POLITECNICO DI TORINO

Collegio di Ingegneria Elettronica

**Corso di Laurea Magistrale**

**in Ingegneria Elettronica**

Operating system project

**Multithread application of Blowfish algorithm**



Yuri Fioretti mat. 211386

Gianluca Orasi mat.211383

September 2014

**Index**

**Blowfish algorithm**

**Multithread implementation**

**Evaluating Performance**

**1 Blowfish algorithm**

* 1. **Introduction**

Blowfish is an encryption/decryption algorithm designed by the American cryptographer Bruce Scheier in 1993 created as an alternative to the DES ( Data Encryption standard). This powerful algorithm is valuated in fact more secure than DES, in particular , for the fact that DES algorithm works with a too short key and with fixed S-boxes, and for some controversies that raised for a possible manipulation by NSA

(National Security Agency).

Blowfish offers lots of benefits that ensure it in much applications. It is simple to implement, fast and can be stored on less than 5Kb of memory space.

* 1. **Description**

Blowfish is a Feistel network cipher of sixteen rounds that works on 64-bit fixed block size. A Feistel network is a particular block structure of some ciphers that is usually symmetric, meaning that the key used for encryption is used for decryption as well, because decryption works with the reverse order of encryption steps XOR E 32 BIT ADD. This key can be variable from 32 bits up to 448 bits length.

The algorithm uses two principal components: a P-Array and four S-boxes, and the only two operation required are XORs and sum. It is made by two principle operations: key expansion and the encipher (decipher).

**1.2.1 Key expansion**

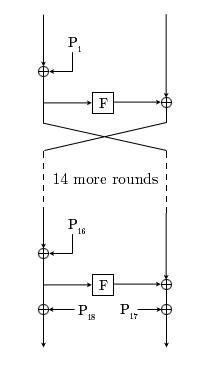
Key expansion operation consists in the usage of the key and an initialization of the P array and the S boxes in order to obtain another P-Array containing 18 sub-keys of 32-bit ( P1, P2, …. P18) and other S-Boxes each containing 256 keys of 32-bit length. Key expansion is composed by many steps:

1. Initialize the P-array and the S-boxes with the hexadecimal representations of pi-mantissa
2. XOR P1 with the first 32-bit of the key, XOR P2 with the successive 32-bit, and so on eventually until P14 in the case of a 448-bit length key. Then repeat with the last element of the P-Array.
3. Cipher an all-0 string with blowfish algorithm using previous P-Array and S-boxes.
4. Substitute P1 and P2 with the output obtained from the previous step, cipher P1 and P2, then substitute P2 and P3 with the output
5. Repeat step 4) until the P-Array is generated and do the same for the S-boxes.

In total, 521 iterations are required to generate all required sub-keys.

**1.2.3 Encryption**

Encryption receives in input a data block with a fixed size of 64-bit. Data is initially split in two 32-bit parts, and it’s processed across 16 round, using the P-array and the S-boxes generated previously computed by key expansion. Each round consists in 32-bit XOR operation, 32-bit additions, and four indexed data array look-up table. Following the Figure the pseudo-code below show the steps of encrypt operation.

Encipher

Input:

D : 64-bit data input

P array { P1, P2, … , P18 )

F() function

Algorithm:

Divide D in two32-bit parts xL,xR

for( i=0; i<16; i+++)

{

xL = xL ⊕ Pi

xR = F(xL) ⊕ xR

swap(xL,xR)

}

xR = xR ⊕ P17

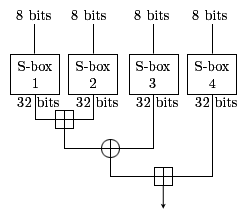
xL = xL ⊕ P18

Output = (xL, xR)

The F( ) function is defined as it follows:

The 32-bit input is divided in four byte. Each byte enter in one of the S-Boxes.

Each S-Box is a matrix that performs a substitution of the 8 bit in input and generate a 32-bit output. The outputs of the first two S-Boxes are added, then XORed with the output of the third and the added to the output of the last S-Box.



Then, the function F( ) can be summarized as:

**1.2.4 Decryption**

Blowfish is a symmetric encryption algorithm. This means that decryption can be done with the same key. In fact, decryption works as encryption, but in the reverse order. It must be implemented with the reversing order of the sub-keys of P-Array.

1. **Implementation of the algorithm**

The implementation of the algorithm itself is composed by two files: blowfish.h and blowfish.c.

The header file contains at first the definitions of the maximum dimension of the key in bit, the dimension of the P-array, and the number of the S-boxes with each dimension.

Then is defined a macro for the Swap operation, and the definitions of two types: BYTE (of 8-bits) and WORD (32-bit).

In order to do not overload blowfish.c or the main.c file with too much code, the header file contains a static declaration of the initialization of the P-Array and the S-boxes with pi’s mantissa values expressed in hexadecimal value. These value could be computed using Bailey–Borwein–Plouffe formula. Following is defined a macro for the swap operation that will be executed in encryption/decryption. Last elements in the header file are the respectively prototype of Initialization, Encryption and Decryption main functions.

Blowfish.c is the file in which is written the algorithm itself, it contains the necessary libraries and the functions (whose prototype resides in blowfish.h) needed for its correct work.

void initialization(WORD pArray[PLENGTH], WORD sBox[NSBOX][SLENGTH])

This function is the initialization of the P-Array and the S-boxes with pi’s mantice hexadecimal values using a memcpy function to copy the static arrays, defined previously on the header file, into same length arrays that the algorithm will really use.

void initializeBlowfish(BYTE key[KLENGTH], WORD pArray[PLENGTH],

WORD sBox[NSBOX][SLENGTH], int klen)

This function generates the P-array and the S-boxes that will be used in encryption/decryption following the steps of key initialization. It is supported by two array (

At first, it must be noted that Blowfish algorithm receives in input a variable-size key which length may vary from 32-bits to 448-bits. So, in order to combine the key with the

P-Array and the S-Boxes, this function copy user input key into a 32-bit array. Then the XOR operations can be executed, and so for the encryption of two initial 32-bits void words. Finally, two for cycles are used to generate the final value of P-Array and S-Boxes by sequential encryptions. Encryption iterations are supported by two 32-bit bit bi-dimensional arrays, inBlock[2] and outBlock[2]. Note that the second for is implemented with single a counter variable for matrix denotations.

void blowfishCipher(WORD inBlock[2], WORD outBlock[2],

WORD pArray[PLENGTH],

WORD sBox[NSBOX][SLENGTH])

This is the function of Blowfish encryption algorithm. It receives in input two 32-bits arrays inBlock[2] ( that can be imaged as xL and xR of the pseudo-code of previous section) . The encryption is implemented with a for cycle that executes XOR, swap operation supported by a tmp variable, and the F( ) function. At the end of the function, the resulting chiper inBlock[2] are copied with a memcpy() into outBlock[2].

void blowfishDecipher(WORD inBlock[2], WORD outBlock[2],

WORD pArray[PLENGTH], WORD sBox[NSBOX][SLENGTH])

This is the function of Blowfish decryption algorithm. As explained in the previous section, it works in the same way of encryption, except for the inverse order of the use of P-Array.

WORD F(WORD in, WORD sBox[NSBOX][SLENGTH])

This is the implementation of Blowfish F( ) function. It requires four casting to BYTE of one 32-bits inBlock[0], then the operations with S-Boxes required are made.

1. **Multi-thread implementation**

The project requirements were to implement a multi-threaded version of Blowfish algorithm able to assign encryption/decryption to a user defined number of threads. Encryption/Decryption can be done on text file or on any other file treated as a binary file. This section shows how the design was made. The entire project it’s divided in three separate files:

- blowfish.h : The header file containing definitions, macros and prototype functions

- blowfish.c : The source file implementing Blowfish key expansion, encryption and decryption functions

- main.c : The main file that manage threads implementation, the interaction between the user and the program and other several tasks.

As a sample representation, the entire design is assumed as a consumer-producer problem.

Threads are divided by:

One Reader : This single thread manages the reading of data from the input file and the filling of a buffer to be cipher/decipher by other threads.

N Ciphers/Deciphers: These threads manage the Encryption/Decryption of the datas in the buffer provided by the reader thread and fill a buffer to be consumed by the writer thread.

One writer : This single thread manages the writing of data provided by Cipher/Decipher on an output file.

FIGURAAAAAAAAAAAAAAAAAAAAAAA

**Inside main.c**

In the main.c are accomplishes several tasks:

Interaction user-program and executes necessary initialization functions

- Manages thread creation, tasks assignment and termination

- Manages thread synchronization

- MANAGE TIME????? HOW PORCODDIOOO

**Interaction user-program and initialization**

The first task main code has to accomplish is receiving all the inputs from the user necessary to the correct working of the program. As the program was designed, when launching the program, user must specify arguments following the syntax :

Blowfish <key> <input> <cipher/decipher> <output>

<key> Variable length key : from 4 bytes up to 56 bytes ( 1byte = 1 char)

<input> Name of the input file ( the program must be launched in the same directory of that file. The file can be a .txt a .jpeg etc.

<c/d> User must specify the operation required (cipher/decipher)

<output> Name of the output file ( must be the same file extension of input one)

Input arguments are collected by int main function as int argc, char \*argv[] arguments and are checked before all tasks.

In addition main.c manages opening and closing file. It also defines the libraries that will be used, the number NTHREADS of Cipher/Decipher threads that the programmer wants to run, declares the functions that will be assigned to threads and declares the variables that the program will use.

Following the producer-consumer structure of the implementation, the two most important data structure are two shared buffer: inBlock[NTHREADS][2] and inBlock[NTHREADS][2]. The first buffer is shared between the Reader thread and the Cipher/Decipher threads, the latter is shared between the Chiper/Decipher threads and the Writer thread. Each buffer consists in NTHREADS array each with two 32-bits elements.

**Threads creation, tasks assignment and threads termination**

In order to manage the threads to which assign tasks, some preliminary operations are made:

threadArray[NTHREAD] is generated in order to collect an identification number of each Cipher/Decipher thread.

The creations and task assignments to the Reader and the Writer threads are made via

pthread\_create(&tR, 0, threadRead, NULL ) and pthread\_create(&tW, 0, threadWrite, NULL) functions (where tR and tW store the thread’s respectively TID).

void \*threadRead(void \*args) is the task assigned to the Reader thread. It reads 8 chars at time and put them in inBlock buffer, whose two 2 WORD elements are casted into 8 BYTE elements. Reader Thread writes until EOF is reached.

void \*threadWrite(void \*args) is the task assigned to the Writer thread. It receives in input outBlock buffer from Cipher/Decipher threads and writes it into an output file via casting in BYTE the buffer.

Encipher and Decipher thread are creating by main depending on the specification of user. To recognize if Encipher or Decipher operation is required, the main use a switch – case statement. If Encryption is required, NTHREAD Cipher threads are created and blowfishCipher function is assigned to them, otherwise NTHREAD Decipher threads are created and blowfishDecipher function is assigned to them.

In order to don’t lose data due threads termination, pthread\_join function is implemented at the finish of all Encryptions/Decryptions for Reader, Cipher/Decipher and Writer threads.

**Threads synchronization**

The design of this program would rise race conditions among the Reader thread and Cipher/Decipher threads (and so for Cipher/Decipher threads and the Writer thread) if it wouldn’t be supported by structures that ensure data protection on shared buffers.

In the main.c file synchronization is achieved with the use of semaphores provided Semaphore.h library.

The semaphores used are:

semIn[NTHREADS] and semCiphIn[NTHREADS] manage the synchronization between Reader thread and Cipher/Decipher threads.

semOut[NTHREADS] and semCiphOut[NTHREADS] manage the synchronization between Reader thread and Cipher/Decipher threads.

As the figure shows:

-Reader thread performs :

wait(semCiphIn[i]) to signal Cipher/Decipher threads that an input buffer is going to be used on reading operation

post(semhIn[i]) to signal Cipher/Decipher threads that an input buffer is released to be used in Encryption (Decryption).

-Cipher/Decipher threads perform

wait(semhIn[i]) to signal Reader thread that an input buffer is going to be used in Encryption (Decryption).

wait(semCiphOut[i]) to signal Writer thread that an output buffer is going to be used in Encryption (Decryption).

post(semhOut[i]) to signal Writer thread that an output buffer is released to be used in writing operation.

post(semCiphIn[i]) to signal Reader thread that an input buffer is released to be used in reading operation.

-Writer thread performs :

wait(semhOut[i]) to signal Cipher/Decipher threads that an output buffer is going to be used on writing operation.

post(semCiphOut[i]) to signal Cipher/Decipher threads that an output buffer is released to be used in Encryption (Decryption).



In the main.c are also initialized two int variables stopCipher and stopWrite. These used to report to Cipher/Decipher that all buffers were processed. This two variables aren’t implemented like semaphores, but simple variables that act as controls.

THREAD TIME