POLITECNICO DI TORINO

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA

OPERATING SYSTEMS

Final Project - Multi-Blowfish

Busignani Fabio 197883
Capoccia Raffaele 198434

July 13, 2014

Abstract

This is a report edited by the students Busignani and Capoccia where the behaviour and results of a multi-thread implementation for *Blowfish* algorithm are shown. The report is divided into four parts. The first one describes the Blowfish algorithm. The second one illustrates the design choices for implementing multi-threads.The third part illustrates the different blocks in which our code is divided. The last part shows the results of test bench that runs on a Linux machine which has kernel version 3.13.0-24 and *Intel(R) Core(TM) i7 CPU Q 740 @ 1.73GHz*.

CONTENTS

INTRODUCTION

In this project a program able to encrypt and decrypt a file using a multi-threaded implementation of the Blowfish algorithm is implemented.

In this document particular attention is dedicated to the analysis of the performance varying the number of threads.

The first chapter regards a detailed description of the algorithm, while the second chapter focuses on data handling; in the third part the implementation, without presenting the entire code, is fully described and, finally, a performance description is given using different text size and number of threads.

As indicated in the project description document, parameters specified by user are:

- Information about operation to do: encryption or decryption
- Input file name
- Cryptographic key
- Output file name
- Number of threads

1 BLOWFISH ALGORITHM

Blowfish is a symmetric-key block cipher, designed in 1993 by Bruce Schneier. This algorithm is designed for cryptography and it has the particularity of being not proprietary.
In this chapter the operations that characterize it will be fully described.

1.1 DESCRIPTION OF THE ALGORITHM

Blowfish is a block cipher with secret key of variable length. It is composed by a Feistel network that repeat one coding algorithm 16 times.The size of block is equal to 64 bits, while the size of the key is variable from 32 to 448 bits. Actually, the most long part of the method is the initialization part which take place before the coding.
There are two parts: the first one is related to the expansion of the key while the second is related to cryption of the data.

1. *Expansion* takes as input the key composed of 448 bits at maximum and gives back a number of array of sub-keys for a total amount of 4168 byte;
2. *Encryption* operation for data includes the use of 16 blocks, each one called round. Inside each round the operations of permutation and substitution are made accordingly to the key and to the data.

The only logical operation used is the XOR while the addition between 32 bits words is the unique mathematical operation. Additionally, each round makes four accesses to array of data.

1.2 ENCRYPTION

The definition of Blowfish is Feistel network composed by 16 round. The input element is composed by 64 bits and the other elements used in the algorithm appertain to two structure called P-array and S-box. How these two structures are computed is fully explained later in this document.

```
1. Input x is divided into two equal parts $x_L$ and $x_R$;
2. for $i = 1$ to 15;
3.     $x_L = x_L \text{ XOR } P_i$;
4.     $x_R = F(x_L) \text{ XOR } x_R$;
5.     $Swap\ X_L\ e\ X_R$;
6. $x_L = x_L \text{ XOR } P_16$;
7. $x_R = F(x_L) \text{ XOR } X_R$;
8. $x_R = x_R \text{ XOR } P_{17}$;
9. $x_L = x_L \text{ XOR } P_{18}$;
10. $\text{CYPHERTEXT} = x_L\ x_R$;
```

The algorithm is shown in the (Fig.1) and described in the pseudo-code above.

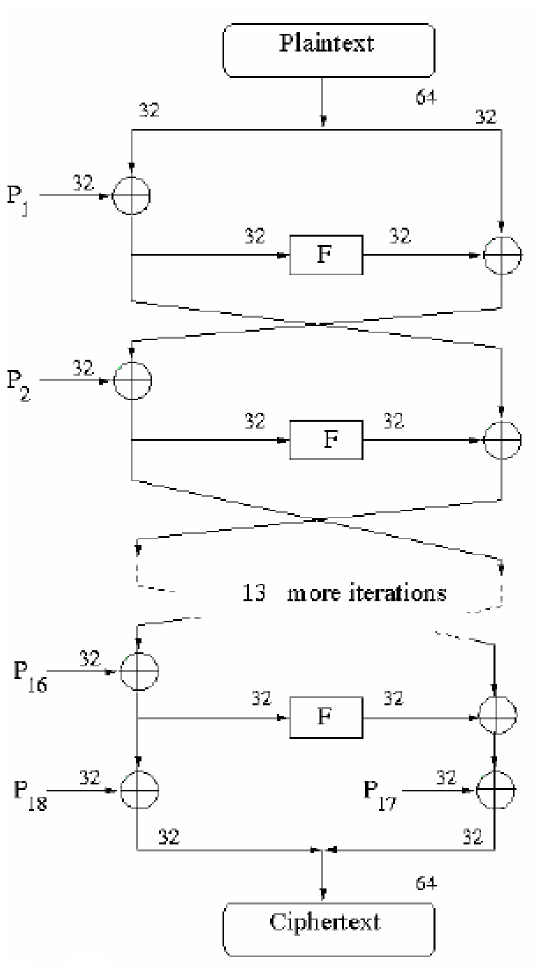


Figure 1: Blowfish diagram

The function **F** implements these operations:

1. Divide x_L in four blocks of 8 bits
2. $F(x_L) = ((S_{1,a} + S_{2,b} \bmod 2^{32}) \text{ XOR } S_{3,c}) + S_{4,d} \bmod 2^{32}$

In the (Fig.2) the function **F** is represented.

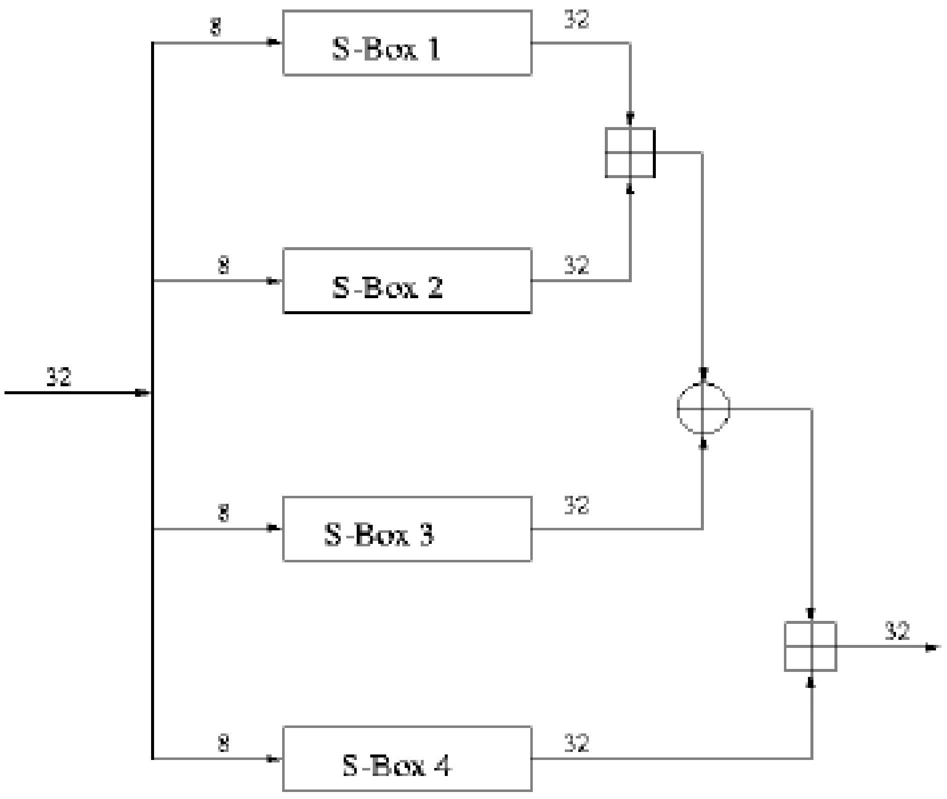


Figure 2: Function F

1.3 DECRYPTION

Decryption works like encryption, except for the order of using sub-keys which is inverted; the difference is shown in the (Fig.3).

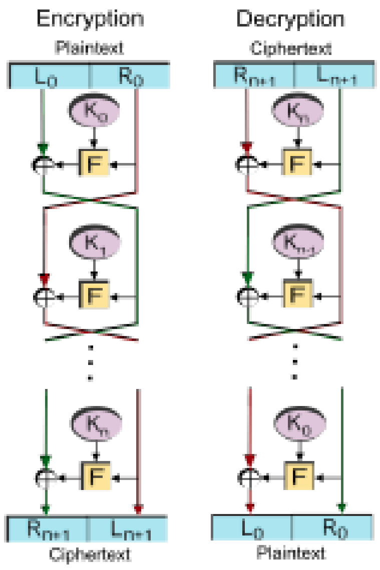


Figure 3: Encryption and Decryption

1.4 SUB-KEYS

Blowfish is characterized by a large number of sub-keys. These keys have to be precomputed even for encoding or decoding datas.

1. *First* subsystem of sub-keys is the P array and it is composed of 18 elements of 32 bits;

2. *Second* subsystem is composed of four S-box utilized as lookup table, addressed by one word of 8 bits and capable of extracting one 32 bits word among 256 words, (Fig.4).

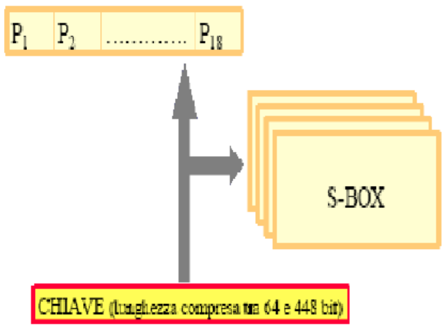
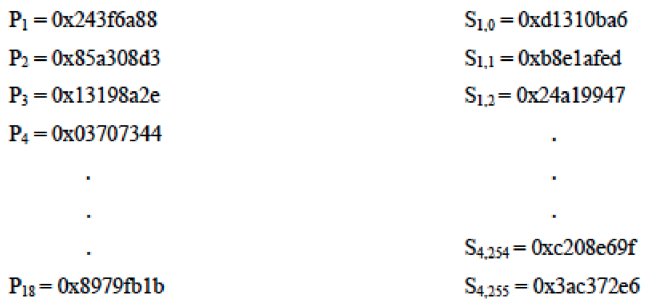


Figure 4: From key to sub-keys

1.4.1 Sub-keys and S-box generation

The key point in the generation of sub-keys is that Blowfish algorithm is used also for this; the method follows the next steps:

1. The first point is to initialize the P-array and then the four S-Box with a fixed string. This string is made by the hexadecimal digit of π, without the initial 3. For example the scheme will be this:



$P_1 = 0x243f6a88$	$S_{1,0} = 0xd1310ba6$
$P_2 = 0x85a308d3$	$S_{1,1} = 0xb8e1afed$
$P_3 = 0x13198a2e$	$S_{1,2} = 0x24a19947$
$P_4 = 0x03707344$	.
.	.
.	.
.	$S_{4,254} = 0xc208e69f$
$P_{18} = 0x8979fb1b$	$S_{4,255} = 0x3ac372e6$

Figure 5: Initial values

2. Then the XOR operation of the first 32 bits of P_1 will be done with the first 32 bits of the key and so on for all the bits of the key (Fig.6).

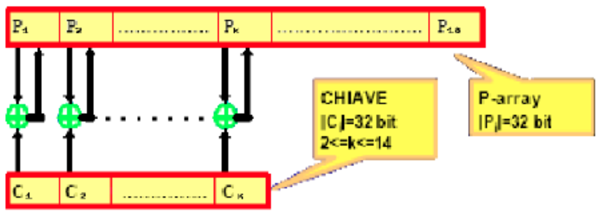


Figure 6: Sub-keys generation

3. This cycle will be repeated until a XOR operation of the full P-array will be executed. Being the key not sufficiently long it will be extended replicating it a certain number of time (Fig.7).

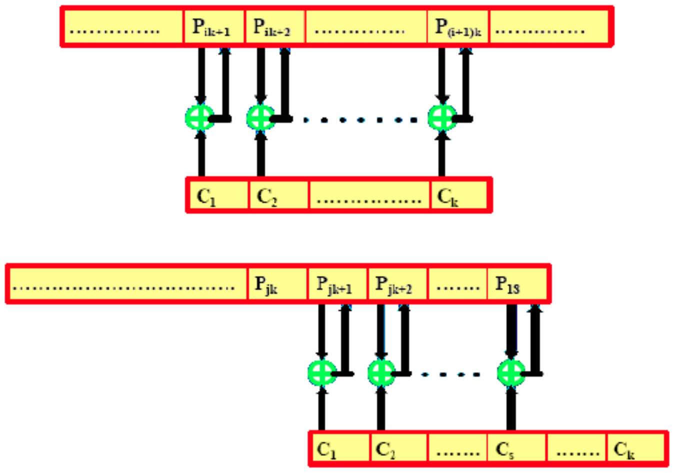


Figure 7: Sub-keys generation

4. A string of zeroes will be coded using Blowfish algorithm with the P_i keys of previous steps.

5. P_1, and, P_2 will be substituted with the output of previous step.

6. The second output will be coded using Blowfish algorithm with the first two sub-keys modified.

7. P_3, and, P_4 will be replaced by the third output(Fig.8).

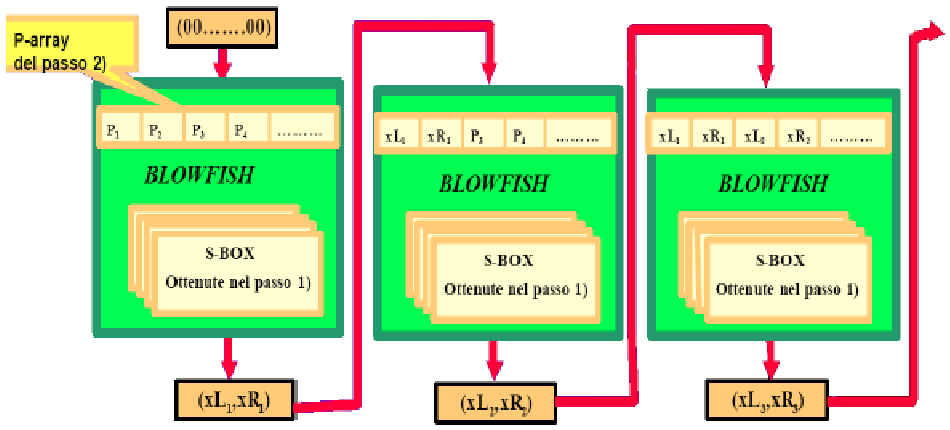


Figure 8: Sub-keys generation

8. This method is iterated until P-array is completed (Fig.9).

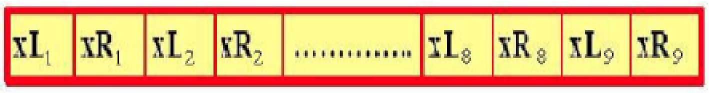


Figure 9: Final P-array

9. Finally, each of the S-box will be replaced using the same method of P-array.

As introduced an high number of iteration is requested: $18/2$ iterations for generating P-array and $256/2$ for each of the S-Box and so a total number of iterations equal to 512.

2 MULTI-THREADING CONSIDERATIONS

In this chapter all the considerations for multi-threading are reported.

First argument is related to the separation of input data into parts, fundamental for work distribution: our certainty is that distribution has to be balanced as much as possible in order to reach maximum speed in execution.

Our choice of design is to use a worker-boss model (Fig.10) for multi-threads, being the most common technique in our concurrent programming experience. Second argument is memorization of the file from disk to RAM memory: program has been tested with a series of different files like `.txt` files, .avi files, .iso file and others. This evaluation step shows that treating with different files means have different percentages of used memory; important point is that file sizes comparable to primary memory size do not allow transferring all the blocks in one time. So a second step of division is implemented and without risking any other different or more complex method we simply re-use the same technique twice.

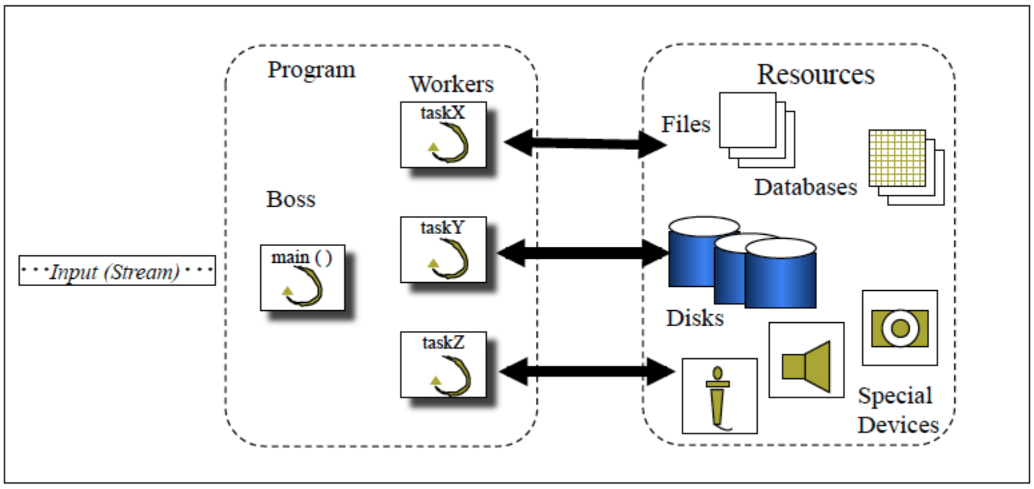


Figure 10: Boss-Workers model

Second division is also easier with respect to first one, because the size of slice has been chosen with a value divisible by eight and so the remainder will have the some property too.

2.1 DATA DISTRIBUTION

The constraint in this operation is that block size has to be a multiple of 64 bits, which is the block size defined by Blowfish algorithm. Otherwise, file length is not correlated to block size; so the idea is to give to each generated thread its block forced to have a 64 bits multiple dimension; the remaining part is computed only after the computation of blocks.

2.2 MEMORY MANAGEMENT

File and blocks can have any size and so store it completely into the RAM can be not always possible; otherwise forcing a lot of small accesses to the disk decrease dramatically the performance. Our solution is based on working on elements called slices composed of multiple groups of 64 bits but inferior to a maximum arbitrary dimension, decided with a lot of testing on the code.
So, when the block size is too much big the second division in slices occurs and second remaining part is implemented and handled as in the previous case, without padding case. We underline the fact that in RAM memory are stored blocks only when they are less than a fixed value, otherwise one slice for each thread is the real element stored in memory.
In this way only when the loaded part is coded/decoded and when the results are stored into the output file the next slice can be loaded. Two level of generation of a remainder is not a problem being the handling algorithm exactly the same.

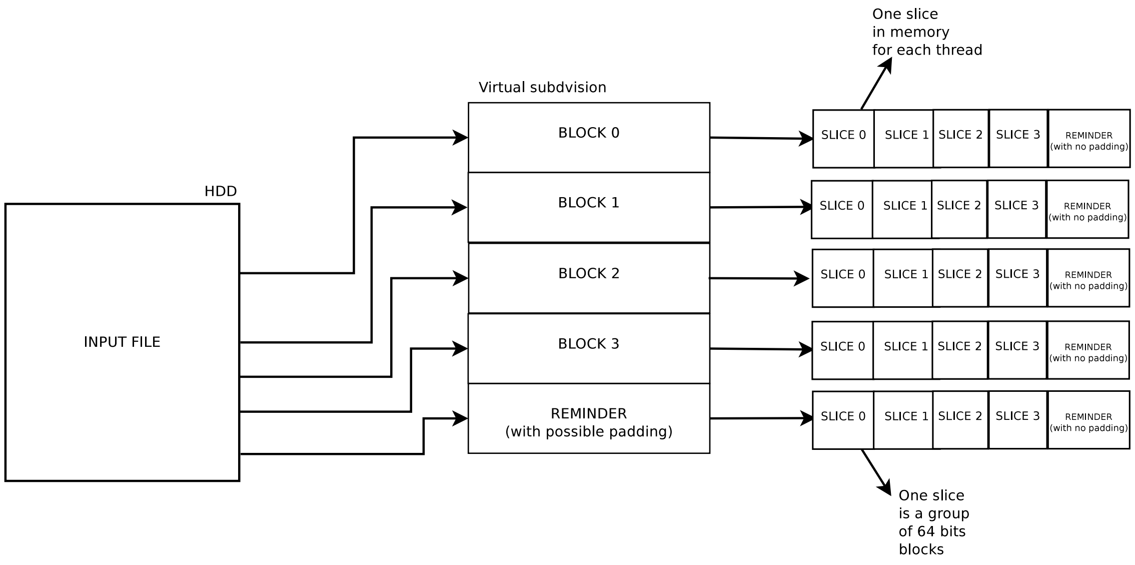

One slice in memory for each thread
Virtual subdvision
HDD
INPUT FILE
BLOCK 0
BLOCK 1
BLOCK 2
BLOCK 3
REMINDER (with possible padding)
SLICE 0
SLICE 1
SLICE 2
SLICE 3
REMINDER (with no padding)
One slice is a group of 64 bits blocks

Figure 11: Data distribution

2.3 HANDLING REMAINING PART

Considering one file with a generic length, remaining part can be equal to zero or different, even among blocks and inside a block; these two possible situations depend on dimension of the file and dimension of the block. Only for remainder of the blocks we have to extend it in order to reach the requested size of 64 bits; for that we use a famous technique implemented in signal theory, called padding. This simple method add a sequence of byte in order to reach the right dimension for elaboration; each byte has as value the number of byte of padding, making easy understanding how many bytes we add. Only at the decryption moment the padding part is removed.
Finally, for reminder inside a block, padding is not necessary.

2.4 SYNCHRONIZATION

The unique point at which accesses are controlled using semaphores are located where read or write operations on files are implemented. We must guarantee that location in a particular point of the file is not disturbed by the same operation done by another thread. Only when read or write operation is terminated the semaphore is released.

3 CODE IMPLEMENTATION

As illustrated in chapter one the algorithm fro encryption or decryption is very complex and this means that committing errors is very simple, especially writing it starting from zero.
The solution founded regards using Paul Kocher's single-threaded implementation downloaded from https://www.schneier.com/code/bfsh-koc.zip. The big advantage is that these functions are already tested and so attention moves to multi-threaded adaptation.

3.1 PAUL KOCHER'S FUNCTIONS

The three functions ready to be used are:

```
void Blowfish_Init(BLOWFISH_CTX *ctx, unsigned char *key, int keyLen)
void Blowfish_Decrypt(BLOWFISH_CTX *ctx, uint32_t *xl, uint32_t *xr)
void Blowfish_Encrypt(BLOWFISH_CTX *ctx, uint32_t *xl, uint32_t *xr)
```

The first function is the first in time and it implements the mechanism of sub-keys generation described in chapter one. Using the key the context, as indicated in the code, is initialized. Finally the context is used by encryption/decryption function; the two functions for encryption and decryption work on 64 bits block.

3.2 CODE STRUCTURE

3.2.1 Project files

The project is composed by three files:

- Blowfish.h is the header file for the functions implemented by Paul Kocher
- Blowfish.c is the file where the functions are implemented
- main.c contains all our code for multithread

3.2.2 Inputs

There are two modes to insert the inputs by the user:

- First method is to insert each variable on e by one by command line
- Second method is to insert all the variables by command line in a single line, using the format :

  ```
  <Name_project> <-(e|d)> <file_input_name> <file_output_name> <number_thread> <key>
  ```

  For example *./multiblowfish -e contets.txt contents_crip.txt 3 rafabuso*

3.2.3 File subdivision

Firstly the file size is computed and then, using the number of threads, we obtain the dimension of the block. Then the block is adjusted in order to became divisible by 8 Byte and so the dimension of the remainder and of the padding are computed.

If the block is not too big according to our threshold, the slice is equal to the block; in the opposite case we assign to the slice its maximum value and we compute the second reminder.

3.2.4 Thread creation

The main function create the threads giving them as parameter one struct object containing information about starting point of the element to elaborate.

3.2.5 Reminder and padding

The reminder is processed by the main function in two steps: in the first step the part that can be adjusted in blocks of 64 bits is elaborated while the second step apply the padding before the cryptographic operation. Between these two parts the main function wait for the ending of all threads.

The decode operation use the property of the padding that make the file divisible by 8 Byte and before terminating the bytes of padding are removed from the output file.

3.2.6 Ending

Before terminating the program two operations are realized: all the dynamic structures and the semaphores are de-allocated and all the variables are assigned to zero value, avoiding that some traces into the memory.

3.2.7 Thread code

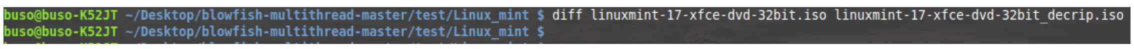
Each thread takes its own blocks and elaborates each 64 bits element into each slice. After that the function of the algorithm are invoked and then the results are stored into the output file. Only in the case that method of slice has been used, the thread handles the reminder, knowing that no padding has to be added.

4 TEST AND PERFORMANCE

In order to check the functionality of our *Multi-Threads Blowfish* we have made some different tests.
Each test is composed by two different measurements:

1. **Computation time**, is the time needed to encrypt a given file, and decrypt it (in next pages we have reported for any case, encrypting time only).

2. **Differences between original file and decrypted one**, after the encrypt-decrypt cycle we need to know whether the initial and final files are equivalent or are not. To accomplish that appointed target we exploited the *diff* UNIX command: `diff [options] from-file to-file`.
   That command allows us to see what are the differences between two files. So, in our cases we had expected that *diff* always returns void parameter, because the two input files have to be equal (Fig.12).



```
buso@buso-K52JT ~/Desktop/blowfish-multithread-master/test/Linux_mint $ diff linuxmint-17-xfce-dvd-32bit.iso linuxmint-17-xfce-dvd-32bit_decrip.iso
buso@buso-K52JT ~/Desktop/blowfish-multithread-master/test/Linux_mint $
```

Figure 12: *diff* - example with `Linux_mint.iso`

The results of tests are function of Kernel version and computer hardware in which our program runs. The following data have been obtained on a Linux machine, which has kernel version 3.13.0-24 and *Intel(R) Core(TM) i7 CPU Q 740 @ 1.73GHz*, which supports *HyperThreading* for running smartly up to two threads per CPU.

The files used to test the program, and their dimension, are listed below:

- *Contents.txt*, `337 bytes`;
- *Lorem_500.txt*, `3.384 bytes`;
- *Lorem_5000.txt*, `34.202 bytes`;
- *pg26798.txt*, `231.823 bytes`;
- *pg21531.txt*, `244.979 bytes`;
- *pg18680.txt*, `281.506 bytes`;
- *Bible.txt*, `4.397.206 bytes`;
- *Kalopsia.mp3*, `11.497.478 bytes`;
- *Mobile_application.mp4*, `111.897.927 bytes`;
- *Linux_mint.iso*, `1.251.999.744 bytes`;

Each file has been tested with different threads number: 1, 2, 4, 8, 6, 10, 15, 20, 25, 32. And the used files allow to cover the program behaviour from few hundreds of bytes up to more than one gigabytes.

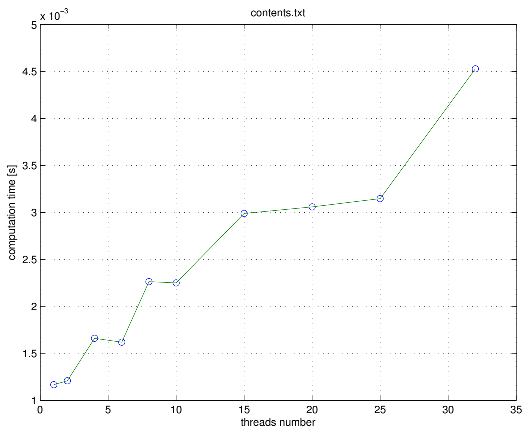
4.1 RESULTS

THREAD NUMBER	CONTENTS	LOREM_500	LOREM_5000	PG26798	PG21531	PG18680
1	0.001166 *sec*	0.001725 *sec*	0.006958 *sec*	0.022432 *sec*	0.018046 *sec*	0.022671 *sec*
2	0.001208 *sec*	0.000877 *sec*	0.004274 *sec*	0.021695 *sec*	0.022763 *sec*	0.020236 *sec*
4	0.001660 *sec*	0.002445 *sec*	0.003399 *sec*	0.014684 *sec*	0.011303 *sec*	0.012489 *sec*
6	0.001619 *sec*	0.002661 *sec*	0.003692 *sec*	0.013037 *sec*	0.011009 *sec*	0.014170 *sec*
8	0.002263 *sec*	0.003208 *sec*	0.004891 *sec*	0.011334 *sec*	0.010262 *sec*	0.010258 *sec*
10	0.002250 *sec*	0.002128 *sec*	0.004060 *sec*	0.010997 *sec*	0.017674 *sec*	0.011799 *sec*
15	0.002989 *sec*	0.002430 *sec*	0.005109 *sec*	0.007741 *sec*	0.011670 *sec*	0.012083 *sec*
20	0.003059 *sec*	0.002979 *sec*	0.005693 *sec*	0.013526 *sec*	0.013580 *sec*	0.012127 *sec*
25	0.003147 *sec*	0.004213 *sec*	0.005403 *sec*	0.026283 *sec*	0.010970 *sec*	0.016384 *sec*
32	0.004529*sec*	0.004151 *sec*	0.004861 *sec*	0.015293 *sec*	0.012811 *sec*	0.016384 *sec*

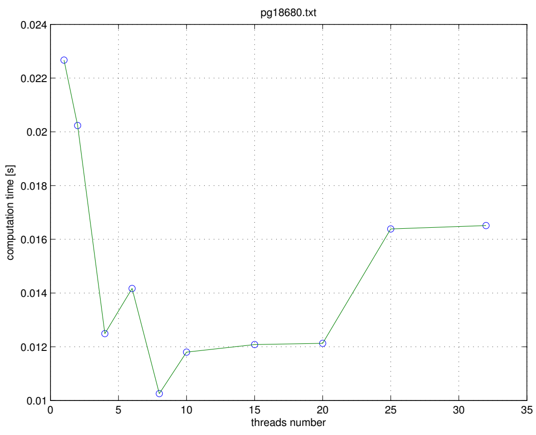
THREAD NUMBER	BIBLE	KALOPSIA	MOBILE_APPLICATION	LINUX_MINT		
1	0.258397 *sec*	0.683342 *sec*	10.429358 *sec*	170.071742*sec*		
2	0.156826 *sec*	0.393428 *sec*	4.095223 *sec*	89.516180 *sec*		
4	0.135139 *sec*	0.332331 *sec*	3.296534 *sec*	69.916838 *sec*		
6	0.100043 *sec*	0.254143 *sec*	2.388198 *sec*	61.714959 *sec*		
8	0.091301 *sec*	0.215308 *sec*	2.159168 *sec*	50.824318 *sec*		
10	0.097573 *sec*	0.226689 *sec*	2.293672 *sec*	43.850544 *sec*		
15	0.090361 *sec*	0.217842 *sec*	2.038637 *sec*	36.783085 *sec*		
20	0.094304 *sec*	0.218173 *sec*	2.692941 *sec*	39.284932 *sec*		
25	0.094643 *sec*	0.222394 *sec*	2.010600 *sec*	36.185404 *sec*		
32	0.096315 *sec*	0.214665 *sec*	2.170294 *sec*	35.900116 *sec*		

4.1.1 Graphs

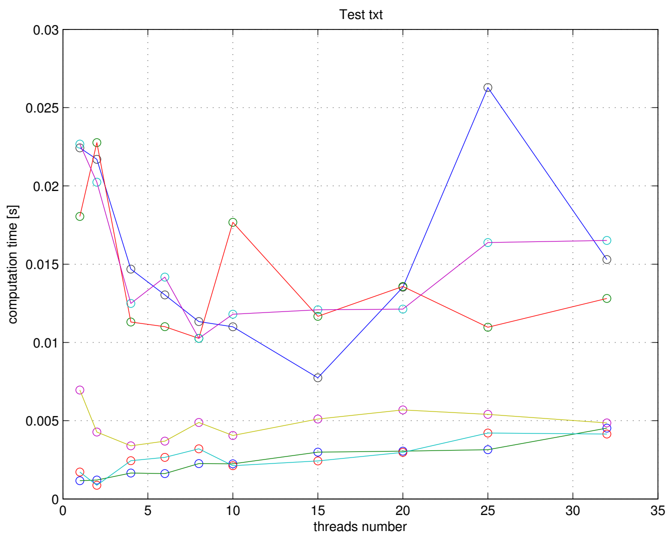
The previous table points out some important characteristic of our program. First of all, multi-threads implementation does not perform very well for all of those files that are too small, for example (Fig.13) shows that anomaly for the whole *.txt* files that have been given to us to test the software. In particular, (Fig.13) shows the computational time in function of threads number for the smallest given text file (`contents.txt`), the biggest one (`pg18680.txt`), and all of them.



(a) `contents.txt`



(b) `pg18680.txt`



(c) All of the `.txt`

Figure 13: Test with small file

The other tested files are: the biggest book known by us, the Bible (Fig.14), a song of the *Queen of the Stone Age*, Kalopsia.mp3 (Fig.15), a video course of *Politecnico di Torino*, Mobile_application.mp4, the disk image of *Linux Mint* distro (Fig.16) and Linux_mint.iso, .
As can be seen, the time required by algorithm decreases very well while the threads number raises, even if the input file is very big.

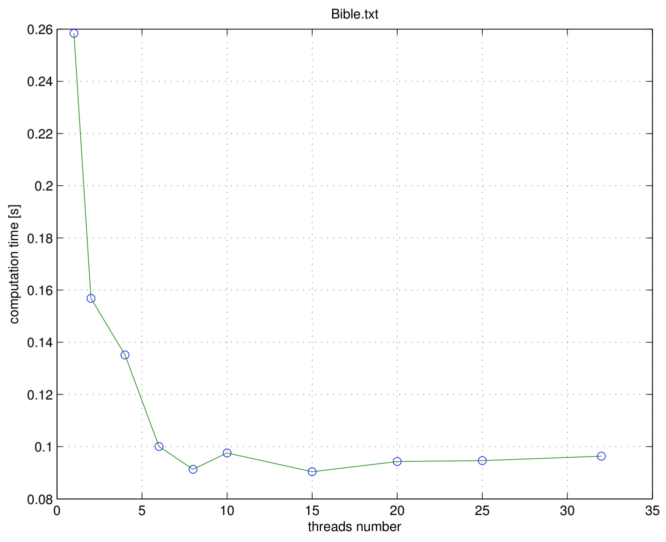


Figure 14: `Bible.txt`

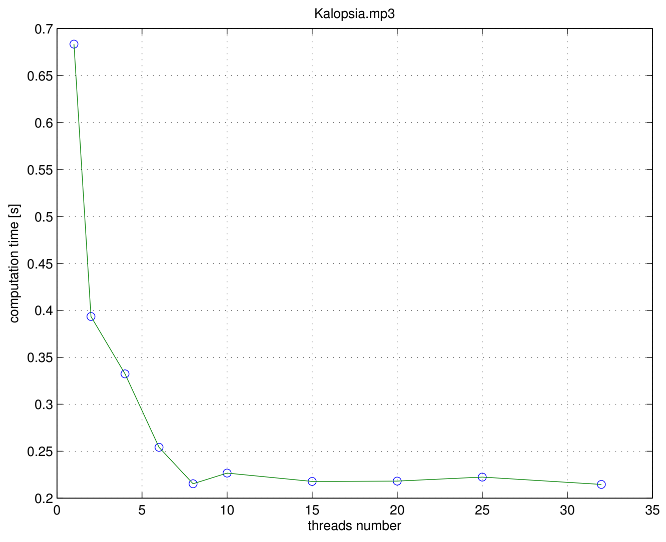


Figure 15: `Kalopsia.mp3`

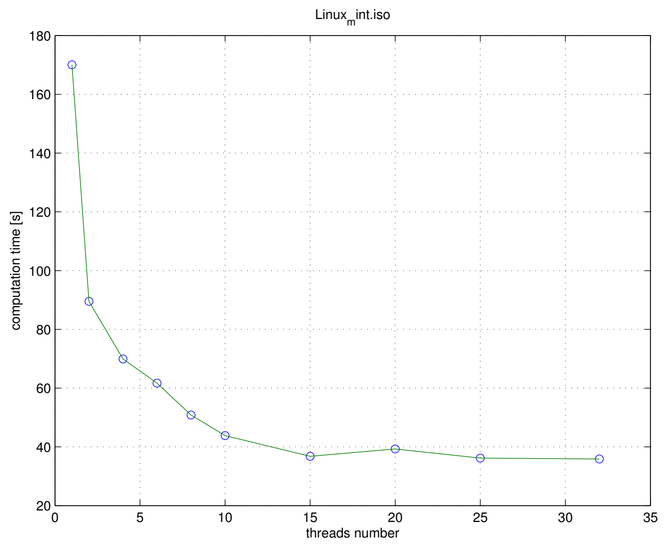


Figure 16: `Linux_mint.iso.mp3`