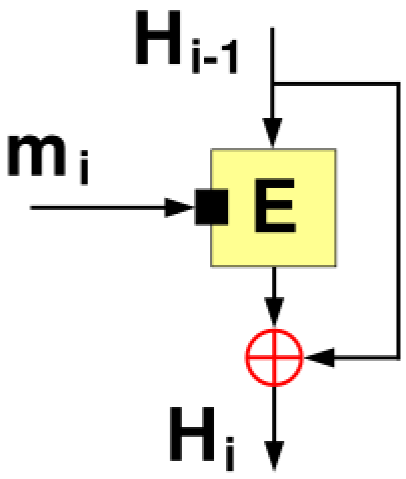
# Something about random and hashing

MD5 and others

Juan Chamero, jach_spain@yahoo.es lectures
Discrete Structures & Algorithms, ds_010, as of April 2006



The Davis-Meyer hash Construction image, from Wikipedia

## Random (101)

Students use “random (101)” function with 101 as argument most times mechanically. However its comprehension will aid to use these types of functions more properly. As a mere “coincidence” we may notice that 101 is the closest number to the 100 prime number. Prime numbers are like “unbreakable” numbers, we can not “fold” them, and they can not be factorized except by themselves and by the unity besides. This characteristic make them appropriates as “seeds” in “pseudo random” generation, like those performed by LFSR’s.

Note: In the particular case of 101 its election permits to generate pseudo random numbers ranged between 0 and 100.

## The Zero

In these types of analysis we realize the importance of the “zero”. We give it for granted. Before its discovering, numbers were represented aided by many symbols and for writing practice the set of symbols grew with the size of numbers represented. Let’s review a little the Roman System.

With symbols I, II, III, IV, V, VI, VII, VIII they represented numbers from 1 to eight with only two elementary symbols (I) and (V). This system was a sort of “quinary” system, based perhaps in the fingers of a hand but this logic found a first challenge when trying to cope with numbers greater than 5, entering into some asymmetry like VI for 6, VII for seven and VIII for eight. Next challenge would be the representation of 10 things and they solved it by creating a third elementary symbol: X ⇔ 10. They complete the series from 1 to 10 by the trick IX to represent nine, meaning “one less than X”. Now let’s proceed: XI, XII, XIII, XIV, XV, XVI, XVII, XVIII, XIX, XX!. So they realized that with these three symbols: [I V X] they could only represent from 1 to 39 (I to XXXIX). Well another challenge to overcome. They solved it this time by creating the symbol L for fifty, then XL ⇔ 40 until L ⇔ 50, and they continue: LX, LXX, LXXX but how do they represented 90 things?. Well they imagined a similar solution to the X challenge creating the Roman “culturally necessary” C for 100, 100 years, 100 soldiers, etcn and repeat the X trick with IC for 90. Along this way they created the M for one thousand and D for 500. They were relatively satisfied because they were not used to manage great numbers.

The advent of zero was really a giant qualitative step!. Positional systems could be mechanically imagined like engines with discs or wheels mounted along a large axis, with each disc/wheel divided in as many parts as the base numbering system, for instance 10 parts if the engine is designed to make counts in the decimal system.

The counting process begins at one end, generally the right end, by rotating the first disc via a driver that only activates this disc. When the user shifts 360 degrees equivalent to 10 positions, a sort of “feedback” mechanism activates to shift one tenth (36 degrees) the second disc. At its turn the mechanism “remember” the relative position of each disc mounted of the common axis. When the second disc completes 360 degrees the mechanism transmits a mechanical “activation” pulse to gear 36 degrees the next disc to its left, and so on and so forth as you may easily imagine. In the sixties not too much ago, these machines were extensively used by engineers in their computations!. Addition and multiplication was performed driving the machine clockwise and counterclockwise for subtraction and division.

## Hashing

A “message hash” is a “message digest”, generally a number constructed based in the text message considered as a text string but of a size significant smaller. The idea is to transform the text string in a number that identifies the string amap, “as much as possible”. This transformation is of type “many to one” and as such different strings may share the same hash number. Hashes are generated by formulae intended to have a low probability that some other text strings share the same hash number. Some applications are security oriented: the sender generates a hash based on the message to be sent, somehow added to it and the recipient decrypts both, the message and the hash number, generates its own hash number and compare the two. If the match comes true there is a high probability that the message was correctly transmitted.

Hashing could also be used to store and access information based on alphanumerical data. One common use is to transform lists of names of size m. It would be fine to find a formula that for each name provides a hash number, ideally one out of m possibilities. If for example m=10,000, we could design a formula that transforms each 30 characters name in an 16 bits (2 bytes) hash number because at first sight a space of 2^16 - 1 = 65,535 possible hash numbers has enough room to accommodate the 10,000 names. If this formula probes to be efficient the economy in access would be substantial!.

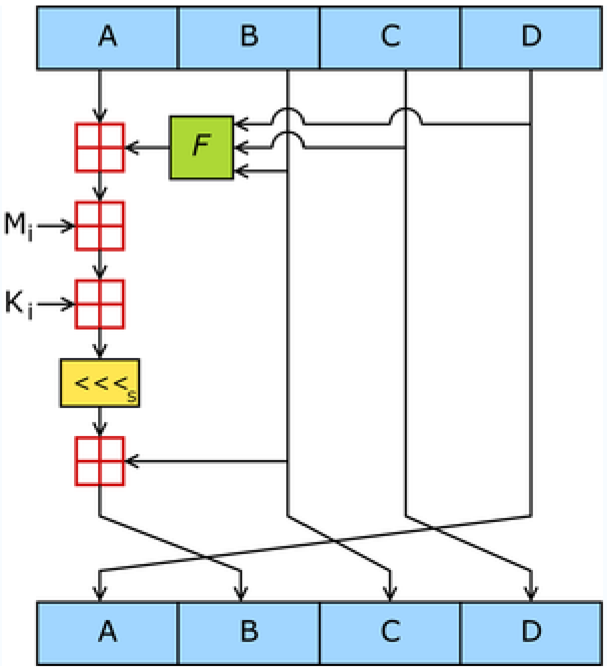
### Hash collisions

It occurs when two distinct data inputs produce identical outputs. Most hash functions have potential collisions but it is highly desirable to design hash formulae that map almost every possible input in different outputs (or at least those inputs of most frequent appearance) of a fixed length. As an example we will analyze the MD5 Message Digest Algorithm 5 created by Ron Rivest, one of the inventors of the RSA Algorithm.

### MD5

This hash number hast a fixed length of 128 bits. It has been used to check files adulterations and to store passwords. MD5 processes a variable length message that is split in chunks and each chunk in 64 consecutive segment pieces Mi into a fixed-length output of 128 bits. Each chunk may have up to 512-bits. Incomplete messages are padded in order to become divisible by 512.

States of 32 bits (A, B, C, and D) are furiously (see below concepts “confusion” and “diffusion”) mixed and then interleaved (A to B, B to C, C to D and D to A) in parallel 32 bits at a time. The mixing core is in A where its content in nonlinearly mixed with B, C and D via a nonlinear Function F, then mixed with two endogenous contributions (in many cryptographic algorithms state registers are injected either linear or non linear with two keys: in this algorithm one key is a piece of the message to be hashed) the corresponding Mi, and an “Operation key Ki” that changes at each operation. Then the result which destiny will be B experiment a shift rotation by s places and finally it’s mixed with B content.



From Wihipedia, MD5

MD5 performs 64 of these operations, grouped in four rounds of 16.

*F* is a non linear function that changes at each round.
*$M_i$* denotes a 32-bit block of the message input, and
*$K_i$* denotes a 32-bit constant, different for each operation.
⋘$_s$ denotes a left bit rotation by *s* places;
*s* varies for each operation.
⊞ It means addition modulo $2^{32}$.

Has all this paraphernalia some logic?. Surely it has. However most mixing and coding work is artisan, scripts should be experimented to see their reliability and efficiency. What is pursued is endurance and proficiency to prevent and block hacker's attacks. Non linear transformations make difficult attacks and inferences in certain cryptanalysis tasks such as "reverse engineering" and "going backwards". Another sought after characteristic is significant output differences as a function of minor alterations in the message. Minor alterations performed over 43-byte ASCII input messages for MD5 .

"The quick brown fox jumps over the lazy dog" ⇔ 9e107d9d372bb6826bd81d3542a419d6

Changing the d of "dog" to c

"The quick brown fox jumps over the lazy **c**og" ⇔ 1055d3e698d289f2af8663725127bd4b

This example brings to our attention two polar characteristics of a good hashing algorithm. If m bits of message are converted to n bits of its message digest being n<<m, 2^(m-n) would be the expected number of collisions (and code "aliases") provided the hashing function is evenly distributed along its spectrum. One polar characteristic would be the "distance" (semantic, orthographic, syntactic, ….) among possible colliding messages; another one would be the "distance" of output results of hashing for minor alterations of the message (semantic, orthographic, syntactic, ….). It would highly desirable to have hashing functions with results (as MD5 in the above examples) significantly distant among them for minor alterations, namely within a predefined "neighborhood".

**MD5 seudocode**

```
//Note: All variables are unsigned 32 bits and wrap modulo 2^32 when calculating
//Define r as the following
var int[64] r, k
r[ 0..15] := {7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22}
r[16..31] := {5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20}
r[32..47] := {4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23}
r[48..63] := {6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21}
```

Note1: To settle **"lefrotate"** argument. You may see the logic of this artisan four shifts (s) more or less evenly distributed along a 32 bits space.

```
//Use binary integer part of the sines of integers as constants:
for i from 0 to 63
    k[i] := floor(abs(sin(i + 1)) × 2^32)
```

Note2: a peculiar way to generate a sort of random 32 bits word with value ranged between 0 and1.

```
//Initialize variables:
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

Note3: another peculiar way to generate four seeds constants. In hexadecimal if we have the sequence 01234567 89ABCDEF, we permute them pair wise from left to right and vice versa. However the distribution of 0's and 1's is not well equilibrated: 20 zeroes and 12 ones in h0 and h3 and 12 zeroes and 20 ones in h1 and h2.

```
//Pre-processing:
append "1" bit to message
append "0" bits until message length in bits ≡ 448 (mod 512)
append bit length of message as 64-bit little-endian integer to message

//Process the message in successive 512-bit chunks:
for each 512-bit chunk of message
    break chunk into sixteen 32-bit little-endian words w(i), 0 ≤ i ≤ 15

    //Initialize hash value for this chunk:
    var int a := h0
    var int b := h1
    var int c := h2
    var int d := h3

//Main loop:
    for i from 0 to 63
        if 0 ≤ i ≤ 15 then
            f := (b and c) or ((not b) and d)
            g := i
        else if 16 ≤ i ≤ 31
            f := (d and b) or ((not d) and c)
            g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47
            f := b xor c xor d
            g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63
            f := c xor (b or (not d))
            g := (7×i) mod 16

        temp := d
        d := c
        c := b
        b := ((a + f + k(i) + w(g)) leftrotate r(i)) + b
        a := temp
```

Note 4: the core of the algorithm. See how words w(g) are generated. First 16 from w(1) sequentially till w(15); second series w(1), w(6), w(11), w(0), w(5),….; third series w(5), w(8), w(11), w(14),….; and the fourth series w(0), w(7), w(14), w(5), …

```
//Add this chunk's hash to result so far:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d

var int digest := h0 append h1 append h2 append h3 //(expressed as little-endian)
```

## Desired Properties of Hash Functions

Source: Prof. Alan Kaminsky
Rochester Institute of Technology -- Department of Computer Science

He defines the virtues One Way Hash Function must have, that maps an arbitrary-length input message M to a fixed-length output hash H(M) with the following properties:

- Given a hash H(M), be difficult to retrieve the message M.

- *Second pre image resistant:* Given a message $M_1$, be difficult to find another message $M_2$ such that $H(M_1) = H(M_2)$.

- *Collision resistant:* be difficult to find two messages $M_1$ and $M_2$ such that $H(M_1) = H(M_2)$.

Note 5: Coherent properties that match our reflections above.

**Avalanche effect**

This effect is a highly desired property of hash functions and refers to the output distance of results when the message experiment minor alterations. It also refers to the propagation process of those minor alterations, for instance only one bit, along the hashing algorithm. This concept was of Shannon concern and defined as Confusion and Diffusion: C*onfusion* in reference to make the relationship between keys and the ciphertexts as complex and involved as possible; D*iffusion* refers to the property that redundancy in the statistics of plaintexts is "dissipated" in the statistics of ciphertexts.

Diffusion is associated with dependency of bits of the output on bits of the input. In a cipher with good diffusion, flipping an input bit should change each output bit with a probability of one half!. This is known as the **Strict Avalanche Criterion**. Now we may understand better the logic of M5D. Effectively the substitution of one symbol by another brings confusion meanwhile transposition brings diffusion. If a cryptographic hash function does not exhibit the avalanche effect to a significant degree hackers may easily unveil the input, being given the output.

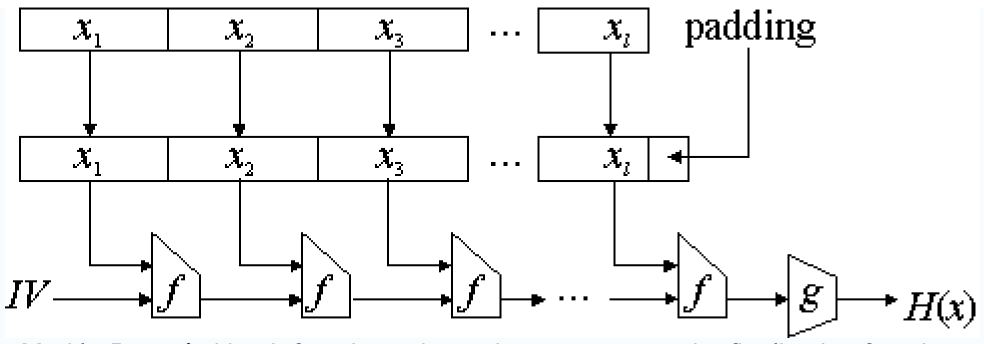
Constructing a cipher or hash to exhibit a substantial avalanche effect is one of the primary design objectives. For these reasons most hash functions process large data blocks and use block ciphering. This type of ciphering works by executing in sequence a number of simple transformations such as substitution, permutation, and modular arithmetic mixing bits via iterations in several *rounds* of the same core process.

Note 6: The Bit Independence Criterion (BIC) states that output bits j & k should change independently when any single input bit i is inverted, for all i, j and k. Most Boolean functions satisfy these two criteria.

## The Merkle-Damgård hash function

Source: Merkle-Damgård, from Wikipedia

It is a generic construction of a cryptographic hash function. All popular hash functions follow this generic construction. A cryptographic hash function must be able to process an arbitrary-length message into a fixed-length output. This can be achieved by breaking the input up into a series of equal-sized blocks, and operating on them in sequence using a compression function that processes a fixed-length input into a shorter, fixed-length output. The compression function can either be specially designed for hashing or be built from a block cipher. The compression function is based on a block cipher that is specially designed for use in a hash function. The Merkle-Damgård hash function breaks the input into blocks, and processes them one at a time with the compression function, each time combining a block of the input with the output of the previous round. They stated that if the compression function is collision-resistant, then the hash function will be also. To accomplish this messages should be padded adding embedded the length of the original message. This is called Merkle-Damgård strengthening.



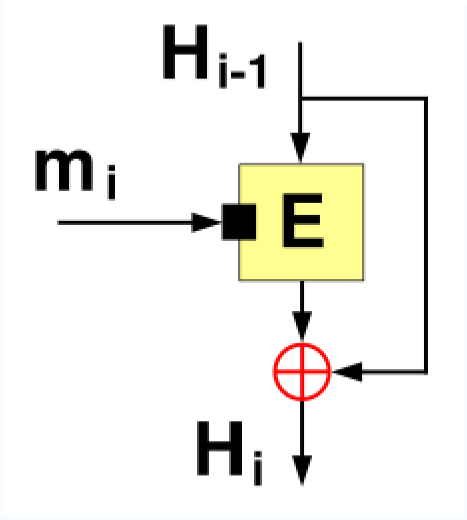
Merkle-Damgård hash function schematic. g represents the finalization function.

In the diagram, the compression function is denoted by f, and transforms a fixed length input to an output of the same size. The algorithm starts with an initial value, the “seed” or initialization vector (IV). For each message block, the compression function f takes the result so far, combines it with the message block, and produces an intermediate result. The last block is padded with zeros as needed and bits representing the length of the entire message are appended.

The finalization function can have several purposes such as compressing a bigger internal state (the last result) into a smaller output hash size or to guarantee a better mixing and avalanche effect on the bits in the hash sum.

A hash function must be able to process an arbitrary-length message into a fixed-length output. This can be achieved by breaking the input up into a series of equal-sized blocks, and operating on them in sequence using a compression function. The compression function can either be specially designed for hashing or be built from a block cipher. The last block processed should, the “length padding block is crucial to the security of this construction.

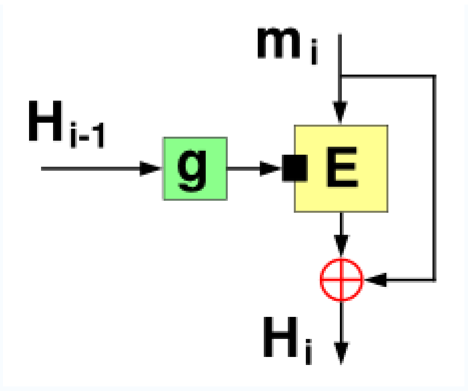
## The Davies-Meyer Logical compression unit



The Davies-Meyer hash compression function feeds each block of the message ($m_i$) as the key to the block cipher. It feeds the previous hash value ($H_{i-1}$) as part of the text to be encrypted. The output ciphertext is then also XORed with the previous hash value ($H_{i-1}$) to produce the next hash value ($H_i$). In the first round when there is no previous hash value it uses a constant pre-specified initial value ($H_0$).

H(i) = E | modified by m(i) + H(i-1)

If the block cipher uses for instance 256-bit keys then each message block ($m_i$) is a 256-bit chunk of the message. If the same block cipher uses a block size of 128 bits then the input and output hash values in each round is 128 bits. Variations of this method replace XOR with any other group operation. This operation hast its dual, inverse operation Matyas-Meyer-Oseas as depicted below

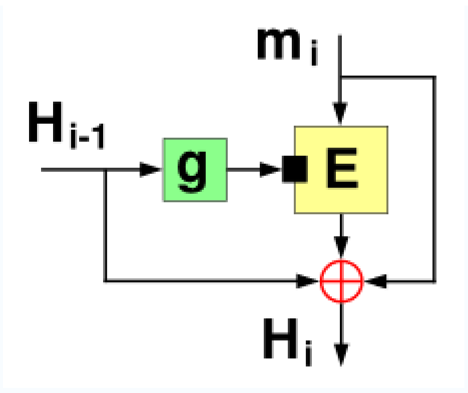


It feeds each block of the message ($m_i$) as the text piece to be encrypted. The output ciphertext is then also XORed with the same message block ($m_i$) to produce the next hash value ($H_i$). The previous hash value ($H_{i-1}$) is fed as the key to the block cipher. In the first round when there is no previous hash value it uses a constant pre-specified initial value ($H_0$).

If the block cipher has different block and key size the hash value ($H_{i-1}$) will have the wrong size for use as the key. The cipher might also have other special requirements on the key. Then the hash value is first fed through the function g( ) to be converted/padded to fit as key for the cipher.

H(i) = E.| modified by g.H(i-1) + m(i)

**The Miyaguchi-Preneel unit**



The Miyaguchi-Preneel hash compression function is an extended variant of Matyas-Meyer-Oseas. It was independently proposed by Shoji Miyaguchi and Bart Preneel. It feeds each block of the message ($m_i$) as the piece of text to be encrypted. The output ciphertext is then XORed with the same message block ($m_i$) and then also XORed with the previous hash value ($H_{i-1}$) to produce the next hash value ($H_i$). The previous hash value ($H_{i-1}$) is fed as the key to the block cipher. In the first round when there is no previous hash value it uses a constant pre-specified initial value ($H_0$).

If the block cipher has different block and key size the hash value ($H_{i-1}$) will have the wrong size for use as the key. The cipher might also have other special requirements on the key. Then the hash value is first fed through the function g( ) to be converted/padded to fit as key for the cipher.

H(i) = E | Modified by g.H(i-1) + H(i-1) + m(i)

## Recommended Lectures

- WHIRLPOOL – an Open Source Contribution
- Source: The Whirpool Hash Function
- Hashing for images, Towards a New Iterative Geometric Methods for Robust Perceptual Image Hashing
- NESSIE Project: New European Schemes for Signatures, Integrity, and Encryption.
- Information Society Technologies, European Union
- Technion, Israel Institute of Technology, Computer Science Department
- Information Security Group, from Royal Holloway University, London, UK