Hashing Algorithms

September 19, 2016

Contents

1 Hash Functions

There are several similar terms to "hash function": checksum, fingerprint, cryptographic hash function. Let's start with stating the difference among them.

Hash function — a function that maps some data of variable length into a predefined range of values. That's all. For example a function that maps every integer into itself can be called a "hash function". Result of applying hash function is usually called *hash value* or *digest*.

Checksums, fingerprint functions and cryptographic hash functions are different types of hash function that have special purposes.

Checksum is a hash function used for checking the integrity of the data. For example, if some error occurred during data transfer, checksum function should detect it. Checksum is not meant to be "hard-to-reverse" function, neither it has to be collision-resistant.

Fingerprint functions have more restrictions on their output. They are meant to uniquely identify some piece of data, usually just files. Some of the fingerprint functions are also designed to be resistant against special kinds of attacks.

Cryptographic hash functions have the most strict requirements. First, let's list the usage of cryptographic hash functions and then deduce which properties they should have.

Further in this paper our attention we will be concentrated only on cryptographic hash functions, so we will call them just hash functions.

For additional details see [8].

1.1 Usage of cryptographic hash functions

- Hashing passwords.
  Storing database with plain text user passwords would cause severe problems, if the database is stolen. Instead, a "sophisticated" developer stores hash values of the passwords and whenever a user enters his password its hash is compared to the saved one, this prevents hacker from discovering the password. However, using just hashed values is not enough, because the hacker may have a precomputed table of hashes for frequent passwords. Therefore during the process of storing password webmasters generate some random data called *salt*, append salt to the real password and hash the "salted" password. Database stores both

hash of the salted password and the salt. Hacker can't retrieve password as he doesn't know to which word he needs to append the salt.

- Integrity check.
  Just as fingerprint functions, cryptographic hash functions are used to check the integrity of data, but hash function protects the output not only from random error, but also from an intentional corruption.

- Digital signatures.
  Cryptographic algorithms used for digital signatures usually can't be applied to arbitrarily data, instead a message to be signed is initially hashed and after its hash is signed. Verifying the authenticity of a hashed digest of the message is considered proof that the message itself is authentic.

- Proof of work.
  A proof-of-work system (or protocol, or function) is an economic measure to deter denial of service attacks (DDOS) and other service abuses such as spam on a network by requiring some work from the service requester, usually meaning processing time by a computer. A key feature of these schemes is their asymmetry: the work must be moderately hard (but feasible) on the requester side but easy to check for the service provider.

1.2 Requirements for hash functions

A "good" cryptographic function has to be:

- hard to reverse;
- collision-resistant;
- having avalanche-effect.

These are general requirements for any hash function, however some special-purpose functions have additional requirements. For example having the same length of every output, or inability to retrieve any information about the input. Hash function doesn't have to be "random", but a good "random" function can be used as a "strong" cryptographic hash function.

Let's examine requirements more thoroughly.

"Hard to reverse" means that it's should be computationally difficult to obtain message whose hash is equal to the given one. In application to password hashing that

means, that is impossible to get plaintext password by its hash value. The term "computationally difficult" depends on the context, usually there are some considerations about hackers computing potential and it's possible to choose appropriately strong hash function. For instance, the most efficient attack reverting SHA-2 needs $\approx 2^{128}$ operations, modern computing power of the whole humanity needs thousands of years to perform it.

Every hash has predefined range of values, usually it is a fixed length string, for example SHA-256 outputs 256 bits. Therefore, there are not greater than 2^{256} different SHA-256 hashes. This number is too big to perform a brute force attack.

Collision resistance means it's hard to find two messages that produce the same hash value. In application to digital signatures that means, that it's impossible to change the document without changing the signature.

Usually it's achieved by so-called avalanche effect: a small change in input data would cause significant change in the output. For example:

> SHA224("The quick brown fox jumps over the lazy dog")
> >0x730e109bd7a8a32b1cb9d9a09aa2325d2430587ddbc0c38bad911525
> SHA224("The quick brown fox jumps over the lazy dog.")
> >0x619cba8e8e05826e9b8c519c0a5c68f4fb653e8a3d8aa04bb2c8cd4c

The only difference between messages is trailing period in the second one, but their hashes are completely different.

2 SHA-2

In this section we will describe the specification of SHA-2 and implement SHA-256 using Wolfram Mathematica language.

SHA-2 (Secure Hash Algorithm 2) is a family of cryptographic hash functions. They vary in the length of produced digest: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256. SHA-224 and SHA-384 are truncated versions of SHA-256 and SHA-512 respectively.

SHA-2 is defined in terms of low-level operations, but Mathematica is a high-level language, so the implementation is bulky and slow. While being low-level is a disadvantage for Mathematica implementation, SHA-2 allows very fast implementations to be built-in in modern processors.

Operations of SHA-2 are defined in terms of 32-bit words for SHA-256 and 64-bit words for SHA-512.

We will use a list of 4 bytes to represent a 32-bit word.

Further we will use numbers written in hexadecimal, for example 170 is "0xaa" or aa_{16} or just "aa", if there is no confusion. Two-digit hexadecimal number represents a byte in a natural way, as its value is between 0 and 255, for instance

$$ff_{16} = 255 = 11111111_2.$$

The data to encrypt we will call message. Its length should not be bigger than $2^{64} - 1$ bits ($\approx$ 2 million terabytes).

Implementation will be explicit and mimic low-level data structures, we will use another approach for MD5 code.

Original specification can be found here: [2].

2.1 Description

2.1.1 Padding

Message should be divided into 512-bit blocks, therefore arbitrarily long message should be padded with some data to have length equal to a multiple of 512 bits.

Suppose we have message M, its length is l bits. Padding is done in the following way:

1. Append bit 1.
2. Append k zero bits, where k is the smallest non-negative solution to the equation $l + k + 1 \equiv 448 \mod 512$.
3. Append length of the original message l as a 64-bit binary number.

First of all we need to load the message into Wolfram Mathematica. Mathematica allows to load file as list of bytes using command:

```
msg=Import["~/path/to/file", "Binary"]
```

Or we can enter the string by ourselves:

```
ImportString["abc", "Binary"]
```

This code outputs list `{97,98,99}`. These 3 bytes are representing ASCII codes for the message "abc". Note, that usually operating systems put a newline character at the end of the file, so the message "abc", that is read as a file has length of 4 bytes and its hash differs from the hash of just "abc".

```
l=Length[msg]*8;
```

Though the specification allows to encrypt message of (almost) any length, usual computer files are stored as a collection of bytes and our implementation will work only for that case. That means it cannot be used to hash data of, for example, 15 bits length; it works just for data with length in bits being a multiple of 8.

We need to append "1" bit to the end of the message. As we consider the message a multiple of bytes, adding a bit will result in adding a byte 1000000, it's binary form of the number 128.

```
AppendTo[msg,128];
```

Then we add zero bytes of padding:

```
For[i=0,i<(k-7)/8,i++,AppendTo[msg,0]];
bitLength=IntegerDigits[l,2,64];
length=ImportString[ExportString[bitLength,"Bit"],"Byte"];
message=msg~Join~length;
```

At first this code adds $k-7$ zero bits as we already added 7 by adding byte 10000000. `bitLength` is the length l written in binary as a 64-bit long integer. Variable `length` is `bitLength` written as 8 bytes. `message` is the padded message.

Note, that a single 1 bit padding is vital, because otherwise messages ending with zeros would be the same after padding:

$$1 \to 10\ldots0$$

$$10 \to 10\ldots0.$$

Of course, both of them would produce the same hash, which is unacceptable.

2.1.2 Partition

Next we need to divide message into 512-bit blocks $M_0, M_1, \ldots M_{n-1}$ and each of the blocks partition into 32-bit words.

The following code does this:

```
M[i_,j_]:=message[[64*i+1+4*j ;; 64*i+4*(j+1)]]
```

So `M[i,j]` will produce jth word from ith block with $0 \leq i \leq n-1$ and $0 \leq j \leq 15$. This word is written as a list of 4 bytes.

For example, `M[0,0]` produces $\{97, 98, 99, 128\}$. Now we need to convert a list of 4 bytes into a 32-bit word.

Exercise 1

Implement a function `bytesTo32Word` converting a list of 4 bytes into a list of 32 bits representing these bytes.

Answer of exercise 1

```
bytesTo32Word[list_] := Flatten[IntegerDigits[list, 2, 8]]
```

For example, the code

```
bytesTo32Word[M[0,0]]
```

outputs list `{0,1,1,0,0,0,0,1,0,1,1,0,0,0,1,0,0,1,1,0,0,0,1,1,1,0,0,0,0,0,0,0}`.

2.1.3 Constants

SHA-2 has sixty-four 32-bit round constants $R[i]$ with $1 \leq i \leq 64$ and eight 32-bit initial values $H_1^0, \ldots, H_8^0$.

H_i^0 is equal to first 32 bits of fractional parts of ith prime number for i from 1 to 8. Written in hexadecimal these values are:

$$H_1^0 = \text{6a09e667} \quad H_2^0 = \text{bb67ae85} \quad H_3^0 = \text{3c6ef372} \quad H_4^0 = \text{a54ff53a}$$
$$H_5^0 = \text{510e527f} \quad H_6^0 = \text{9b05688c} \quad H_7^0 = \text{1f83d9ab} \quad H_8^0 = \text{5be0cd19}$$

We will use variables `H[0,i]` for H_i^0. Mathematica allows to input hexadecimal numbers in the following way:

```
H[0,1]:=bytesTo32Word@IntegerDigits[16^^6a09e667,256]
  ...
H[0,8]:=bytesTo32Word@IntegerDigits[16^^5be0cd19,256]
```

These eight values are the initial state of the hash.

Next we need 64 constants for 64 rounds of hashing. These constants are the first 32 bits of fractional parts of cube roots of the first sixty-four primes.

Exercise 2

Implement function to input these values.

Answer of exercise 2

```
R=Table[bytesTo32Word@IntegerDigits[
  Floor[2^32*FractionalPart[(Prime@i)^(1/3)]],256],{i,1,64}];
```

Their values in hexadecimal are:

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```

2.1.4 Functions

Hashing is uses 6 operations:

- bitwise XOR ($\oplus$);
- bitwise AND ($\wedge$);
- bitwise NOT ($\neg$);
- addition modulo 2^{32} ($+$);
- right shift by n bits (S^n);
- right rotation by n bits (R^n).

Each of these operations is meant to act on 32-bit words. Let's define all of them.

For the first three we can use built-in Mathematica functions: `BitXor`, `BitAnd`, `BitNot`.

Next we define a function for summing words modulo 2^{32}. First we translate 32-bit words into decimal integers and after we just sum them modulo 32 and transform the result back to a 32-bit word:

```
wordToInteger[word_]:=FromDigits[word,2];
plus[word1_,word2_]:=IntegerDigits[
  Mod[wordToInteger@word1+wordToInteger@word2,2^32],2,32]
```

Function `plus` can be used in infix form: `word1~plus~word2= plus[word1,word2]`.

Exercise 3

Define a function `shift32Word` that takes 2 parameters: 32-bit word x and a non-negative integer n and returns word x shifted by n positions to the right.

Answer of exercise 3

```
shift32Word[x_,n_]:=PadRight[x,32,0,n]
```

Right rotation can be done by using built-in Mathematica function `RotateRight`.

Next we define 6 functions that are used in every round:

$$
\begin{aligned}
Ch(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
\Sigma_0(x) &= R^2(x) \oplus R^{13}(x) \oplus R^{22}(x) \\
\Sigma_1(x) &= R^6(x) \oplus R^{11}(x) \oplus R^{25}(x) \\
\sigma_0(x) &= R^7(x) \oplus R^{18}(x) \oplus S^3(x) \\
\sigma_1(x) &= R^{17}(x) \oplus R^{19}(x) \oplus S^{10}(x).
\end{aligned}
$$

Exercise 4

Implement all these functions.

Answer of exercise 4

```
Ch[x_,y_,z_]:=BitXor[BitAnd[x,y],BitAnd[BitNot[x],z]]
Maj[x_,y_,z_]:=BitXor[BitAnd[x,y],BitAnd[x,z],BitAnd[y,z]]
Σ0[x_]:=BitXor[RotateRight[x,2],RotateRight[x,13],RotateRight[x,22]]
Σ1:=BitXor[RotateRight[x,6],RotateRight[x,11],RotateRight[x,25]]
σ0:=BitXor[RotateRight[x,7],RotateRight[x,18],shift32Word[x,3]]
σ1:=BitXor[RotateRight[x,17],RotateRight[x,19],shift32Word[x,10]]
```

Next we need a function to expand the 512-bit block (16 words) into 2048-bit (64 words). These 64 words are needed for 64 round of hashing.

$W(i,j) = M(i,j)$ for j from 0 to 15
$W(i,j) = \sigma_1(W(i,j-2) + W(i,j-7) + \sigma_0(W(i,j-15)) + W(i,j-16)$ for j ≥ 16

Exercise 5

Define function $W[i,j]$.

Answer of exercise 5

```
W[i_,j_]:=If[j<=15,bytesTo32Word[M[i,j]],
   σ1[W[i,j-2]]~plus~W[i,j-7]~plus~
   σ0[W[i,j-15]]~plus~W[i,j-16]]
```

2.1.5 Rounds

For every 512-bit block M^i SHA-256 hashing involves 64 rounds. Each round operates on eight 32-bit variables a, b, c, d, e, f, g, h. At the start of each round the values of these variables are set to the intermediate hash values from the previous round.

For the first round the values are set to be constants H_i^0. Then after 64 rounds final values for the variables are found and the new intermediate hash value is obtained by adding the previous intermediate hash value to the variables a, b, d, e, f, g, h (modulo 2^{32}). Round actions are done by using defined above functions and combining intermediate hash values. It can be done by the following Mathematica code:

```
n =Length[message]/64;
For[i=1,i<=n,i++,
 a = H[i-1,1];
 b = H[i-1,2];
 c = H[i-1,3];
 d = H[i-1,4];
 e = H[i-1,5];
 f = H[i-1,6];
 g = H[i-1,7];
 h = H[i-1,8];
 Do[
  T1=h~plus~Σ1[e]~plus~Ch[e,f,g]~plus~R[[j+1]]~plus~W[i-1,j];
  T2=Σ0[a]~plus~Maj[a,b,c];
  h=g;
  g=f;
  f=e;
  e=d~plus~T1;
  d=c;
  c=b;
  b=a;
  a=T1~plus~T2,{j,0,63}];
H[i,1]=a~plus~H[i-1,1];
H[i,2]=b~plus~H[i-1,2];
H[i,3]=c~plus~H[i-1,3];
H[i,4]=d~plus~H[i-1,4];
H[i,5]=e~plus~H[i-1,5];
H[i,6]=f~plus~H[i-1,6];
H[i,7]=g~plus~H[i-1,7];
H[i,8]=h~plus~H[i-1,8];]
```

2.1.6 Testing

One of the most obvious and useful ways to validate the implementation of the algorithm is to test on some data. Let's see the hash of the message "abc".

```
Table[BaseForm[FromDigits[H[n, j], 2], 16], {j, 1, 8}]
```

Outputs this:

> ba7816bf, 8f01cfea, 414140de, 5dae2223, b00361a3, 96177a9c, b410ff61, f20015ad

Which is the needed result. Test vectors and their hashes can be found for instance here.

Let's change our message to the empty one and rerun the code.

> e3b0c442, 98fc1c14, 9afbf4c8, 996fb924, 27ae41e4, 649b934c, a495991b, 7852b855

Which is also the correct value.

2.2 SHA-2 family

Other functions from the family, such as SHA-512, have similar structure, but with other constants, block and word lengths, number of rounds and slightly different functions Σ and σ.

SHA-224 and SHA-384 are just truncated SHA-256 and SHA-512 with other initial vectors. SHA-512/224 and SHA-512/256 are also truncated versions of SHA-256 and SHA-512, but their initial vectors are defined by a special algorithm.

For instance, SHA-512 has 512-bit output, 80 rounds and operations defined for 64-bit words.

Other values can be seen in the following table (all length values are in bits).

Hash function	Digest length	Internal state (intermediate value) length	Block length	Maximal input length	Word length	Number of rounds
SHA-256	256	256	512	$2^{64} - 1$	32	64
SHA-224	224	256	512	$2^{64} - 1$	32	64
SHA-512	512	512	1024	$2^{128} - 1$	64	80
SHA-384	384	512	1024	$2^{128} - 1$	64	80
SHA-512/256	256	512	1024	$2^{128} - 1$	64	80
SHA-512/224	254	512	1024	$2^{128} - 1$	64	80

3 MD5

MD5 (Message Digest Algorithm 5) is a hashing algorithm developed in April 1992 by Ronald Rivest as an improvement to MD4.

Nowadays MD5 is considered insecure, as there are found collisions and its usage as a cryptographic hash function is now deprecated, while it still can be used as a checksum function.

On the contrary to the SHA-2 implementation, the following one will use features of Wolfram Mathematica language and won't explicitly use low-level operations and treat words as integer numbers.

Original specification can be found here: [1].

3.1 Description

MD5 is similar to SHA-2. Both of them are based on Merkle–Damgård construction, it is a quite general method of creating hash functions. Briefly its idea may be described as follows:

- divide the message into blocks of equal length,;
- construct a function (or a collection of functions) that takes as inputs intermediate hash value (the result from the previous block) and the next block of the message;
- iteratively apply the function to the blocks of the message starting with some predefined values called initial vector or initial values(IV).

3.1.1 Padding

Padding is almost the same as that of SHA-2.

Suppose we have message M, it length is l bits. Padding is done in the following way:

1. Append bit 1.
2. Append k zero bits, where k is the smallest non-negative solution to the equation $l + k + 1 \equiv 448 \mod 512$.

3. Append length of the original message l as a 64-bit *little-endian*[1] integer.

3.1.2 Partition

MD5 has the same block length of 512 bits. Partition the padded message into n consecutive 512-bit blocks $M_0, M_1, \ldots, M_{n-1}$.

Next we need to partition the message and pad it. As we've already done these steps for SHA-2 we won't examine it step-by-step:

```
data=Partition[
 Join[FromDigits[Reverse@#,256]&/@
   Partition[PadRight[Append[#,128],Mod[56,64,Length@#+1]],
     4],Reverse@IntegerDigits[8Length@#,2^32,2]]&@
   ImportString["abc","Binary"],16]
```

This code does both partitioning and padding.

3.1.3 Constants and functions

MD5 involves 64 rounds. For each round there is a rotation constant and an addition constant.

Rotation constants:

```
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}
```

Addition constant for ith round is 32-bit fractional part of $|\sin(i)|$.

```
k = Table[Floor[2^32*Abs@Sin@i],{i,1,64}]
```

MD5 operates on 128-bit state that is represented by four 32-bit words. Their initial values are:

[1] That means the least significant byte goes first, see `https://en.wikipedia.org/wiki/Endianness`.

```
h0 = 16^^67452301;
h1 = 16^^efcdab89;
h2 = 16^^98badcfe;
h3 = 16^^10325476;
```

MD5 uses 4 different functions for round computations, i is the number of round:

$$\begin{aligned}
&\text{for } 1 \le i \le 16 \quad f(x,y,z) = (x \wedge y) \oplus (\bar{x} \wedge z),\ g = i - 1 \\
&\text{for } 17 \le i \le 32 \quad f(x,y,z) = (z \wedge x) \oplus (\bar{z} \wedge y),\ g = 5i - 4 \mod 16 \\
&\text{for } 33 \le i \le 48 \quad f(x,y,z) = x \oplus y \oplus z,\ g = 3i + 2 \mod 16 \\
&\text{for } 49 \le i \le 64 \quad f(x,y,z) = y \oplus (\bar{z} \vee x),\ g = 7i - 7 \mod 16.
\end{aligned}$$

We will define f and g in the body of the round loop.

We also need a left rotation of a word, but as we represent words as numbers we can't use `RotateLeft`

Exercise 6

Implement a left-rotate function that can operate on 32-bit integers (not list of bits).

Answer of exercise 6

```
rotateLeft[i_,p_]:=BitOr[BitShiftLeft[i,p],BitShiftRight[i,32-p]]
```

i stands for "integer", p for "positions".

3.1.4 Rounds

Now we have all ingredients to construct the main loop:

```
Do[{a,b,c,d}={h0,h1,h2,h3};
 Do[Which[1<=i<=16,
   f=BitOr[BitAnd[b,c],BitAnd[BitNot[b],d]];
   g=i-1,
   17<=i<=32,
   f=BitOr[BitAnd[d,b],BitAnd[BitNot[d],c]];
   g=Mod[5i-4,16],
   33<=i<=48,
   f=BitXor[b,c,d];
```

```
   g=Mod[3 i +2 ,16] ,
   49<=i <=64,
   f=BitXor[c ,BitOr[b ,BitNot[d]+2^32]];
   g=Mod[7 i - 7 ,16]];
   {a ,b ,c ,d}={d ,rotateLeft[#1,#2]&[
    Mod[a+f+k[[ i ]]+w[[ g +1]] ,2^32] ,r[[ i ]]]+b ,b ,c} ,{i ,
   1 ,64}];{h0 ,h1 ,h2 ,h3}=
 Mod[{h0 ,h1 ,h2 ,h3}+{a ,b ,c ,d} ,2^32] ,{w, data }];
```

As you can see, for every round we compute f and g and combine the intermediate hash values.

Finally, let's write a function that takes a string and returns its hash:

```
md5[ string_String]:=
Module[{ 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} ,
  k=Table[Floor[2^32*Abs@Sin@i] ,{ i ,1 ,64}] ,h0=16^^67452301,
  h1=16^^efcdab89 ,h2=16^^98badcfe ,h3=16^^10325476,
  data=Partition[
   Join[FromDigits[Reverse@#,256]&/@
     Partition[
      PadRight[Append[# ,128] ,Mod[56 ,64 ,Length@#+1]] ,4] ,
     Reverse@IntegerDigits[8Length@#,2^32,2]]&@
    ImportString[string ,"Binary"] ,16] ,a ,b ,c ,d ,f ,g} ,
  Do[{a ,b ,c ,d}={h0 ,h1 ,h2 ,h3};
  Do[Which[1<=i <=16,
    f=BitOr[BitAnd[b ,c] ,BitAnd[BitNot[b] ,d]]; g=i - 1 ,
    17<=i <=32, f=BitOr[BitAnd[d ,b] ,BitAnd[BitNot[d] ,c]];
    g=Mod[5 i - 4 ,16] ,33 <=i <=48, f=BitXor[b ,c ,d];
    g=Mod[3 i +2 ,16] ,49 <=i <=64,
    f=BitXor[c ,BitOr[b ,BitNot[d]+2^32]];
    g=Mod[7 i - 7 ,16]];{a ,b ,c ,d}={d ,
     BitOr[BitShiftLeft[#1,#2] ,BitShiftRight[#1,32- #2]]&[
      Mod[a+f+k[[ i ]]+w[[ g +1]] ,2^32] ,r[[ i ]]]+b ,b ,
     c} ,{i ,1 ,64}];{h0 ,h1 ,h2 ,h3}=
   Mod[{h0 ,h1 ,h2 ,h3}+{a ,b ,c ,d} ,2^32] ,{w, data }];
  "0x"~~
```

```
IntegerString[
 FromDigits[
 Flatten[Reverse@IntegerDigits[#,256,4]&/@{h0,h1,h2,h3}],
 256],16,32]]
```

3.1.5 Testing

Let's start with the empty string.

```
md5[""]
```

Outputs "0xd41d8cd98f00b204e9800998ecf8427e". That's correct!

```
md5["abc"]
```

Outputs "0x900150983cd24fb0d6963f7d28e17f72" which is also the right value. Other test vectors you can find here.

4 Security

4.1 Collision attacks

4.1.1 Birthday paradox

For this subsection we need to have a shallow dive into mathematics.

There is a known problem called "birthday paradox"[2]. It can be stated as follows:

> What is the probability that in a set of 23 randomly chosen people at least two people have birthday on the same day of the year?

By intuition probability is not expected to be very large, as for a particular person the probability of him having birthday on a given day is $\frac{1}{365}$. Therefore the probability for a person to have the same birthday as another person

$$23 \cdot \frac{1}{365} \approx 6.3\%.$$

That's false. Let's have more accurate computations. We denote the probability of all people from the group having different birthdays as $\bar{p}$.

Obviously,

$$\bar{p} = 1 - p,$$

where p is the probability of (at least) two people having the same birthday.

Let's take a random person from the group and remember his birthday. Take another person, the probability he has different birthday is

$$1 - \frac{1}{365}.$$

Let's take third person, the probability of him having the birthday distinct from the previous two persons is

$$1 - \frac{2}{365}.$$

Proceeding this deduction we come up with the final formula:

$$\bar{p} = (1 - \frac{1}{365}) \cdot (1 - \frac{2}{365}) \cdot \ldots \cdot (1 - \frac{22}{365}) \approx 49.27\%.$$

[2]To be precise, it's not a paradox, rather a "pseudoparadox", because it doesn't lead to a logical contradiction, just its correct solution is counter intuitive

Therefore

$$p = 1 - \bar{p} = 50.73\%.$$

So a group of 23 people more likely has two people with the same birthday than not having.

Generalizing that idea shows, that for a "good" hash function that outputs n bits we need only $2^{\frac{n}{2}} = \sqrt{2^n}$ tries to find a pair of inputs that produce the same hash (instead of expected 2^n).

4.1.2 Comparison

Hash function	Security claim	Best attack
MD5	2^{64}	2^{18}
SHA-1	2^{80}	2^{60}
SHA-256	2^{128}	31 of 64 rounds
SHA-512	2^{256}	24 of 80 rounds

Modern PC can find collision for MD5 in few seconds, but breaking SHA-1 is not possible using desktop computer. However, attack is feasible with large amounts of computation power like a cluster or a supercomputer.

Example of two different messages with the same hash (in hexadecimal):

1. 4dc968ff0ee35c209572d4777b721587d36fa7b21bdc56b74a3dc0783e7b9518afbfa200
   a8284bf36e8e4b55b35f427593d849676da0d1555d8360fb5f07fea2
2. 4dc968ff0ee35c209572d4777b721587d36fa7b21bdc56b74a3dc0783e7b9518afbfa202
   a8284bf36e8e4b55b35f427593d849676da0d1d55d8360fb5f07fea2

4.1.3 Preimage attacks

Preimage attack is a process of finding the message that has hash digest equal to the specific one. Birthday paradox is not applicable for this attack, therefore performing a successful preimage attack is much harder than collision attack. For example, both SHA-256 and MD5 are considered strong against this attack today (2016).

Hash function	Security claim	Best attack
MD5	2^{128}	$2^{123.4}$
SHA-1	2^{160}	45 of 80 rounds
SHA-256	2^{256}	43 of 64 rounds
SHA-512	2^{512}	46 of 80 rounds

References

[1] Network Working Group, *The MD5 Message-Digest Algorithm* https://www.ietf.org/rfc/rfc1321.txt

[2] NIST, *Descriptions of SHA-256, SHA-384, and SHA-512* https://web.archive.org/web/20130526224224/http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf

[3] Marc Stevens, *Single-block collision attack on MD5.* https://marc-stevens.nl/research/md5-1block-collision/md5-1block-collision.pdf

[4] Philip Hawkes, Michael Paddon, Gregory G. Rose, *Musings on the Wang et al. MD5 Collision.* http://eprint.iacr.org/2004/264.pdf

[5] Tao Xie, Fanbao Liu, Dengguo Feng, *Fast Collision Attack on MD5.* https://eprint.iacr.org/2013/170.pdf

[6] A. Z. Broder, *Some applications of Rabin's fingerprinting method. In Sequences II: Methods in Communications, Security, and Computer Science.* Springer-Verlag, 1993, pages 152-153

[7] Katz, Jonathan and Lindell, Yehuda, *Introduction to Modern Cryptography.* Chapman & Hall/CRC, 2007

[8] Bruce Schneier, *Applied Cryptography* John Wiley & Sons, Inc., 1996

[9] Douglas R. Stinson, *Cryptography: theory and practice.* CRC Press, 1995