

DATA PROTECTION

Lab work 2: Hash functions and MAC forgery attacks

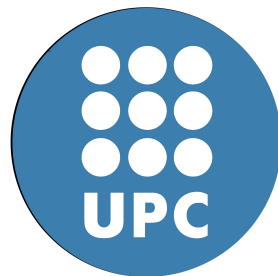
Students

Alejandro Capella del Solar

Cristian Fernández Jiménez

Professor

Jorge Luis Villar Santos



November 15, 2022

Contents

List of Figures	2
1 Recreation of known MAC forgery attacks	3
1.1 CBC-MAC concatenation attack	3
1.2 One-pass HMAC length extension attack	6
1.2.1 Length extension attack	7
1.2.1.1 Adding a padding	7
1.2.1.2 Forge a message	9
1.2.2 Attack verification	11
2 Building Merkle hash trees	12
2.1 Our first merkle tree	12
2.2 Adding files to our merkle tree	13
2.3 Let's build a proof	15
2.4 Let's build a verifier	16
A	19

List of Figures

1.1	Creation of random key, message files and 16 bytes null header. . . .	3
1.2	Tag generation for the two messages (header included).	4
1.3	Binary representation of message 1.	4
1.4	Padded message by adding missing bytes until end of block.	5
1.5	Forged message creation and inspection, and tag generation and verification.	6
1.6	Padded message by adding an extra necessary block.	8
1.7	Padded message by finishing last block.	8
1.8	Complete attack sequence and tag verification.	11
2.1	Content of the hashtree.txt file containing the Merkle tree.	13
2.2	Printing a 16 nodes merkle tree.	15
2.3	Proof for the included 4th node waiting to be verified	16
2.4	The verifier says OK.	17
2.5	Proof for the not included 3rd doc waiting to be verified.	17
2.6	The verifier says KO for the 3rd document in fifth position.	18

Chapter 1

Recreation of known MAC forgery attacks

1.1 CBC-MAC concatenation attack

CBC-MAC is insecure if used with variable length messages, we will try then to perform a MAC forgery attack against it. We have recreated the attack following these steps:

1. Create a random AES-128 key (random string of 16 bytes), and choose two arbitrary messages. We used the ones suggested and saved them in two files. Also we generated a random key and a 16 bytes null header, in order to append it to the messages:

```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ openssl rand -hex 16 > hexkey.dat
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ cat mess1.dat
"What about joining me tomorrow for dinner?"
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ cat mess2.dat
Oops, Sorry, I just remember that I have a meeting very soon in the morning.
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ truncate -s 16 nullbytes > head.dat
```

Figure 1.1: Creation of random key, message files and 16 bytes null header.

2. For simplicity, we assumed that the system adds a header to the messages consisting of 16 zero bytes, in order to create the tag of each message. Then, previously to CBC-MAC generation, was necessary to create two files with both header and message.

We stucked both messages with the header with the simple following commands:

```
1 cat head.dat mess1.dat > headmess1.dat
```

```
2 cat head.dat mess2.dat > headmess2.dat
```

Listing 1.1: Concatenating messages with headers.

3. Next up, generating the corresponding AES-128-CBC-MACs for the two messages with headers and store them in the files tag1.dat and tag2.dat.

```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ openssl enc -aes-128-cbc -K 'cat hexkey.dat' -iv 0 -in headmess1.dat | tail -c 16 > tag1.dat
hex string is too short, padding with zero bytes to length
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ openssl enc -aes-128-cbc -K 'cat hexkey.dat' -iv 0 -in headmess2.dat | tail -c 16 > tag2.dat
hex string is too short, padding with zero bytes to length
```

Figure 1.2: Tag generation for the two messages (header included).

4. In order to investigate the padding that AES-128-CBC introduces in the last incomplete block, we encrypted a message with AES-128-CBC, and then decrypted the result with the option `-nopad`, recovering the padded version of the first message (that will be used to generate the forgery). The encryption/decryption lines to investigate the padding would be something like

```
1 openssl enc -aes-128-cbc -K $key -iv 0 -in message.dat -out
  cipher.dat
2
3 openssl enc -d -aes-128-cbc -K $key -iv 0 -nopad -in cipher.dat
  -out padded.dat xxd padded.dat
```

If we inspect the file now with `xxd`, we can check how it behaves:

```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ xxd padded.dat
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ....
00000010: e280 9c57 6861 7420 6162 6f75 7420 6a6f  ...What about jo
00000020: 696e 696e 6720 6d65 2074 6f6d 6f72 726f  ining me tomorro
00000030: 7720 666f 7220 6469 6e6e 6572 3fe2 809d  w for dinner?...
00000040: 0a0f 0f0f 0f0f 0f0f 0f0f 0f0f 0f0f 0f0f  ....
```

Figure 1.3: Binary representation of message 1.

As we see, the last block incomplete is terminated with the value of the bytes missing, in this case 15 (0f). Which is, if there are n missing bytes in the block, then the padding is n bytes all with the value n .

In a sort of practical attack, and following the previous instructions, we also came up with a script to generate a padded message, without having the key: We decrypt with the option `nopad` set up.

```
1 #! /usr/bin/bash
2
3 if [ "$#" -ne "1" ]
4 then
```

```

5
6 echo 'Introduce file with message to pad'
7 echo "$0 file"
8 exit 1
9
10 else
11
12 echo "Getting message padded from $1 (into padded.dat)"
13 total_bytes='wc -c $1 | cut -f1 -d" "'
14 padding=$(( 16 - ($total_bytes%16) ))
15
16 if [[ $padding -eq 0 ]] # FACT 1, IF IV=01FFxx
17 then
18 padding=16
19 fi
20
21 cat $1 > padded.dat
22
23 for (( i=0; i<$padding; i++ ))
24 do
25 printf "\x$(printf "%02x" "$padding)" >> padded.dat
26 done
27 fi

```

Listing 1.2: Bash script to get AES-128-CBC padding from a message file.

```

[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ bash get_padding.sh mess2.dat
Getting message padded from mess2.dat (into padded.dat)
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ xxd padded.dat
00000000: 4f6f 7073 2c20 536f 7272 792c 2049 206a  Oops, Sorry, I j
00000010: 7573 7420 7265 6d65 6d62 6572 2074 6861  ust remember tha
00000020: 7420 4920 6861 7665 2061 206d 6565 7469  t I have a meeti
00000030: 6e67 2076 6572 7920 736f 6f6e 2069 6e20  ng very soon in
00000040: 7468 6520 6d6f 726e 696e 672e 0a03 0303  the morning....

```

Figure 1.4: Padded message by adding missing bytes until end of block.

As we perceive, 3 bytes are appended, as those are the missing ones to complete the block.

5. The last step would be create a forgery message, from the known messages and their tags. In this case, we would concatenate as follows:

$$HEADER + MESS1 + PADDING + TAG1 + MESS2$$

The creation of the forgery and its tags are shown in these commands, in addition with *diff* we verify there are no differences between the actual tag and the forged one. With these steps we have proved that is possible to come with a forged message/tag pair using CBC-MAC.

```
[Cristian Fernández Jinénez & Alejandro Capella del Solar:]$ cat padded.dat tag1.dat mess2.dat > forgery.dat
[Cristian Fernández Jinénez & Alejandro Capella del Solar:]$ xxd forgery.dat
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ....
00000010: e289 9c57 6861 7420 6162 6f75 7420 6a6f  ...What about jo
00000020: 696e 696e 6720 6d65 2074 6f6d 6f72 726f  ining me tonorro
00000030: 7720 666f 7220 6469 6e6e 6572 3fe2 808d  w for dinner?...
00000040: 0a0f 0f0f 0f0f 0f0f 0f0f 0f0f 0f0f 0f0f  ....
00000050: 7c5c a895 bf38 33bb 2062 dd9f d0fb 09e9  |\...B3. b.....
00000060: 4f6f 7073 2c20 536f 7272 792c 2049 206a  Oops, Sorry, I j
00000070: 7573 7420 7265 6d65 6d62 6572 2074 6861  ust remember tha
00000080: 7420 4920 6861 7665 2061 206d 6565 7469  t I have a meeti
00000090: 6a67 2076 6572 7920 736f 6f6e 2069 6e20  ng very soon in
000000a0: 7469 6520 6d6f 726e 696e 672e 0a      the morning..
[Cristian Fernández Jinénez & Alejandro Capella del Solar:]$ openssl enc -aes-128-cbc -K 'cat hexkey.dat' -iv 0 -in forgery.dat | tail -c 16 > forgertag.dat
hex string is too short, padding with zero bytes to length
[Cristian Fernández Jinénez & Alejandro Capella del Solar:]$ diff tag2.dat forgertag.dat
```

Figure 1.5: Forged message creation and inspection, and tag generation and verification.

1.2 One-pass HMAC length extension attack

In a similar way, we can forge a pass message without knowing the key. Knowing a message and its corresponding tag, an attacker can perform a length extension procedure with the only limitation that the concatenated string must start with a particular padding to the first message.

The structure that we are going to follow on this exercise is the following, in order to simulate an actual attack:

Preparation phase: Simulation of the victim, who would generate a message and a random 16-byte key. Then, obtain the digest of their concatenation, by using the MD5 option of OpenSSL.

```
1 cat key.dat message.dat | openssl dgst -md5 -binary > tag.dat
```

We assume this done and does not require any further methods.

Attack phase: We start by knowing message and tag. The goal is to concatenate a message and receive a valid tag that makes it authentic. The procedure will consist of a padding generation of the initial message to *glue* the second one and make MD5 to generate a valid tag. This will be achieved by tweaking the code in the OpenSSL library, so we can start a concatenation of data from a valid state (end of the initial padded message).

Validation phase: Once the tag has been generated, we may be able to validate it by comparing the one that we could generate if we knew the key. The forged tag and the legit one must be equal.

1.2.1 Length extension attack

As it was said before, the message must be padded if we plan to *glue* an additional one. Then the tag would be processed only if the forgery is correct and the state is initialized correctly.

1.2.1.1 Adding a padding

For the MD5 hash function, there is a block size of 512 bits. Padding takes the form `0x80 0x00 ... 0x00 n0 n1 ... n7`, where the last 8 bytes encode the length in bits of the unpadded message (little-endian) and the 0s are optional. This means that padding requires 65 mandatory bits, or as we have assumed, 72 bits (bytes will not be split as we are using plain text messages).

We have implemented a bash script to achieve it. This script will accept the message (without the key)¹ and will generate the necessary padding depending on its size.

```

1  if [[ $lastblock_bits -eq 0 || $lastblock_bits -le $(( $block_size
   - $mandatory_bits )) ]] #LAST BLOCK IS COMPLETED (64 BYTES) or
   FITS MANDATORY PADDING (9 BYTES)
2  then
3
4      necessary_zeros=$(( $bytes_available - $mandatory_bytes ))
5
6      printf "\x$(printf "%02x" "128")" >> padded.dat #APPEND 0x80
7      for run in $( seq 1 $necessary_zeros ); do printf "\\x$(printf "%
   x" "0")"; done >> padded.dat #APPEND NECESSARY 0x00
8      echo 00: 'printf "%016x" $total_bits' | xxd -r | xxd -g 8 -e |
   xxd -r >> padded.dat #APPEND SIZE in LITTLE-ENDIAN (8 BYTES)
9
10  elif [[ $lastblock_bits -gt $(( $block_size - $mandatory_bits )) &&
   $lastblock_bits -lt $block_size ]] #LAST BLOCK DOESN'T FIT
   MANDATORY PADDING
11  then
12
13      necessary_zeros=$(( $bytes_available - 1 ))
14
15      printf "\x$(printf "%02x" "128")" >> padded.dat #APPEND 0x80
16      for run in $( seq 1 $necessary_zeros ); do printf "\\x$(printf "%
   x" "0")"; done >> padded.dat #APPEND NECESSARY 0x00 TO FINISH
   BLOCK
17
18  #—BLOCK FINISHED—

```

¹**Important:** Note that 16 bytes are missing, as we don't know the key. This is done mainly to act fully as an attacker, the bits will be counted back in the forgery script.


```

19   for run in {1..56}; do printf "\\x$(printf "%x" "0"); done >>
20   padded.dat #APPEND 56 BYTES OF 0x00 TO FINISH BLOCK
21   echo 00: 'printf "%016x" $total_bits' | xxd -r | xxd -g 8 -e |
    xxd -r >> padded.dat #APPEND SIZE in LITTLE-ENDIAN (8 BYTES)
22
23
24   fi

```

Listing 1.3: Script building the message padding.

```

[CristianFdez J & Alejandro J Capella:]$ cat mess1.dat
"What about joining me tomorrow for dinner?"
[CristianFdez J & Alejandro J Capella:]$ xxd mess1.dat
00000000: e280 9c57 6861 7420 6162 6f75 7420 6a6f  ...What about jo
00000010: 696e 696e 6720 6d65 2074 6f6d 6f72 726f  ining me tomorro
00000020: 7720 666f 7220 6469 6e6e 6572 3f22 0a    w for dinner?".
[CristianFdez J & Alejandro J Capella:]$ bash get_padding.sh mess1.dat
Getting message padded from mess1.dat (into padded.dat)
[CristianFdez J & Alejandro J Capella:]$ xxd padded.dat
00000000: e280 9c57 6861 7420 6162 6f75 7420 6a6f  ...What about jo
00000010: 696e 696e 6720 6d65 2074 6f6d 6f72 726f  ining me tomorro
00000020: 7720 666f 7220 6469 6e6e 6572 3f22 0a80  w for dinner?"..
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000060: 0000 0000 0000 0000 f801 0000 0000 0000  .....

```

Figure 1.6: Padded message by adding an extra necessary block.

```

[CristianFdez J & Alejandro J Capella:]$ cat mess2.dat
"What about joining me?"
[CristianFdez J & Alejandro J Capella:]$ xxd mess2.dat
00000000: 2257 6861 7420 6162 6f75 7420 6a6f 696e  "What about join
00000010: 696e 6720 6d65 3f22 0a                      ing me?".
[CristianFdez J & Alejandro J Capella:]$ bash get_padding.sh mess2.dat
Getting message padded from mess2.dat (into padded.dat)
[CristianFdez J & Alejandro J Capella:]$ xxd padded.dat
00000000: 2257 6861 7420 6162 6f75 7420 6a6f 696e  "What about join
00000010: 696e 6720 6d65 3f22 0a80 0000 0000 0000  ing me?".....
00000020: 0000 0000 0000 0000 4801 0000 0000 0000  .....H.....

```

Figure 1.7: Padded message by finishing last block.

1.2.1.2 Forge a message

To come with the length extension attack, it's not as trivial as the previous attack, as we need to tweak the code in the OpenSSL library in order to establish a valid current state to append the new data.

The computation of a hash function in OpenSSL consists of three steps: initialize, update and finalize. All the internal state of the computation is placed into a context object. Seen this, we need to implement an intermediate step, for the purpose of initialize the context with on the *glueing* point.

The following function is intended to be called just after `MD5_Init(MD5_CTX *)`:

```
void set_ctx(MD5_CTX *pctx, const char *digest, unsigned long nblocks) {
    pctx->A = gethexword32(digest);
    pctx->B = gethexword32(digest+4);
    pctx->C = gethexword32(digest+8);
    pctx->D = gethexword32(digest+12);
    nblocks <= 9; // converting into bits
    pctx->Nh = nblocks>>32;
    pctx->Nl = nblocks&0xFFFFFFFFul;
}
```

Note that shifting is being reduced to the half, as digest is read as plain text from a file. In this way we read the 16 bytes properly.

In addition, `MD5_LONG gethexword32(const char *digest)` needs to be implemented. It reads a 4 byte integer from its hexadecimal representation, interpreting the byted in little-endian order.

```
1 MD5_LONG gethexword32(const unsigned char *digest){
2
3     MD5_LONG var =(unsigned) digest[0]|((unsigned) digest[1]<<8)|((
    unsigned) digest[2]<<16)|((unsigned) digest[3]<<24); //Little endian
    to Int
4
5     return var;
6 }
```

From this point, it is straightforward to reconstruct the last internal state of the computation and obtain a forged tag.

We propose the following function, which accepts a padded message file, its tag and the new data to append and returns a valid new tag:

```
unsigned char *file2md5(const char *filename, const char *digest_file,
    const char *newdata_file)
```

Inside this function, we are going to follow the same procedure² that OpenSSL would do to generate the digest, but with an intermediate point to establish the wanted state.

```

1 unsigned char *file2md5(const char *filename, const char *digest_file
  , const char * newdata_file) {
2     char * msg = 0;
3     long length;
4     long total_bytes;
5     unsigned long nblocks;
6
7     \\... Read message file into msg
8
9     total_bytes=length+16; //Bytes of padded message + key size
10    nblocks=total_bytes/64;
11    MD5_CTX c;
12    unsigned char * digest=0;
13
14    \\... Read tag file into digest
15
16    MD5_Init(&c);
17
18    set_ctx(&c,digest,nblocks); //Set context according to padded
    message and passing the tag
19
20    char * new_data=0;
21
22    \\... Read new message file into new_data
23
24    //Call update for each new block of data
25    while (length > 0) {
26
27        if (length > 64) {
28            MD5_Update(&c, new_data, 64);
29        } else {
30            MD5_Update(&c, new_data, length);
31        }
32        length -= 64;
33        new_data += 64; //Shift to next block of data
34    }
35
36    unsigned char new_digest[MD5_DIGEST_LENGTH];
37
38    MD5_Final(new_digest, &c); //Get the new forged tag
39
40    \\... Return new digest

```

²Note that 16 bytes are counted back, as we omitted them in the padding gen.

41

}

1.2.2 Attack verification

Once we have achieved to forge a tag for an extended message, we must put it alongside a legitimate one, that we would generate using the real key.

```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ cat key.dat mess1.d
at | openssl dgst -md5 -binary > tag.dat
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ bash get_padding.sh
mess1.dat
Getting message padded from mess1.dat (into padded.dat)
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ xxd padded.dat
00000000: e280 9c57 6861 7420 6162 6f75 7420 6a6f  ...What about jo
00000010: 696e 696e 6720 6d65 2074 6f6d 6f72 726f  ining me tomorro
00000020: 7720 666f 7220 6469 6e6e 6572 3f22 0a80  w for dinner?"..
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000060: 0000 0000 0000 0000 f801 0000 0000 0000  .....
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ xxd new_data.dat
00000000: 4c65 7427 7320 6d65 6574 2064 6f77 6e74  Let's meet downt
00000010: 6f6e 7720 6174 2031 393a 3030 2c20 696e  onw at 19:00, in
00000020: 2066 726f 6e74 206f 6620 7468 6520 6361  front of the ca
00000030: 7468 6564 7261 6c2e 0a          thedral..
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ ./attack padded.dat
tag.dat new_data.dat
FORGER TAG: a5c367e4c3f5601ff0bfd539add6c3b0
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ cat key.dat padded.
dat new_data.dat | openssl dgst -md5 -binary > forger_tag.dat
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ xxd forger_tag.dat
00000000: a5c3 67e4 c3f5 601f f0bf d539 add6 c3b0  ..g...`...9....
```

Figure 1.8: Complete attack sequence and tag verification.

We created a message file `mess1.dat` and hashed it on `tag.dat`. As attackers, calling the script to get the padding would only need the message. Once received, we generated a new message to forge. With these files, padded message, tag and new message we are able to call the attack script, getting a brand new tag.

We may validate this by generating a digest of the concatenation of key, padded initial message and new message. And as we can see, `forger_tag.dat` contains the same digest as the one generated by our script.

With these steps we have proved that is possible to come with a forged message/tag pair using HMAC.

Chapter 2

Building Merkle hash trees

Merkle Trees are binary trees used to save valid identifiers of a group of files. In this way, every file is identified by a hash and every parent node is a contribution from its leaves. There will exist $\log_2(n)$ levels for n leaves (docs). Furthermore, a proof will be of size $\log_2(n)$ as well. Therefore, to forge a proof, an attacker needs to find a collision of the hash function.

In this chapter, we will operate with them, implementing a creation of a Merkle tree hash, the adding of files, verification of files against an specific Merkle hash tree and, at last, generating proofs of membership.

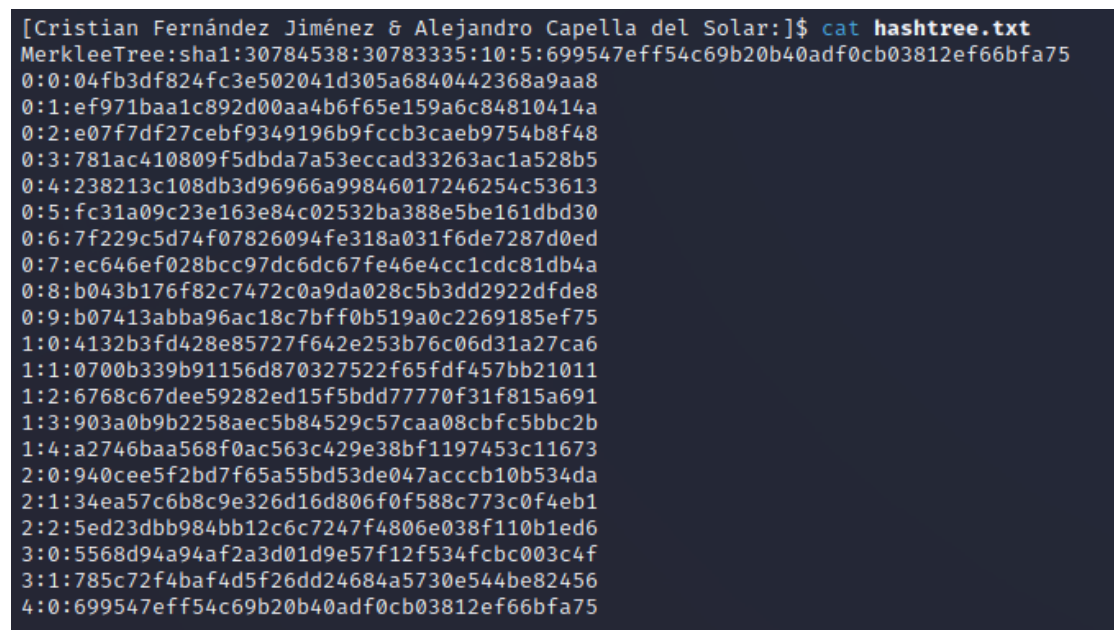
2.1 Our first merkle tree

First of all we have to build the merkle hash tree. We have decided to build a merkle hash tree made out of 10 documents, each of them filled with a simple word and an iterator. We have followed these steps to accomplish the task all of them coded in the file `merkle.sh`:

- We write all the headers of the file that the output produces and we leave the keyword `linux` as the final hash. We will use this mark later.
- We have build a loop with an `echo` and a pipe creating the ten files with the word `'hello'` and an iterator so every file has different content.
- Now we have to create the leaves from each document. We take the prefix and execute a `cat` along with the cat of each document and create a digest in hex format and we store it in the files `./nodes/node0:i`

- We set up a double loop of four levels in the outside loop and 10 positions in the inner loop. At each loop we check whether exist the $2 \times j$ position and the $2 \times j + 1$, or only the $2 \times j$ position. In the first case we concatenate the content of both nodes with the prefix, in the second case only the node in the $2 \times j$ position.
- At every loop we add a line in the resulting text file with the position i and j of the node and the corresponding hash just like the exercise statement says.
- When the double loop is over, we use the `sed` command to substitute the word `linux` by the resulting hash of the Merkle tree.

In figure 2.1 we print the hash tree file content, produced by the script called *merkle.sh* that builds a merkle tree of 10 documents.



```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:]$ cat hashtree.txt
MerkleeTree:sha1:30784538:30783335:10:5:699547eff54c69b20b40adf0cb03812ef66bfa75
0:0:04fb3df824fc3e502041d305a6840442368a9aa8
0:1:ef971baa1c892d00aa4b6f65e159a6c84810414a
0:2:e07f7df27cebf9349196b9fccb3caeb9754b8f48
0:3:781ac410809f5dbda7a53eccad33263ac1a528b5
0:4:238213c108db3d96966a99846017246254c53613
0:5:fc31a09c23e163e84c02532ba388e5be161dbd30
0:6:7f229c5d74f07826094fe318a031f6de7287d0ed
0:7:ec646ef028bcc97dc6dc67fe46e4cc1cdc81db4a
0:8:b043b176f82c7472c0a9da028c5b3dd2922dfde8
0:9:b07413abba96ac18c7bff0b519a0c2269185ef75
1:0:4132b3fd428e85727f642e253b76c06d31a27ca6
1:1:0700b339b91156d870327522f65fdf457bb21011
1:2:6768c67dee59282ed15f5bdd77770f31f815a691
1:3:903a0b9b2258aec5b84529c57caa08cbfc5bbc2b
1:4:a2746baa568f0ac563c429e38bf1197453c11673
2:0:940cee5f2bd7f65a55bd53de047acccb10b534da
2:1:34ea57c6b8c9e326d16d806f0f588c773c0f4eb1
2:2:5ed23dbb984bb12c6c7247f4806e038f110b1ed6
3:0:5568d94a94af2a3d01d9e57f12f534fcbc003c4f
3:1:785c72f4baf4d5f26dd24684a5730e544be82456
4:0:699547eff54c69b20b40adf0cb03812ef66bfa75
```

Figure 2.1: Content of the hashtree.txt file containing the Merkle tree.

2.2 Adding files to our merkle tree

Now we have to add nodes to our preexistent merkle tree. For that goal we have built a little program that receives as a parameter the number of nodes you want to add to our merkle tree and rebuilds the whole tree updating the file `hashtree.txt`. We begin with the header and then we perform a little calculus

for the height of the resulting tree with a little loop dividing by two the total number of leaves like this:

```
1 height=0
2 while [ $temp -gt 0 ]
3 do
4     temp=$(( $temp/$division ))
5     ((height+=1))
6 done
```

Listing 2.1: Calculating the height of the tree, temp is number of leaves.

The code only works if the `merkle.sh` script has been executed before because this code only adds the extra documents, but all the previous documents must have been created with the initial script. We must point out that there is a special check before this condition. If the number of leaves has an integer square root we subtract 1 to the number of elements.

```
1 d=$(echo "sqrt($temp)" | bc)
2 if [[ $d =~ ^[0-9] ]]
3 then
4     ((temp-=1))
5 fi
6
7
```

Listing 2.2: Calculating square root of number of leaves.

Notice that the `bc` command must be installed (`sudo apt-get install bc`). This code is always taking into account that the initial tree has always 10 elements, which is the merkle tree created by the previous script. We could have modified this only counting the number of lines from the `hashtree.txt` beginning with 0: with the `grep -c` command, but initially we code it like this and we left it like that but counting the leaves and rebuilding the whole tree wouldn't be any harder. In figure 2.2 we can see a tree where we have added 16 nodes to the previous merkle tree.

```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:] $cat hashtree.txt
MerkleeTree:sha1:30784538:30783335:16:5:ed5e8e67b0999fc114e0738104aaa8dbaa9a6ec4
0:0:04fb3df824fc3e502041d305a6840442368a9aa8
0:1:ef971baa1c892d00aa4b6f65e159a6c84810414a
0:2:e07f7df27cebf9349196b9fccb3cae9754b8f48
0:3:781ac410809f5dbda7a53eccad33263ac1a528b5
0:4:238213c108db3d96966a99846017246254c53613
0:5:fc31a09c23e163e84c02532ba388e5be161dbd30
0:6:7f229c5d74f07826094fe318a031f6de7287d0ed
0:7:ec646ef028bcc97dc6dc67fe46e4cc1cdc81db4a
0:8:b043b176f82c7472c0a9da028c5b3dd2922dfde8
0:9:b07413abba96ac18c7bff0b519a0c2269185ef75
0:10:352f00cd185bcea0b25d1a788fb339b0be7cf03e
0:11:dba0d135584d6859b610186d49d42571bc7cf951
0:12:d19a142a301be3a5f1b4275df759a3d0837ce1ac
0:13:a0a1907f24c1966b48525dd1f95813bd0d1ae75c
0:14:2763ca011418390fcebcbf77aed2cf9a37b52dfc2
0:15:bc34b10e4c2f479578cf185c802447deb58f5d77
1:0: 4132b3fd428e85727f642e253b76c06d31a27ca6
1:1: 0700b339b91156d870327522f65fdf457bb21011
1:2: 6768c67dee59282ed15f5bdd77770f31f815a691
1:3: 903a0b9b2258aec5b84529c57caa08cbfc5bbc2b
1:4: a2746baa568f0ac563c429e38bf1197453c11673
1:5: 7b9a249c4411aa683e32b87d455ed729258ad378
1:6: f3a86b2eef01072ca6fcfc6fa703ed67c68798341
1:7: 7776e27a4b031ab69a13867c999bf32a061d2d89
2:0: 940cee5f2bd7f65a55bd53de047acccb10b534da
2:1: 34ea57c6b8c9e326d16d806f0f588c773c0f4eb1
2:2: 20f8e98878a3089f634dea2c93065a005220a7bc
2:3: c41569f82b16578b1e2f105dbb1a91c33c0275e3
3:0: 5568d94a94af2a3d01d9e57f12f534fcbcb003c4f
3:1: f7af44bb27b08cf2a49d4b9d09f5bea7ebcd4592
4:0: ed5e8e67b0999fc114e0738104aaa8dbaa9a6ec4
```

Figure 2.2: Printing a 16 nodes merkle tree.

2.3 Let's build a proof

This new script will come with a proof of a given node, so that it's possible to verify whether the element is on the tree or not. There are some previous necessary steps before execution. It is necessary to have created the tree with the `merkle.sh` script before or added nodes by executing `addnode.sh`. Proof will take the hash of the original tree and place it in the header of the `proof.txt` file. Take into account as well that they need to have the same number of leaves. As first parameter, the document to verify is requested. The second parameter is the position of the leaf, and the third will require the total number of documents. The script is very simple, a single loop creating all the leaves. For the output example we have created a proof for 16 nodes: [2.3](#)


```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:] $cat proof.txt
MerkleeTree:sha1:30784538:30783335:16:5:ed5e8e67b0999fc114e0738104aaa8dbaa9a6ec4
0:0:04fb3df824fc3e502041d305a6840442368a9aa8
0:1:ef971baa1c892d00aa4b6f65e159a6c84810414a
0:2:e07f7df27ceb9f349196b9fccb3caeb9754b8f48
0:3:781ac410809f5dbda7a53eccad33263ac1a528b5
0:4:238213c108db3d96966a99846017246254c53613
0:5:fc31a09c23e163e84c02532ba388e5be161dbd30
0:6:7f229c5d74f07826094fe318a031f6de7287d0ed
0:7:ec646ef028bcc97dc6dc67fe46e4cc1cdc81db4a
0:8:b043b176f82c7472c0a9da028c5b3dd2922dfde8
0:9:b07413abba96ac18c7bfff0b519a0c2269185ef75
0:10:352f00cd185bcea0b25d1a788fb339b0be7cf03e
0:11:dba0d135584d6859b610186d49d42571bc7cf951
0:12:d19a142a301be3a5f1b4275df759a3d0837ce1ac
0:13:a0a1907f24c1966b48525dd1f95813bd0d1ae75c
0:14:2763ca011418390fcebcbf77aed2cf9a37b52dfc2
0:15:bc34b10e4c2f479578cf185c802447deb58f5d77
```

Figure 2.3: Proof for the included 4th node waiting to be verified

2.4 Let's build a verifier

Given the previous file now let's check whether it is included or not in the tree. This verifier reads the file called `proof.txt` in the same directory and builds the remaining tree. Comparing the result with the previous hash will tell if it's an included node of the tree. The algorithm is pretty similar, it considers both cases of having both type of nodes, the $2 \times j$ and the $2 \times j + 1$, and builds the tree. Instead of picking up the nodes from the disk, it's done from the file.

```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:] $cat verify.txt
MerkleeTree:sha1:30784538:30783335:16:5:ed5e8e67b0999fc114e0738104aaa8dbaa9a6ec4
0:0:04fb3df824fc3e502041d305a6840442368a9aa8
0:1:ef971baa1c892d00aa4b6f65e159a6c84810414a
0:2:e07f7df27ceb9349196b9f9cc3caeb9754b8f48
0:3:781ac410809f5dbda7a53eccad33263ac1a528b5
0:4:238213c108db3d96966a99846017246254c53613
0:5:fc31a09c23e163e84c02532ba388e5be161dbd30
0:6:7f229c5d74f07826094fe318a031f6de7287d0ed
0:7:ec646ef028bcc97dc6dc67fe46e4cc1cdc81db4a
0:8:b043b176f82c7472c0a9da028c5b3dd2922dfde8
0:9:b07413abba96ac18c7bfb0b519a0c2269185ef75
0:10:352f00cd185bcea0b25d1a788fb339b0be7cf03e
0:11:dba0d135584d6859b610186d49d42571bc7cf951
0:12:d19a142a301be3a5f1b4275df759a3d0837ce1ac
0:13:a0a1907f24c1966b48525dd1f95813bd0d1ae75c
0:14:2763ca011418390fcebfcf77aed2cf9a37b52dfc2
0:15:bc34b10e4c2f479578cf185c802447deb58f5d77
1:0:4132b3fd428e85727f642e253b76c06d31a27ca6
1:1:0700b339b91156d870327522f65fdf457bb21011
1:2:6768c67dee59282ed15f5bdd77770f31f815a691
1:3:903a0b9b2258aec5b84529c57caa08cbfc5bbcb2b
1:4:a2746baa568f0ac563c429e38bf1197453c11673
1:5:7b9a249c4411aa683e32b87d455ed729258ad378
1:6:f3a86b2ee0f01072ca6fcf6fa703ed67c68798341
1:7:7776e27a4b031ab69a13867c999bf32a061d2d89
2:0:940cee5f2bd7f65a55bd53de047accb10b534da
2:1:34ea57c6b8c9e326d16d806f0f588c773c0f4eb1
2:2:20f8e98878a3089f634dea2c93065a005220a7bc
2:3:c41569f82b16578b1e2f105dbb1a91c33c0275e3
3:0:5568d94a94af2a3d01d9e57f12f534fcbb003c4f
3:1:f7af44bb27b08cf2a49d4b9d09f5bea7ebcd4592
4:0:ed5e8e67b0999fc114e0738104aaa8dbaa9a6ec4
Verification ok
```

Figure 2.4: The verifier says OK.

We see that everything went OK from the picture in figure 2.4.

Now let's test the same thing but for a file in other position, a wrong one. First let's create a different proof. We will give doc3 to the program but in position 5. Let's watch at the different proof at figure 2.5

```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:] $bash proof.sh 5 ./docs/doc3.dat 16
[Cristian Fernández Jiménez & Alejandro Capella del Solar:] $cat proof.txt
MerkleeTree:sha1:30784538:30783335:16:5:ed5e8e67b0999fc114e0738104aaa8dbaa9a6ec4
0:0:04fb3df824fc3e502041d305a6840442368a9aa8
0:1:ef971baa1c892d00aa4b6f65e159a6c84810414a
0:2:e07f7df27ceb9349196b9f9cc3caeb9754b8f48
0:3:781ac410809f5dbda7a53eccad33263ac1a528b5
0:4:238213c108db3d96966a99846017246254c53613
0:5:781ac410809f5dbda7a53eccad33263ac1a528b5
0:6:7f229c5d74f07826094fe318a031f6de7287d0ed
0:7:ec646ef028bcc97dc6dc67fe46e4cc1cdc81db4a
0:8:b043b176f82c7472c0a9da028c5b3dd2922dfde8
0:9:b07413abba96ac18c7bfb0b519a0c2269185ef75
0:10:352f00cd185bcea0b25d1a788fb339b0be7cf03e
0:11:dba0d135584d6859b610186d49d42571bc7cf951
0:12:d19a142a301be3a5f1b4275df759a3d0837ce1ac
0:13:a0a1907f24c1966b48525dd1f95813bd0d1ae75c
0:14:2763ca011418390fcebfcf77aed2cf9a37b52dfc2
0:15:bc34b10e4c2f479578cf185c802447deb58f5d77
```

Figure 2.5: Proof for the not included 3rd doc waiting to be verified.

Now if everything is OK, the verification should fail. Let's execute the verifier 2.6:

```
[Cristian Fernández Jiménez & Alejandro Capella del Solar:] $cat verify.txt
MerkleeTree:sha1:30784538:30783335:16:5:0b84083c6309e887f1bd3e351c04497f30054c73
0:0:04fb3df824fc3e502041d305a6840442368a9aa8
0:1:ef971baa1c892d00aa4b6f65e159a6c84810414a
0:2:e07f7df27ceb9f349196b9fccb3caeb9754b8f48
0:3:781ac410809f5dbda7a53eccad33263ac1a528b5
0:4:238213c108db3d96966a99846017246254c53613
0:5:781ac410809f5dbda7a53eccad33263ac1a528b5
0:6:7f229c5d74f07826094fe318a031f6de7287d0ed
0:7:ec646ef028bcc97dc6dc67fe46e4cc1cdc81db4a
0:8:b043b176f82c7472c0a9da028c5b3dd2922dfde8
0:9:b07413abba96ac18c7bfff0b519a0c2269185ef75
0:10:352f00cd185bcea0b25d1a788fb339b0be7cf03e
0:11:dba0d135584d6859b610186d49d42571bc7cf951
0:12:d19a142a301be3a5f1b4275df759a3d0837ce1ac
0:13:a0a1907f24c1966b48525dd1f95813bd0d1ae75c
0:14:2763ca011418390fcebfcf77aed2cf9a37b52dfc2
0:15:bc34b10e4c2f479578cf185c802447deb58f5d77
1:0:4132b3fd428e85727f642e253b76c06d31a27ca6
1:1:0700b339b91156d870327522f65fdf457bb21011
1:2:d72a0116234a7f6adf53c406716dc69c51c6090b
1:3:903a0b9b2258aec5b84529c57caa08cbfc5bbcb2b
1:4:a2746baa568f0ac563c429e38bf1197453c11673
1:5:7b9a249c4411aa683e32b87d455ed729258ad378
1:6:f3a86b2eef01072ca6fcf6fa703ed67c68798341
1:7:7776e27a4b031ab69a13867c999bf32a061d2d89
2:0:940cee5f2bd7f65a55bd53de047acccb10b534da
2:1:e420d1301127f736ca5e7d8cfa8cf58ffcb10a18
2:2:20f8e98878a3089f634dea2c93065a005220a7bc
2:3:c41569f82b16578b1e2f105dbb1a91c33c0275e3
3:0:aca79e7df1ede2cf34b3f1c827a176f7bb76d569
3:1:f7af44bb27b08cf2a49d4b9d09f5bea7ebcd4592
4:0:0b84083c6309e887f1bd3e351c04497f30054c73
Verification ko
```

Figure 2.6: The verifier says KO for the 3rd document in fifth position.

As we see the textfile `verify.txt` indicates that the node doesn't belong to that position.

Important: Note there is a script called `restart.sh` that will delete all nodes and documents. Remaining documents or nodes in the local environment from previous executions may interfere with correct working of the scripts, be careful.

Appendix A

A github repository with all scripts is available [online](#). Everything is inside LAB2 folder. If necessary, please follow the execution instructions given in this document. Execute the sections in order, in case of the Merkle Tree exercise.