Universidad
Carlos III de Madrid

SeTI Research group · *Computer Science Department*

Carlos III University of Madrid

Hash functions

IT Security practices
Course 2010/2011

Jorge López Hernández-Ardieta / Adapted by Iván Rivera 2010/2011

Table of contents

1. Assignment goal

The aim of this assignment is to develop different hash functions, having different complexities, in order to instruct the student in how to design those functions. Moreover, the student could be able to study the security level of these functions.

2. Assignment description

A hash function H receives as input a data block of undefined length (taken from an entire message M), and computes a compressed value of fixed length (hash):

$$H(M) = hash$$

This input is normally completed with some padding data so its length is a multiple of a given value.

A desired property of a hash function is that, given a big set of input messages, their corresponding hash value should be as random as possible. Thus, a little change in the input messages must produce a big change in the output hash value. Hash functions must also fulfill other properties:

- The input message must be any bits long.
- The output hash value must have a defined and fixed length.
- Computing the hash value must be fast in both software and hardware implementations.
- One-way property: Given a hash value h, it should be computationally infeasible to find a message M' whose hash value coincides with h. *Given h, find M' / H(M') = h*
- Weak collision resistance: Given a message M, it should be computationally infeasible to find a message M' whose hash value coincides with the one of M. *Given M, find M' ≠ M / H(M) = H(M')*
- Strong collision resistance: It should be computationally infeasible to find two messages M and M' with the same hash value. *Find M and M', M ≠ M' / H(M) = H(M')*

- The output must satisfy the pseudo-randomness requirements

Each hash function has a hash space of size |h| (in bits)

$$|h| = 2^n$$

Because there are infinite possible input messages, with variable sizes, and given that a hash function generates hash values of a fixed size (n), we can find two messages M and M' that generate the same hash value. This fact is known as **collision**.

A computationally infeasible problem means that there are no searching collision algorithms or techniques more efficient than brute force. If the hash space is large enough, it's thought that, using existing HW or SW resources it is not possible to find a collision in a reasonable time. The hash function design must take into account this property.

Different hash functions, each one having a greater complexity than the previous one, will be developed in this practice.

3. Student tasks

This section introduces the three different parts that must be completed by the student. Information and exercises (with their corresponding percentage weight on the final grade) are given for each part.

3.1 Supporting functions

The student must implement three supporting functions to be invoked from the hash functions.

3.1.1 Inclusion of the message length

Appending the message length L to message M increases the difficulty of finding a collision, considering that the length is inserted at a byte level, so that numbers lower than 256 would be inserted on a single byte. In this case, the attacker should find two messages of equal length that generate the same hash value, or two messages of different length that, once their lengths are appended, they generate the same hash value.

The task requested in this part of the practice is:

Exercise	Mandatory/ Optional	Weight (% on the final assessment)
• Implement the method **`public byte[] anyadirLongitud(byte[] datos)`** of the class **`Datos.java`**.	Mandatory	5%

The message must be at most 255 bytes long (see the Annex Document - Simplifications). Therefore, the length value (measured in bytes) could be stored in a single byte, which has to be added to the message in the MSByte of the message (last position).

3.1.2 Padding

Depending on the hash function design, the message length must probably fulfill some requirements. A padding function appends at the end of the message a set of information, based on a defined pattern, with the goal of having a total length multiple of a given value.

Since the block size of the data to be processed is always 8 bytes (see the Annex Document - Simplifications), enough padding characters must be added into the last positions of the array so that the length of the message be a multiple of 8.

The task requested in this part of the practice is:

Exercise	Mandatory/ Optional	Weight (% on the final assessment)
• Implement the method **`public byte[] anyadirPadding(byte[] datos, int tamanyoBloque, int tipoPadding)`** of the class **`Datos.java`**.	Mandatory	5%

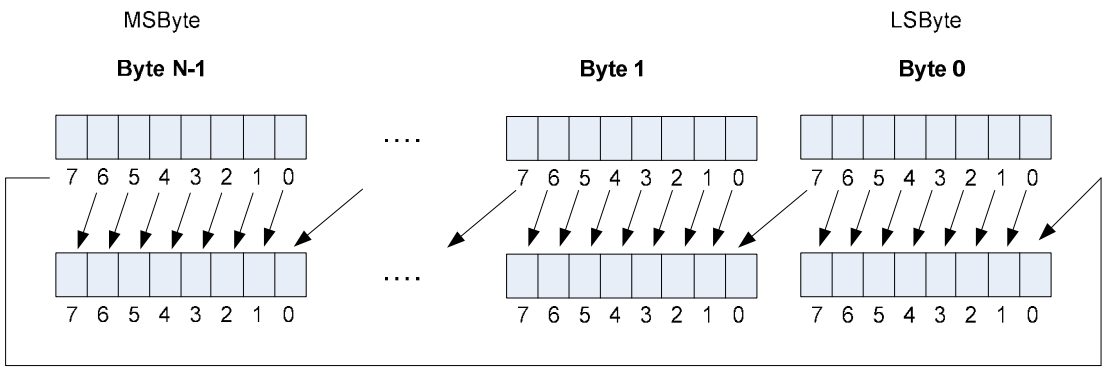
There are 3 different types of padding defined:

- PADDING_0x0n: Add as many 0's as necessary
- PADDING_0x1n: Add as many 1's as necessary
- PADDING_0x10n: Add as many 0's as necessary, except the first bit which will be set to '1'

The student must implement each type of padding into the same method (looking for the parameter *tipoPadding*).

3.1.3 Shifting the byte array in one bit

This method has to left shift the array of bytes in 1 bit. The operation to be performed is shown in the next figure:



The task requested in this part of the practice is:

Exercise	Mandatory/ Optional	Weight (% on the final assessment)
• Implement the method **`public byte[] rotarIzquierdaBit (byte[] a)`** of the class **`Datos.java`**.	Mandatory	5%

3.2 Basic hash function: XOR

One of the simplest yet insecure hash functions applies an exclusive OR (XOR) over each of the n length blocks of the input message M. The output hash value will be n bytes long.

More precisely, this hash function operated with the message applying a binary XOR in each block as follows:

$$C_i = b_{i,1} \text{ XOR } b_{i,2} \text{ XOR } ... \text{ XOR } b_{i,m}$$

Being:

- C_i the i^{th} bit of the hash, with 1<= i <= n
- *m* the number of blocks (of size n) of the message
- $b_{i,j}$ the i^{th} bit of the j^{th} block
- *n* the block size. It coincides with the generated hash value size
- *XOR* bitwise exclusive OR operation

This hash function is normally used into cyclic redundancy check and works well when verifying random data integrity. However, when the input message isn't random (for instance a text), the structure of the information may cause a serious deterioration of the effectiveness of the hash. For example, the MSBit of the ASCII characters will be always 0. For that, when computing a hash value of 128 bits, the real effectiveness would be 2^{-112} instead of 2^{-128} (16 bits will always be 0). This situation would decrement the amount of operations required to obtain a collision with the brute force technique.

Moreover, no avalanche effect occurs, because an original text modification would not affect the entire hash value.

The task requested in this part of the practice is:

Exercise	Mandatory/ Optional	Weight (% on the final assessment)
• Implement the class **`FuncionResumenXORBasica.java`** a) Without including the message length b) Including the padding, if necessary	Mandatory	10%
• Solve the next question: a) Generate 2 messages, M and M', whose hash values coincide. Justify the selection.	Optional	5%

Requirements and simplifications for this part are:

- The input message length is variable, with a maximum of 255 bytes.
- The block size must be 64 bits long (8 bytes).
- The hash value must be 64 bits long (8 bytes).
- The message must be completed with some padding to reach a size multiple of the block length (8), using the auxiliary function **`anyadirPadding`**. The padding type must be chosen by the student.

3.3 Improved hash function: Shifted XOR

A simple way to improve the security of the previous hash function is to rotate (circular shift) one or more bits from a temporal hash value calculated in each step.

This technique can be summarized as follows:

1. Initialize with 0's the hash value (n bits long).
2. For each block i of size n extracted from the message:
    a. Shift the hash value 1 bit to the left (circular shift)
    b. Apply a XOR operation between the block i and the hash value
    c. Update the hash value with the previous one

where n stands for the block size. Please note that this value is also the same as the hash value length.

This simplification improves the input randomness. Moreover, it corrects repetitions or existing patterns from the input message.

The task requested in this part of the practice is:

Exercise	Mandatory/ Optional	Weight (% on the final assessment)
• Implement the class **`FuncionResumenXORDesplazamiento.java`** c) Without including the message length a) Including the padding, if necessary	Mandatory	15%
• Solve the following question: a) Generate 2 messages, M and M', whose hash values coincide. Justify the selection. Indicate which kind of attack could be applied using this scheme (pre-image attack, second image attack or collision attack). b) What's the effect, in terms of security, of initializing the hash value with 0s? Justification. c) Suppose that the block size is m bytes, the message length is a multiple of 8*m*m, and that we're using ASCII characters (each of them can be stored in 1 byte). What would happen? What would happen if the initial hash value were public?	Optional	a) 10% b) 5% c) 5%

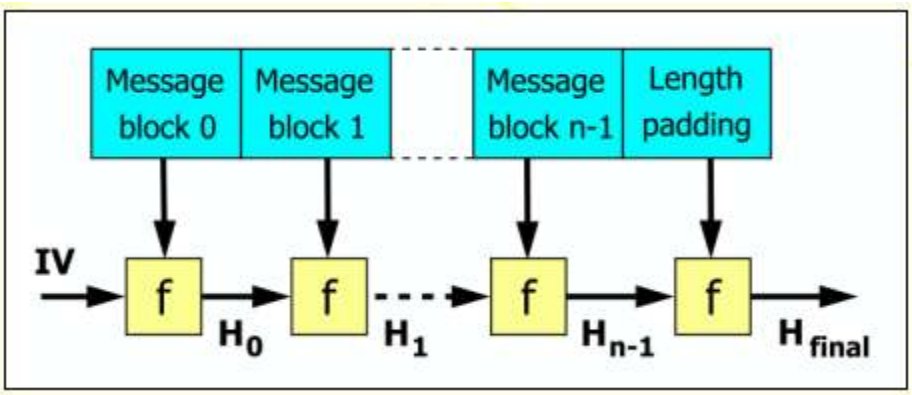
Requisites and simplifications in this part are:

- The input message must be any bytes length, with a maximum of 255 bytes.
- The block size must be 64 bits (8 bytes).
- The hash value must be 64 bits long (8 bytes).

- The message must be completed with some padding to reach a size multiple of the block length (8), using the auxiliary function **`anyadirPadding`**. The padding type must be chosen by the student.

3.4 Enchained hash function

In this phase, the student must develop a secure hash function based on the Mekle-Damgard structure. The following figure shows the general scheme of that kind of functions:



The core of the scheme is the function f, named **compression function**, which is applied in each algorithm iteration. More specifically:

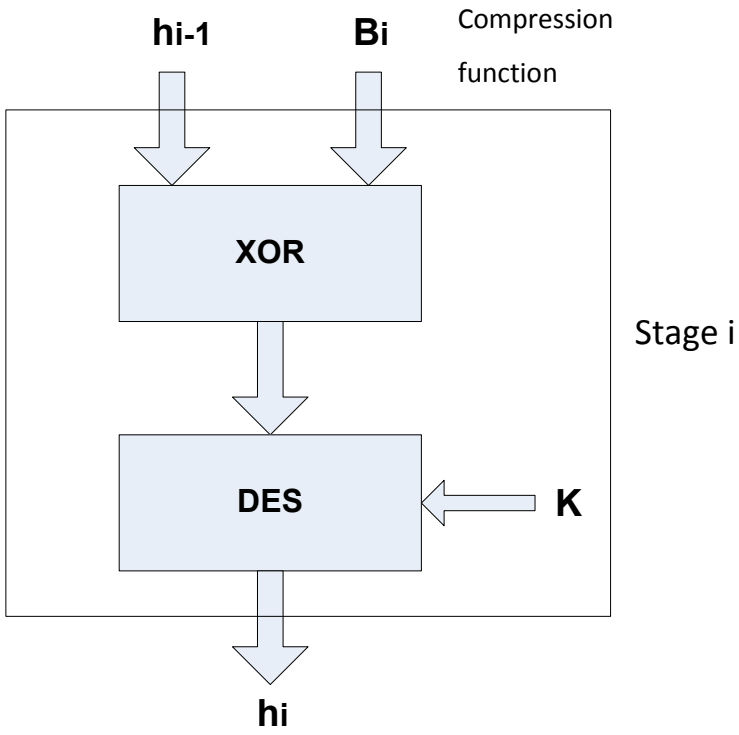
- The length of the message **M** is appended at the end.
- The message is then divided into blocks **B** of size *b*. If the message length isn't multiple of *b*, then a padding is appended, where the first bit is '1' and the rest are '0'.
- The compression function receives 2 inputs: previous output (or IV if it is the first stage) and the corresponding stage block.
- Each stage produces a n bits hash value, being n also the final hash value length.

It's shown that, if the compression function is collision resistant, so is the hash function. That's why the most important phase when using this hash function is the design of the compression function.

The hash function will be based in a simplification of the Merkle-Damgard structure, especially when implementing the compression function. The scheme will follow the philosophy of the CBC (*Cipher Block Chaining*) mode of operation.

- The compression function will apply a XOR between the input message block and the previous stage result, or the IV if it is the first stage.
- Then the resulting 64 bits block will be encrypted using the DES cipher provided along with the code of the practice.

The following figure shows the compression function design requested.



The task requested in this part of the practice is:

Exercise	Mandatory/ Optional	Weight (% on the final assessment)
• Implement the class **`FuncionResumenEncadenada.java`**. a) Including the message length b) Including the padding (if necessary), being compliant with the given specification	Mandatory	20%
• Solve the next issues:	Optional	5%

Exercise	Mandatory/ Optional	Weight (% on the final assessment)
a) What would happen if the encryption operation were removed from the compression function? Justify.		

Requisites and simplifications in this part are:

- The input message must be any bytes length, with a maximum of 255 bytes.
- The block size must be 64 bits (8 bytes).
- The hash value must be 64 bits long (8 bytes).
- The message must be completed with some padding to reach a size multiple of the block length (8), using the auxiliary function **`anyadirPadding`**. The padding type must be chosen by the student.
- Both the final and intermediate hash values must be 64 bits (8 bytes) long.
- The message length must be calculated and appended at its end, using the auxiliary function **`anyadirLongitud.`**
- The Initialization Vector (IV) must be 8 bytes long (64 bits), and it value is fix and established in the given code.

3.5 Calculating collisions and result analysis

In order to distinguish the robustness of the previously implemented hash functions, the student must implement a method to calculate several hash values of a big set of input messages.

Concretely, the student must calculate:

- Total number of different hash values calculated.
- The number of hash values, and the percentage on the total, where at least one collision has been found.
- The total number of collisions found.

Students must perform these calculations for each one of the hash functions implemented.

Note that, given a function and two equal input messages, the output hash value will be the same, but this must not be counted as a collision.

The task requested in this part of the practice is:

Exercise	Mandatory/ Optional	Weight (% on the final assessment)
• Implement the method **`calcularTablaColisiones`** of the class **`PruebaColisiones.java`**. a) Execute the method for each implemented hash function, using buffer reader sizes (messages length) of 20 and 255 bytes. b) Analyze, compare and comment the obtained results.	Mandatory	10%

The method to obtain the input messages is as follows. A big file must be read by parts, each one having a fixed length. Then, the hash value for each of them must be calculated. The file to be used can be found at http://www.gutenberg.org/etext/673.

The class **`PruebaColisiones.java`** is ready to apply this strategy:

- The *tamaño_buffer_lectura* parameter indicates the segment size, which will be that of the input messages for each hash function. In other words, the file must be decomposed into different messages (segments) of size *tamaño_buffer_lectura*. Then, the hash value for each of these segments must be calculated. Finally, it must be counted how many of them collide.

- In any case, the tamaño_buffer_lectura must be lower than 255.

4. Assessment criteria

Each part described in the section **Student tasks** contains the mandatory and the optional tasks, as well as the weight (in percentage) of each task with respect to the total mark of this practice (**1 point**). For the optional points to be taken into account the mandatory part must be done.

5. Practice delivery

Results to be delivered of this practice are:

- The full Java **STIgrado10_p3** project with the corresponding source code.
- A **Document** indicating the conclusions, analysis and the answers to the questions raised in the different parts of the practice.

Both the project and the document must be delivered into a common ZIP file with the name:

STI-grado-89 P3-YYYYYYYYY_ZZZZZZZZZ.zip

YYYYYYYYY, ZZZZZZZZZ → complete NIA

Delivery must be done through **Aula Global 2**. Deadline is on **April 27th, 2011.**

6. Clarifications on the delivery and grading of the practice

- The source files delivered must compile without any change. In other case they will not be considered and thus graded with 0 points.
- The student wil NOT modify the headers (including names and types of parameters) of the existing methods or the return values of those.
- The calls to the executable of the practice must be respected. It is not allowed to map, in the code, names of input files.
- The following source files must be handed in through Aula Global (changing ".java" extension to ".txt" so that Turnitin will accept them):

- grado.sti.auxiliar.Datos.java
- grado.sti.hash.FuncionResumenEncadenada.java
- grado.sti.hash.FuncionResumenXORBasica.java
- grado.sti.hash.FuncionResumenXORDesplazamiento.java
- grado.sti.hash.test.PruebaColisiones.java

- It is not recommended the use of accents for the names of variables. Some O.S. do not import such names properly in case of accents, thus creating non-compilable code (which will not be considered).
- It will be negatively considered that the code shows information not requested (for example, console output that the student uses for checking the code).