe chamada de MD5, esta classe contém 8 variáveis, sendo elas: um "array(shitfAmounts)" com números decimais, um "array(premadeSineTable)" com números hexadecimais, 4 "strings" de números binários, a mensagem original que irá ser codificada e a mensagem codificada resultante, ambas inicializando como strings vazias. Após rodar a função de execução, chamada de "run", a mesma irá receber por parâmetro uma "string" de números binários, e passará a mesma para as variáveis "message" e "originalMessage", então será concatenado um bit 1 no final da mensagem, para depois fazer o congruente de 448 com 512. Com o congruente, é concatenado mais uma vez o mesmo com o tamanho da mensagem original, em 64 bits, e a mensagem resultante é quebrada em blocos de 512 bits. Logo, é feito um "loop" em cima do "array" formado pelos blocos de 512 bits, existindo dentro deste contexto 4 variáveis, A, que receberá o valor da variável a0, B, que irá receber o valor de b0, C, recebendo o valor de c0 e D, recebendo o valor de d0. Dentro do contexto desse loop, cada bloco de 512 será quebrado em blocos de 32 "bits", para então ser realizado um segundo loop 32 vezes, tendo como contador a variável "j". Dentro do "loop" de 32, será realizado um segundo "loop" 64 vezes no qual o código está representado abaixo:

Algoritmo 2. Bloco central de nossa implementação

```
1 for(let k = 0; k < 64; k++) {
2   let F = '00000000000000000000000000000000'
3   let g = 0
4
5   if (k <= 15) {
6     F = or32(and32(B, C), and32(not32(B), D))
7     g = k % 16
8   } else if (16 <= j && j <= 31) {
9     F = or32(and32(D, B), and32(not32(D), C))
10    g = (5 * j + 1) % 16
11  } else if (32 <= j && j <= 47) {
12    F = xor32(B, xor32(C, D))
13    g = (3 * j + 5) % 16
14  } else if (48 <= j && j <= 63) {
15    F = xor32(C, or32(B, not32(D)))
16    g = (7 * j) % 16
17  }
18
19  F = sum32(F, sum32(A, sum32(this.premadeSineTable__K__[k].toString(2)
      , M[g])))
```

```
20
21   A = D
22   D = C
23   C = B
24   B = sum32(B, this.leftRotate(F, this.shiftAmount[k]))
25 }
```

Nossa implementação utiliza blocos de `if else` ao invés de desenrolar o `for` loop principal. As primeiras 16 iterações no pseudo código:

Algoritmo 3. Pedaço do pseudo código referente as primeiras 16 iterações do bloco principal:

```
1 ...
2 /* Let [abcd k s i] denote the operation
3 a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
4 /* Do the following 16 operations. */
5 [ABCD  0  7  1]  [DABC  1 12  2]  [CDAB  2 17  3]  [BCDA  3 22  4]
6 [ABCD  4  7  5]  [DABC  5 12  6]  [CDAB  6 17  7]  [BCDA  7 22  8]
7 [ABCD  8  7  9]  [DABC  9 12 10]  [CDAB 10 17 11]  [BCDA 11 22 12]
8 [ABCD 12  7 13]  [DABC 13 12 14]  [CDAB 14 17 15]  [BCDA 15 22 16]
9 ...
```

No nosso código essa parte é representada da seguinte maneira:

Algoritmo 4. Pedaço do nosso código referente as Algoritmo 3:

```
1 ...
2 if (k <= 15) {
3     F = or32(and32(B, C), and32(not32(B), D))
4     g = k % 16
5 }
6 ...
```

Depois das rodadas de embaralhamento os digest são guardados no registradores a0, b0, c0 e d0 usando a função de soma:

Algoritmo 5. guardando valores dos registradores:

```
1 this.a0 = sum32(this.a0, A)
2 this.b0 = sum32(this.b0, B)
3 this.c0 = sum32(this.c0, C)
4 this.d0 = sum32(this.d0, D)
```

Ao final do processamento de todos os blocos de 512 bits o digest é feito com a concatenação dos registradores:

Algoritmo 6. Concatenação para o retorno:

```
1 return this.a0.concat(this.b0.concat(this.c0.concat(this.d0)))
```

1.5.4. Funções bitwise

O algoritmo MD5 usa para o embaralhamento (hashing) dos bits funções bitwise de AND, OR, XOR, NOT, SUM, LEFTSHIFT e RIGHTSHIFT. Em seguida será descrito cada uma das implementações dessas funções auxiliares e exemplos de uso. Todas funcionam considerando que os parâmetros de entrada e a saída são strings de tamanho 32.

Algoritmo 6. AND bitwise usado na implementção:

```
1 export function and32(firstOperand: string, secondOperand: string):
     string  {
2   const totalLength = 32
3
4   const firstOp = firstOperand.padStart(totalLength, '0')
5   const secondOp = secondOperand.padStart(totalLength, '0')
6
7   let returnStr = ""
8
9   for (let i = totalLength - 1; i >= 0; i--) {
10    if ((firstOp[i] === '1') && (secondOp[i] === '1'))
11      returnStr = '1'.concat(returnStr)
12    else
13      returnStr = '0'.concat(returnStr)
14  }
15
16  return returnStr
17 }
```

Exemplos AND:

```
10101010101010101010101010101010
            and32
00000000000000001111111111111111
            =
00000000000000001010101010101010

11111111111111111111111111111111
```

```
            and32
00000000000000000000000000000000
            =
00000000000000000000000000000000

11111111111111111111111111111111
            and32
11111111111111111111111111111111
            =
11111111111111111111111111111111
```

Algoritmo 7. OR bitwise:

```
1 export function or32(firstOperand: string, secondOperand: string):
    string  {
2   const totalLength = 32
3
4   const firstOp = firstOperand.padStart(totalLength, '0')
5   const secondOp = secondOperand.padStart(totalLength, '0')
6
7   let returnStr = ""
8
9   for (let i = totalLength - 1; i >= 0; i--) {
10    if ((firstOp[i] === '1') || (secondOp[i] === '1'))
11      returnStr = '1'.concat(returnStr)
12    else
13      returnStr = '0'.concat(returnStr)
14  }
15
16  return returnStr
17 }
```

Exemplos OR:

```
10101010101010101010101010101010
            or32
00000000000000001111111111111111
            =
10101010101010101111111111111111

11111111111111111111111111111111
            or32
00000000000000000000000000000000
            =
11111111111111111111111111111111

11111111111111111111111111111111
```

```
            or32
11111111111111111111111111111111
            =
11111111111111111111111111111111
```

Algoritmo 7. XOR bitwise:

```
1 export function xor32(firstOperand: string, secondOperand: string):
     string  {
2   const totalLength = 32
3
4   const firstOp = firstOperand.padStart(totalLength, '0')
5   const secondOp = secondOperand.padStart(totalLength, '0')
6
7   let returnStr = ""
8
9   for (let i = totalLength - 1; i >= 0; i--) {
10    if ((firstOp[i] === '1') !== (secondOp[i] === '1'))
11      returnStr = '1'.concat(returnStr)
12    else
13      returnStr = '0'.concat(returnStr)
14  }
15
16  return returnStr
17 }
```

Exemplos XOR:

```
10101010101010101010101010101010
xor32
00000000000000000111111111111111
=
10101010101010101001010101010101

11111111111111111111111111111111
xor32
00000000000000000000000000000000
=
11111111111111111111111111111111

11111111111111111111111111111111
xor32
11111111111111111111111111111111
=
00000000000000000000000000000000
```

Algoritmo 7. NOT bitwise:

```
1 export function not32(operand: string): string  {
2   const totalLength = 32
3
4   const firstOp = operand.padStart(totalLength, '0')
5
6   let returnStr = ""
7
8   for (let i = totalLength - 1; i >= 0; i--) {
9     if ((firstOp[i] === '1'))
10      returnStr = '0'.concat(returnStr)
11    else
12      returnStr = '1'.concat(returnStr)
13  }
14
15  return returnStr
16 }
```

Exemplos NOT:

```
            not32
00000000000000000111111111111111
            =
11111111111111110000000000000000

            not32
00000000000000000000000000000000
            =
11111111111111111111111111111111

            not32
11111111111111111111111111111111
            =
00000000000000000000000000000000
```

Algoritmos 8: SUM bitwise

```
1 export function sum32(firstOperand: string, secondOperand: string):
    string  {
2   const totalLength = 32
3
4   const firstOp = firstOperand.padStart(totalLength, '0')
5   const secondOp = secondOperand.padStart(totalLength, '0')
6
7   let returnStr = ""
8   let carry = false
```

```
9
10   for (let i = totalLength - 1; i >= 0; i--) {
11     const sum = ((firstOp[i] === '1') !== (secondOp[i] === '1') !==
       carry)
12
13     returnStr = sum ? '1'.concat(returnStr) : '0'.concat(returnStr)
14
15     carry =
16       ((firstOp[i] === '1') && (secondOp[i] === '1')) ||
17       ((secondOp[i] === '1') && carry) ||
18       ((firstOp[i] === '1') && carry)
19   }
20
21   return returnStr
22 }
```

Exemplos SUM:

```
11010101010101110000011101101111
            sum32
11010101010101110000011101101111
            =
10101010101011100000111011011110
```

1.6. Problemas relacionados

A mesma foi uma função muito utilizada no passado, inclusive para encriptação de senhas, porém com o aumento da velocidade dos computadores e vaga utilização da mesma, ela se tornou uma alternativa pouco segura, por causa da facilidade de gerar colisões, tópico que iremos visitar a seguir.

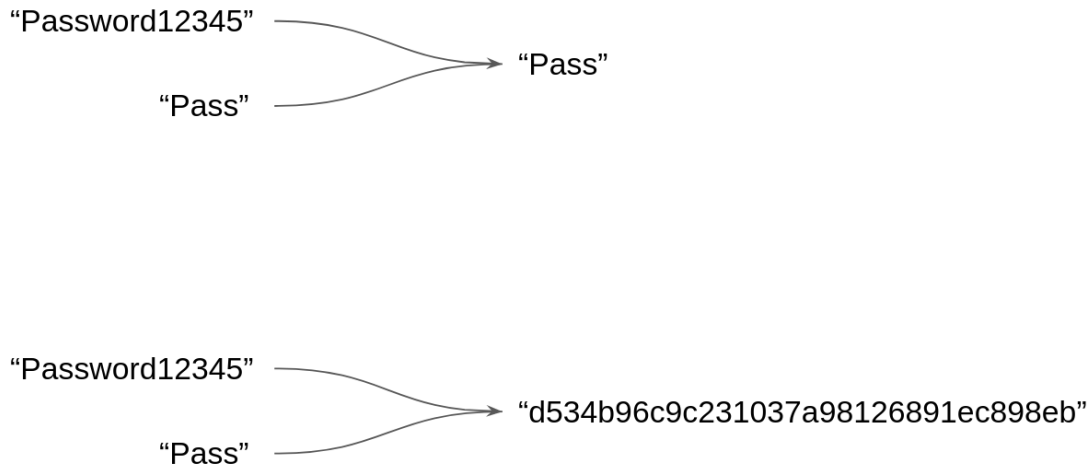

Figure 1.1. Exemplo de colisão. Na figura, ambos "Password12345" e "Pass" compartilham do mesmo hash

Method	Hashes per second
MD5	668896
SHA-1	597014
SHA-256	588235
SHA-3(224-bit)	331674
Bcrypt	336

Figure 1.2. Hashes por segundo de diferentes algoritmos rodados em um browser em um computador moderno

1.6.1.

Colisões Quando funções de hash são usadas no contexto de criptografia, a função de hash utilizada deve ser resistente a colisões. Resistência a colisões pode ser definido que deve ser computacionalmente difícil encontrar as entradas `m` e `m'` tal que `H(m') = H(m)`, sendo que `H` é uma função de hash [Cryptography Made Simple, 2016]. Colisões são um dos maiores problemas quando tratamos sobre hashing. Ela ocorre quando dois pedaços distintos de dados possuem o mesmo valor hash(saída da função de hashing), digital virtual ou digest criptográfico. Isso é um problema pois quando dois arquivos tem o mesmo digest um arquivo pode se passar pelo outro sem ser percebido. Na tabela 1 temos o hashrate de diversas funções de hash. Note-se que o MD5 é o mais computacionalmente fácil.

1.7. Referências

- Smart, Nigel P. Cryptography made simple. Cham, Switzerland: Springer, 2016. Print.
- The MD5 Message-Digest Algorithm, MIT Laboratory for Computer Science and RSA Data Security, Inc. April 1992
- Stephens, Rod. Essential algorithms : a practical approach to computer algorithms. Indianapolis, IN: Wiley, 2013. Print.