B.U.G.S .
DYNAMIC CRYPTOGRAPHY ALGORITHM

DOCUMENTATION

Author: Sylvain Martinez
Version: 5.1
Last Updated: 28/11/06

Forewords

There has been no cryptanalysis done on the algorithm discussed in this document. The author does not claim, nor guarantee, for it to be secure, strong, unbreakable, ground breaking, etc.

It is an algorithm which has been created for fun, by a person interested in cryptography and computer security over a period of about 10 years.

The reader should not require a high level of mathematics knowledge but some basic cryptography knowledge is required to appreciate the document's content.

The first chapter of this documentation will briefly introduce the concept of the BUGS algorithm and its history. The second chapter will give an overview of the algorithm while the third and last chapter will focus on the algorithm details.

CONTENTS

Chapter I – BUGS Algorithm Introduction

I.1 Terminology

The following abbreviations and assumptions will be used in this document:

BUGS – **B**ig and **U**seful **G**reat **S**ecurity . The meaning was "made up" in 1997 to match the algorithm name as explained in I.1 History.

KD – **K**ey **D**ependant – a value which is "Key Dependant" will be different each time a different key is used.

PRN – **P**seudo **R**andom **N**umber – In general, all references to Random Number in this document should be taken as a reference to a Pseudo Random Number.

LO – **L**ogical **O**peration – This can be also referenced as a ***XOR***, ***AND***, ***NOR***, ***NAND***, etc

LFSR - **L**inear **F**eedback **S**hift **R**egister – Algorithm used to generate ***PRN*** with the least repetition possible for its output numbers.

Integer Size is 32 bits – The use of a 32 bits hardware platform is assumed.

ISAAC – This is a fast cryptography random generator created by Bob Jenkins and used in ***BUGS***. The details of the ***ISAAC*** Algorithm will not be discussed in this document.

I.1 History

This algorithm started as a personal project in a beautiful but somewhat boring summer holiday of 1995. A first version of the algorithm was named after my student nickname, ***BUGS***, and completed in early 1996 while studying at a French computer school in Lyon (DUT Informatique, Lyon1). A small contest to test its strength ended up being posted on different cryptography forums. Pressure from the French cryptography laws at the time, as well as the DST (French Domestic Secret Service), proved to be too much for my University's director who asked me to cancel the contest and stop developing the algorithm.

If anything, this gave me the motivation boost I needed and I redesigned most of the algorithm in 1997 as part of my BSc project in the UK.

In 1999, the algorithm attracted the attention to few people and companies thanks to its portability and being open source. One person especially, spent a lot of times trying to break the algorithm. That person, who I will just name "Simon", introduced himself as a bored teenager and over few months showed a very high level of understanding of the algorithm itself, highlighting a number of weaknesses and translating most of the algorithm in pure assembler for efficiency and test purposes. I was responding to each weakness highlighted by the unusually technically talented and politically opinionated teenager, by spending days and night working for an improved algorithm design. This resulted in 10 intense months of work and the latest version of the algorithm (v4.x); as well as the end of the communication with my "muse" who suddenly had personal problems and could no longer spend time on my algorithm.
As much as I would like this "Simon" to be more than just a talented teenager, this is more than unlikely and just an interesting anecdote.

This brief history summary should give you the following information on ***BUGS***:
- It was created by a cryptography enthusiast with no real knowledge in that field except from reading about it in different books (Scheiner's, etc).
- It has evolved a lot since its creation taking users' comments into consideration.
- It has never been through a cryptanalysis.
- It is an unproven cryptography algorithm and should be dealt with as such, with caution.

I.2 Algorithm Concept

The algorithm is based on two main concepts, Logic and Dynamic, it is not based on mathematics,

I.2.1 Logical Concept

The "logical" approach taken was similar to answer to the following question:

- What would I do to make a newspaper unreadable to someone without the right knowledge?

The answer was:

- I would cut the newspaper into pieces, shuffle those pieces in a certain way, substitute some letters in the process and eventually glue the pieces back together. Resulting in a similar size newspaper but with no meanings unless one would know how to reverse the initial process.

There is no mathematics involved and the first version of the algorithm was doing just that, for a given message the characters where shuffled and substituted.

I.2.2 Dynamical Concept

The current ***BUGS*** algorithm is similar in some ways to its original version 10 years ago. It is no longer dealing with characters directly but with bits and has a more complex substitution/operation section.
The main difference lays with the fact the algorithm is as dynamic as possible, this means almost all of its components will change depending of the password itself used to generate a key or crypt a file. This is called "Key Dependency" (***KD***) and has been pushed to the extreme in the new ***BUGS*** Algorithm with dynamically linked components such as:

- The number of rounds
- The operation used on each bits
- The shift window
- The direction of each operation (Left or Right)
- The size of the key buffer when crypting a file
- The size of the block shuffle and working file block.

The algorithm behaviour changes to great extend when used with different passwords. All default settings can also be changed by the user, which means the knowledge required to decrypt/reproduce a key gets extended to the environment settings as well as the password itself.

CHAPTER II – BUGS Algorithm Overview

The algorithm is made of a number of modules and sub-modules.
This documentation will only focus on two of the main modules:

- The Key Generator module
- The File Encryption module

The “file decryption” module has been left out in purpose as it is nothing more than running the “file encryption” module backwards.

II.1 Key Generator Overview

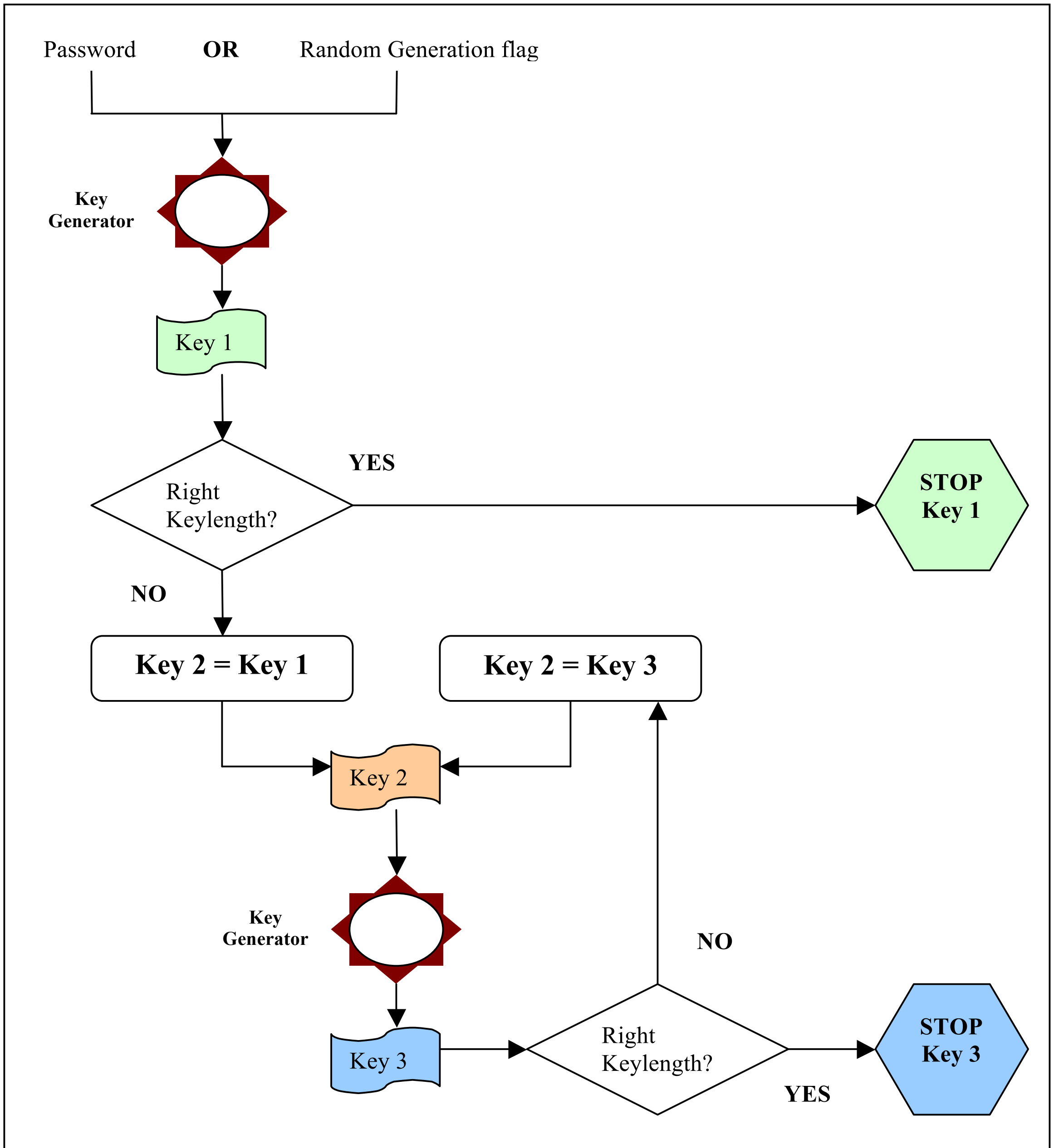
II.1.1 Pre-requisites

There are a number of pre-requisites to this module:

- Seed source
  The Seed can be a password, a key, or the result of a random algorithm (***ISAAC***, time/date based, or user customised).
- Keylength Output:
  The minimum keylength that can be generated is 128bits. Larger keys will be a multiple of 128. There is no upper limit to the size of key.
- Keylength Input:
  The seed must be at least of size N/2 when generating a key of size N. In other words, when generating a 128bits key and using a password, the user must enter a password of at least 8 characters (8*8bits = 64bits).

II. 1.2 Process

An overview of the key generation process is described in the following diagram:



f.1.1 – Key Generator - Overview

Below is a brief description of the different Key Generator steps

- STEP 1: Key Padding
  If the initial seed used to generate the key is not equal to the size of the keylength to be generated some Pseudo-Random numbers (***KD***) will be inserted at a position with is ***KD***.

- STEP 2: Bits Concatenation
  The seeds bits will be stored into one long string of bits. An Integer array will be used for this purpose and the size of each element will be dependant of the hardware platform used: 64, 32 or 16 bits. (or even 128, 256, etc when available).

- STEP 3: Initial Scrambling
  The different seed bits will be combined together to generate a Pseudo Random Number (which is therefore ***KD***). This PRN will then be added to the each seed element. Each time the PRN is added is will change (***KD***).

- STEP 4: Key Encryption
  The seeds will now be referenced as the key. Each of its bits will be treated individually and will be subject to some Logical Operations (***LO***). This is called “a round”. In each round the following happens:
  - A bit swap or a ***LO***
  - The distance between 2 bits (Shift Window) is ***KD***
  - The nature of the operation (a swap or ***LO***) is ***KD***
  - The number of round is ***KD***
  - The direction of the round is ***KD*** (left or right)

  For each key generated the minimum number of rounds is two, this will ensure that all bits have been swapped at least once AND have had a ***LO***.
  The number of rounds is also ***KD***.

- STEP 5: Final Scrambling
  A PRN is generated if no random seed is provided and will be added to each element of the Key. Each time the PRN is added it is changed using a Linear Feedback Shift Register (***LFSR***)

II.2 File Encryption Overview

II.2.1 Process

Below is a brief description of the different File Encryption steps

- STEP 1: Initialisation
    - The file is mapped into a virtual array and splited into blocks. The length of the block is ***KD***
    - Several keys are generated from the password or keyfile
    - Only one key will be generated with a random number, encrypted and inserted into the encrypted file. The insertion position is ***KD***.
    - That random key will be used to generate a PRN

- STEP 2: Seeding
    - A number of keys (***KD***) will be generated and stored into a Key Buffer using a derivation of the initial key generated and the PRN previously generated as the random seed.
    - From that key buffer 2 keys will be selected (***KD***) and an ***AND*** will be conducted. The result is an Encryption Key
    - A block will be selected from the file virtual array (***KD***) and a ***XOR*** will be conducted with the Encryption Key.
    - A New key will be derivated from one of the 2 keys used in creating the Encryption key and replace one of the 2 keys.
    - The process start again at STEP 2 until all the blocks have been encrypted (seeded).

- STEP 3: Shuffling
    - The same virtual array used in STEP 1 will be use again.
    - Two blocks will be selected (***KD***)
    - One out of three possible ***LO*** will be conducted (***KD***) on those two blocks. The result is an Encryption block.
    - A third block will be selected (***KD***) as the block to be encrypted and a ***XOR*** will be conducted with the Encryption Block.
    - One of the two blocks used to generate the Encryption block will then be selected to be the next block to be encrypted (***KD***).
    - The process start again at STEP 3 until all the blocks, but the last two, have been encrypted (shuffled).
    - The last 2 blocks will be crypted using a ***XOR*** with 2 new keys generated. Not shuffling the last 2 blocks is required for the decryption process.

Chapter III – Bugs Algorithm Details

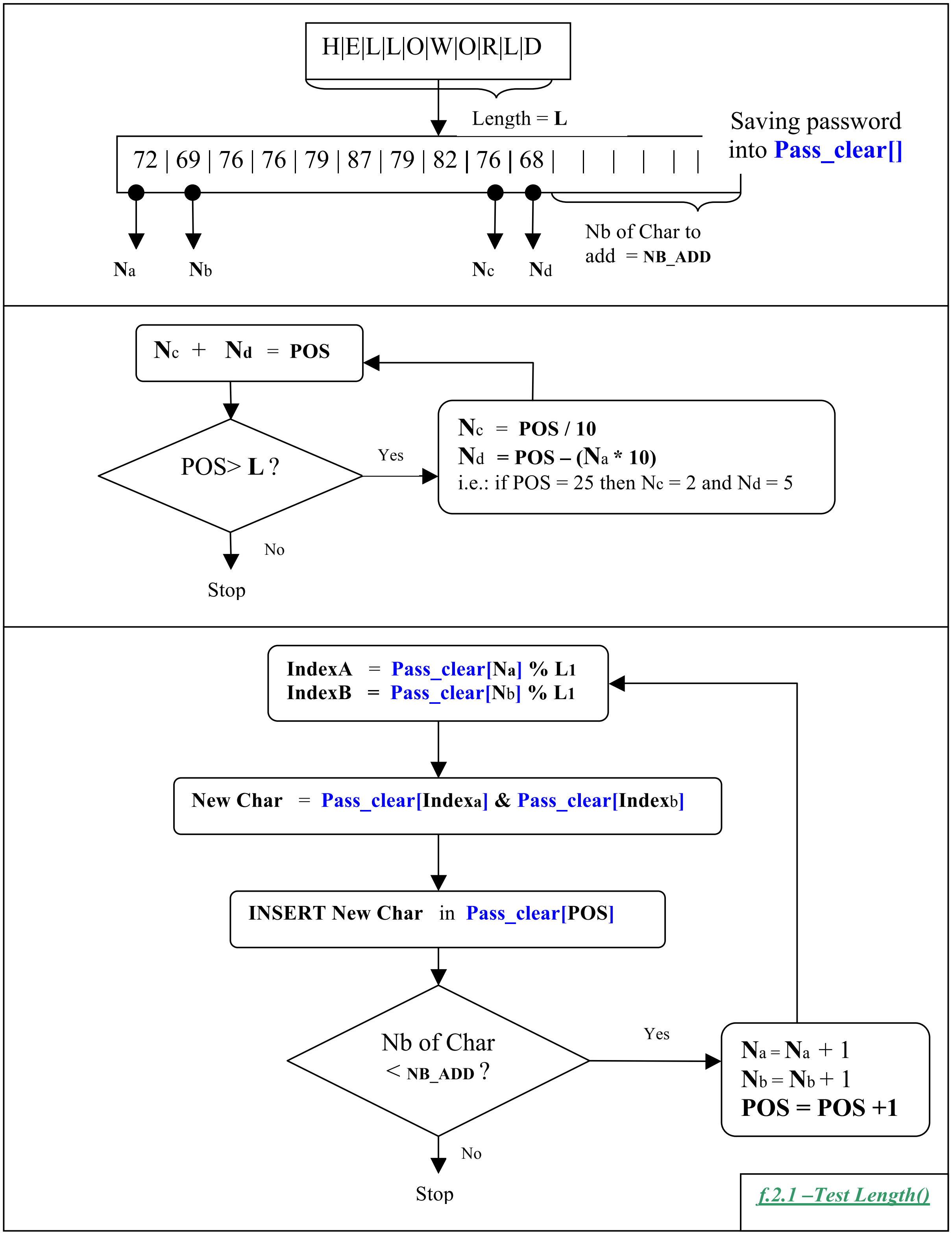
This Chapter describes in details the different steps of the following two main modules:

- The Key Generator module
- The File Encryption module

The reader is expected to have read the previous chapter and have an overall understanding of the ***BUGS*** algorithm. The details are mainly explained through Diagrams.

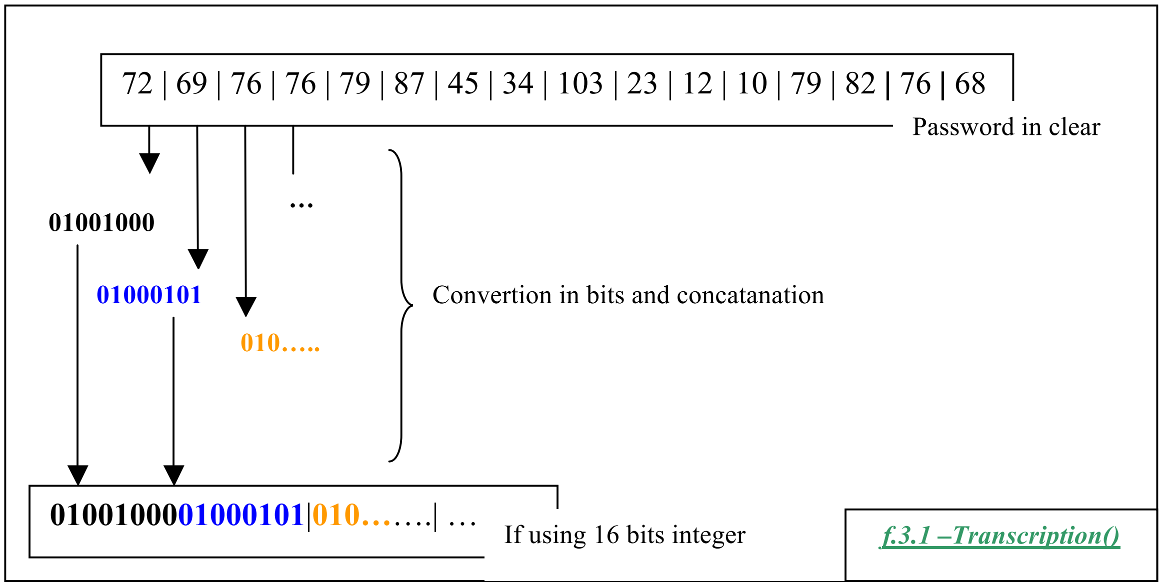
III.1 Key Generator Details

III.1.1 Key Padding



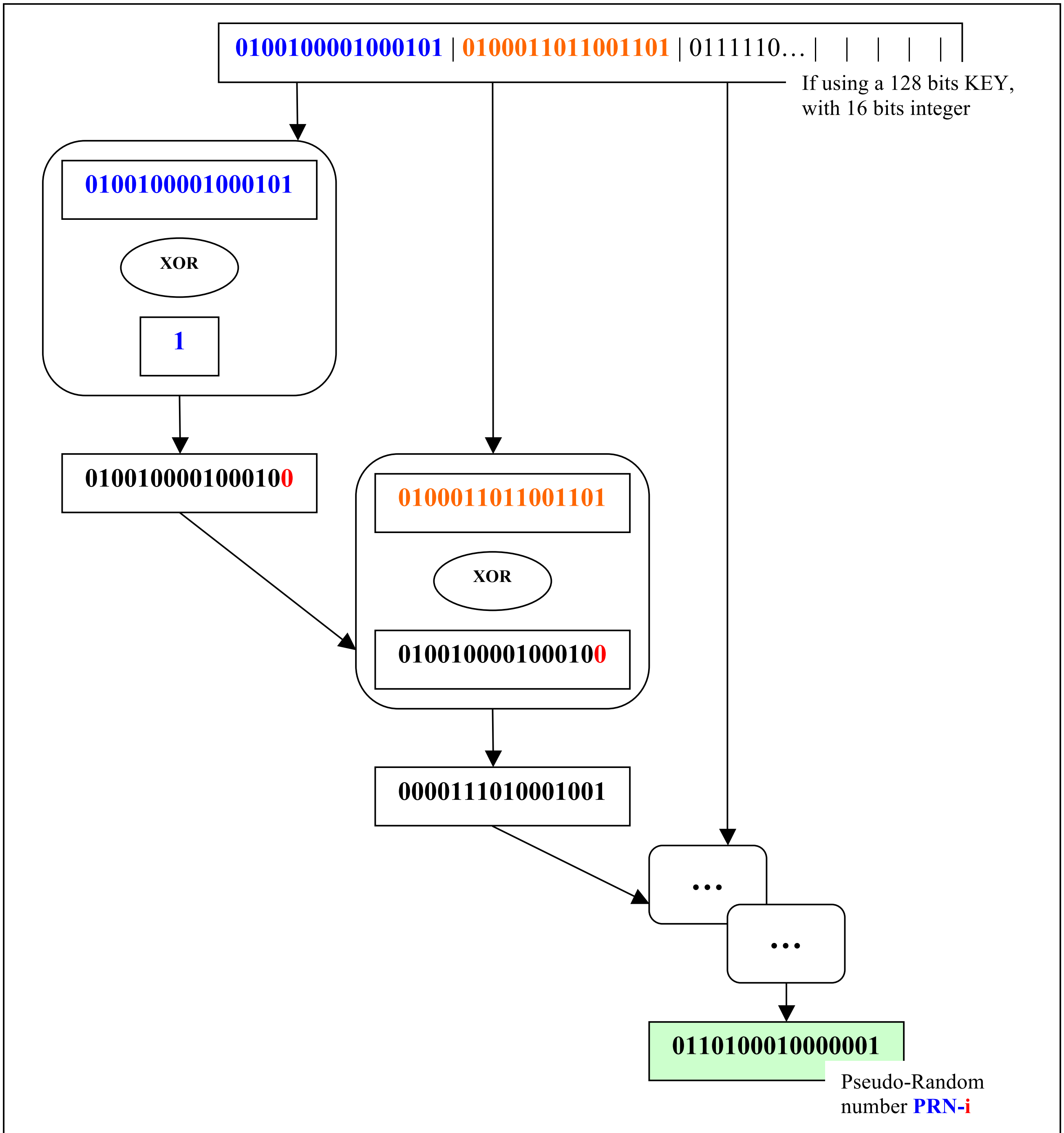
f.2.1 –Test Length()

III.1.2 Bits Concatenation

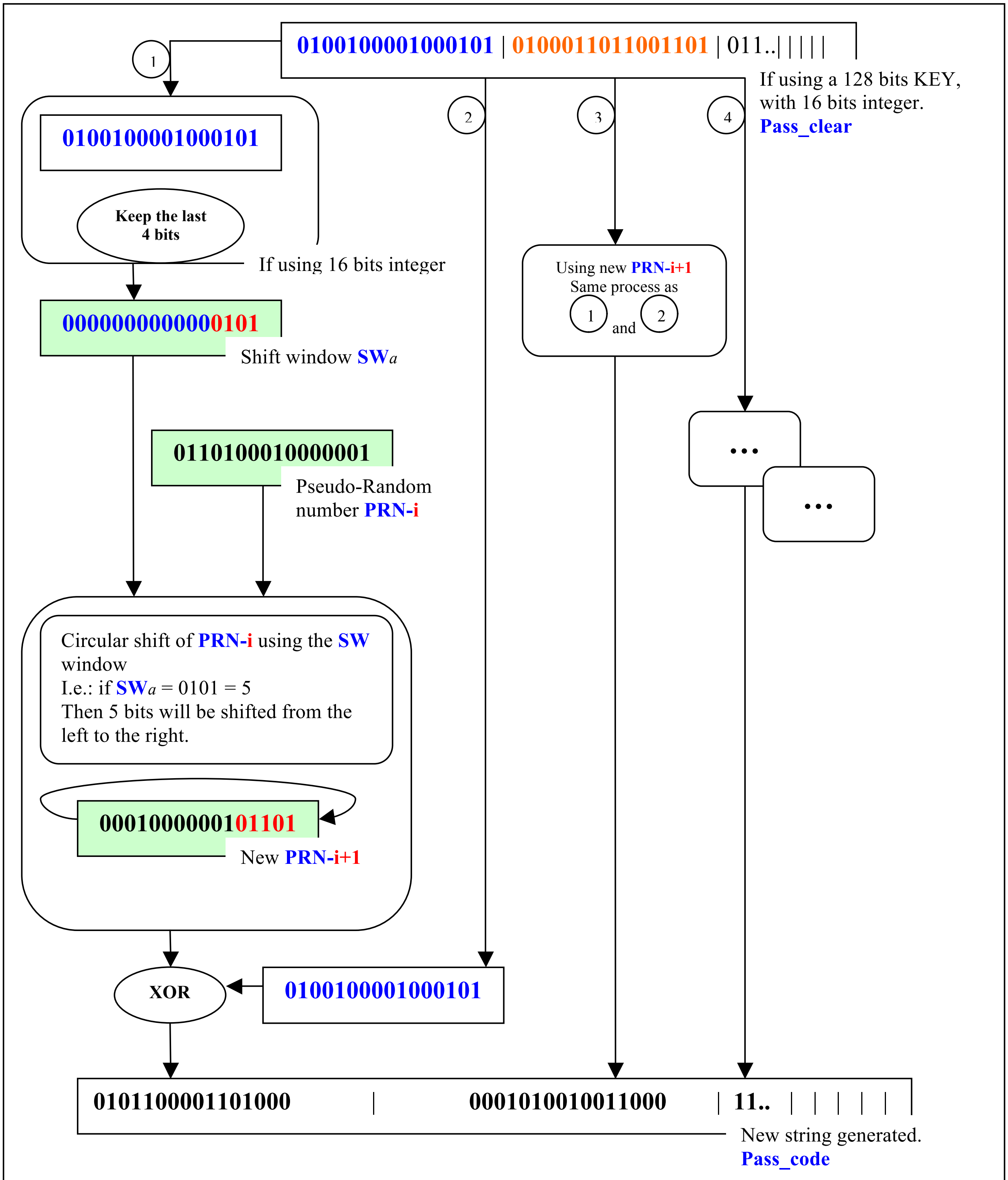


III.1.3 Initial Scrambling

See *f.4.1 - Add() - Random Generation*
and *f.4.2 - Add() - Pass_code Generation*



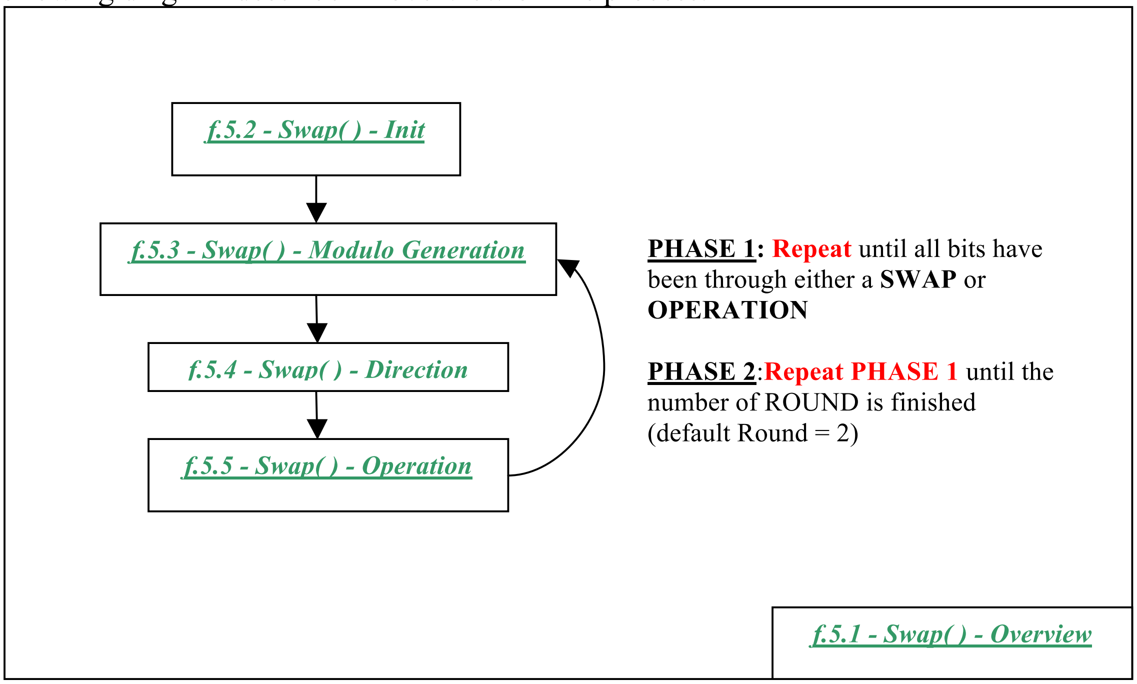
f.4.1 - Add() - Random Generation



f.4.2 - Add() - Pass_code Generation

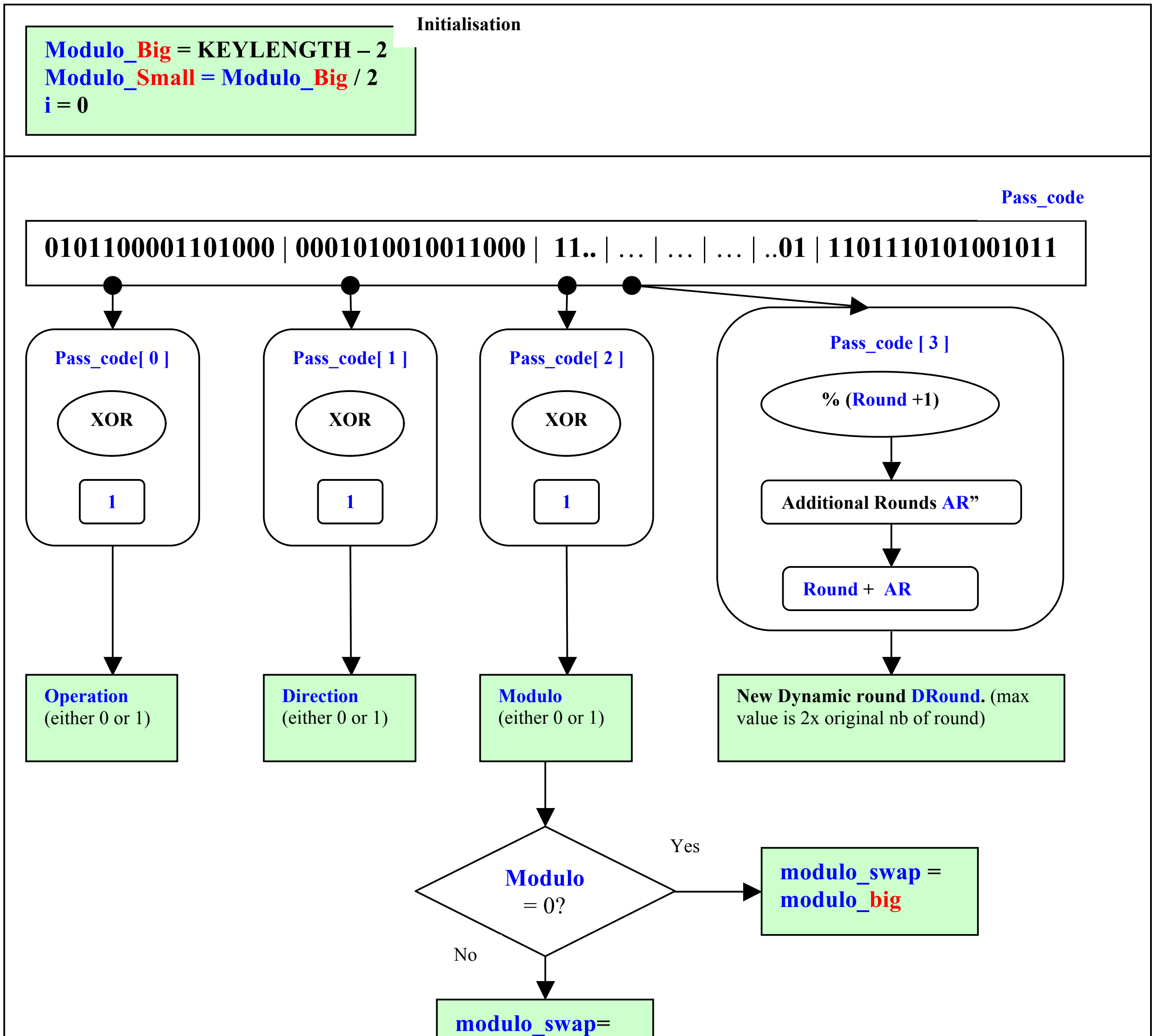
III.1.4 Key Encryption

The following diagram describe an overview of this process:

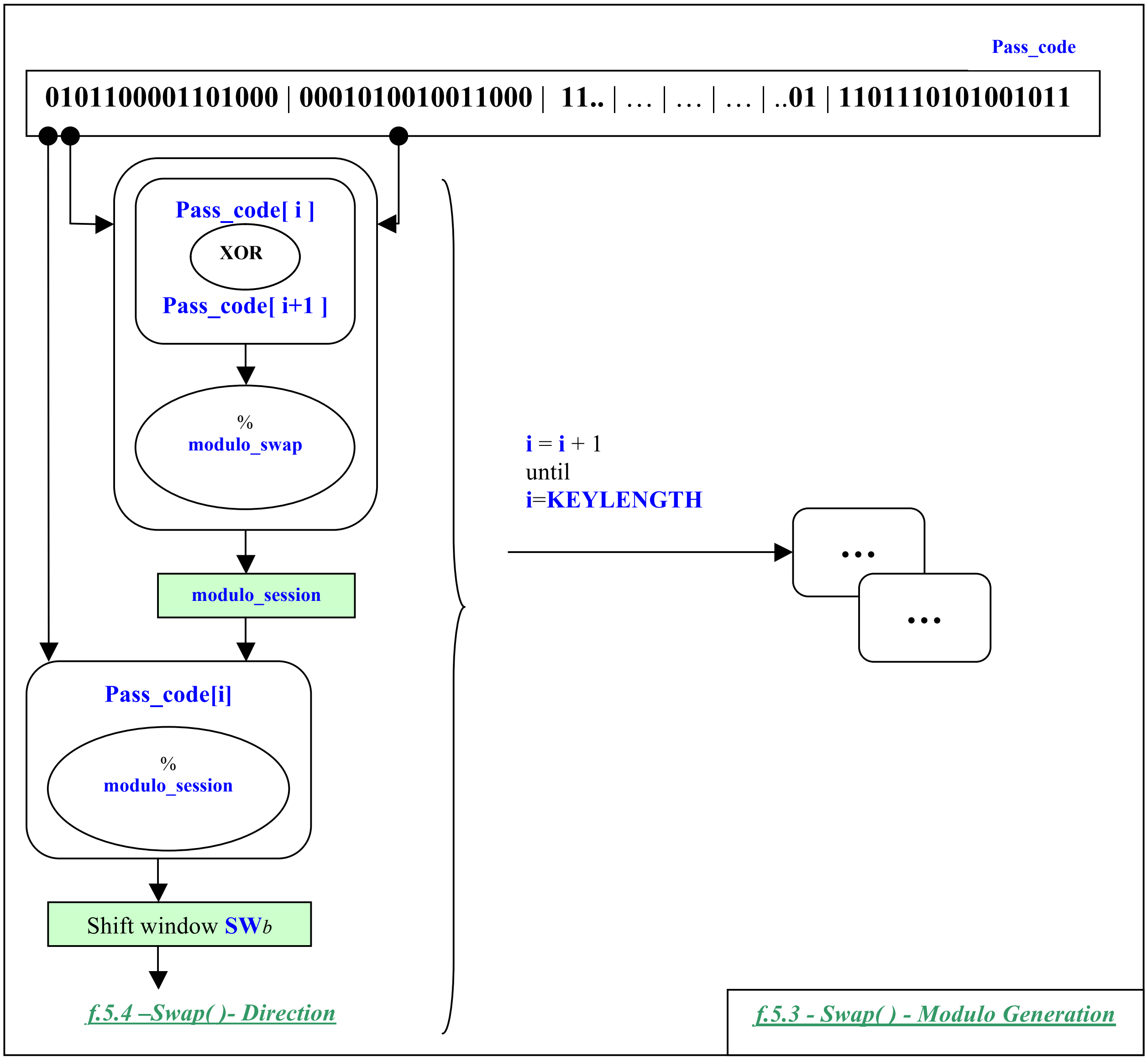


The following 4 diagrams describe the detailed process:

f.5.2 - Swap( ) – Init
f.5.3 - Swap( ) - Modulo Generation
f.5.4 - Swap( ) – Direction
f.5.5 - Swap( ) - Operation

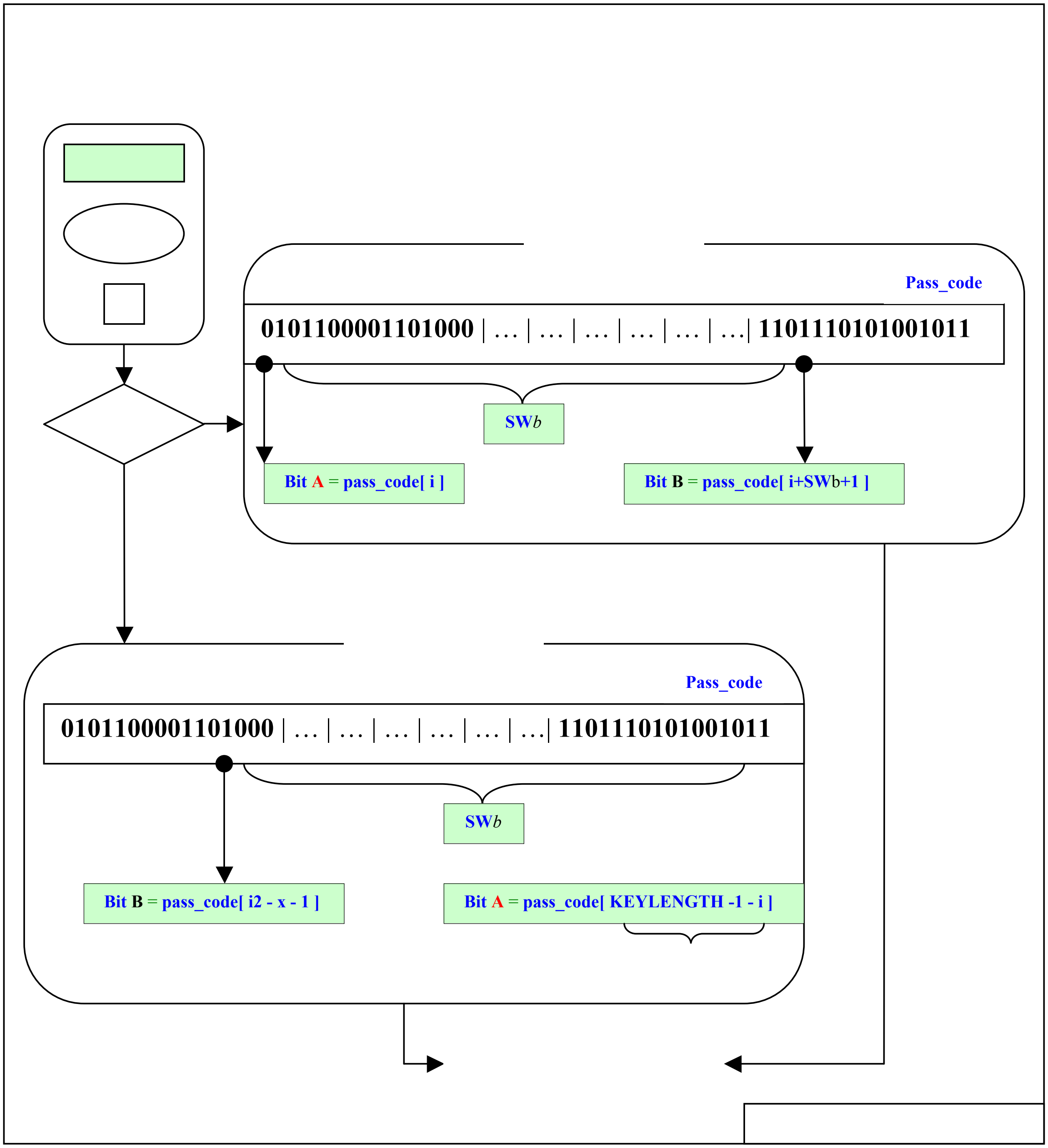


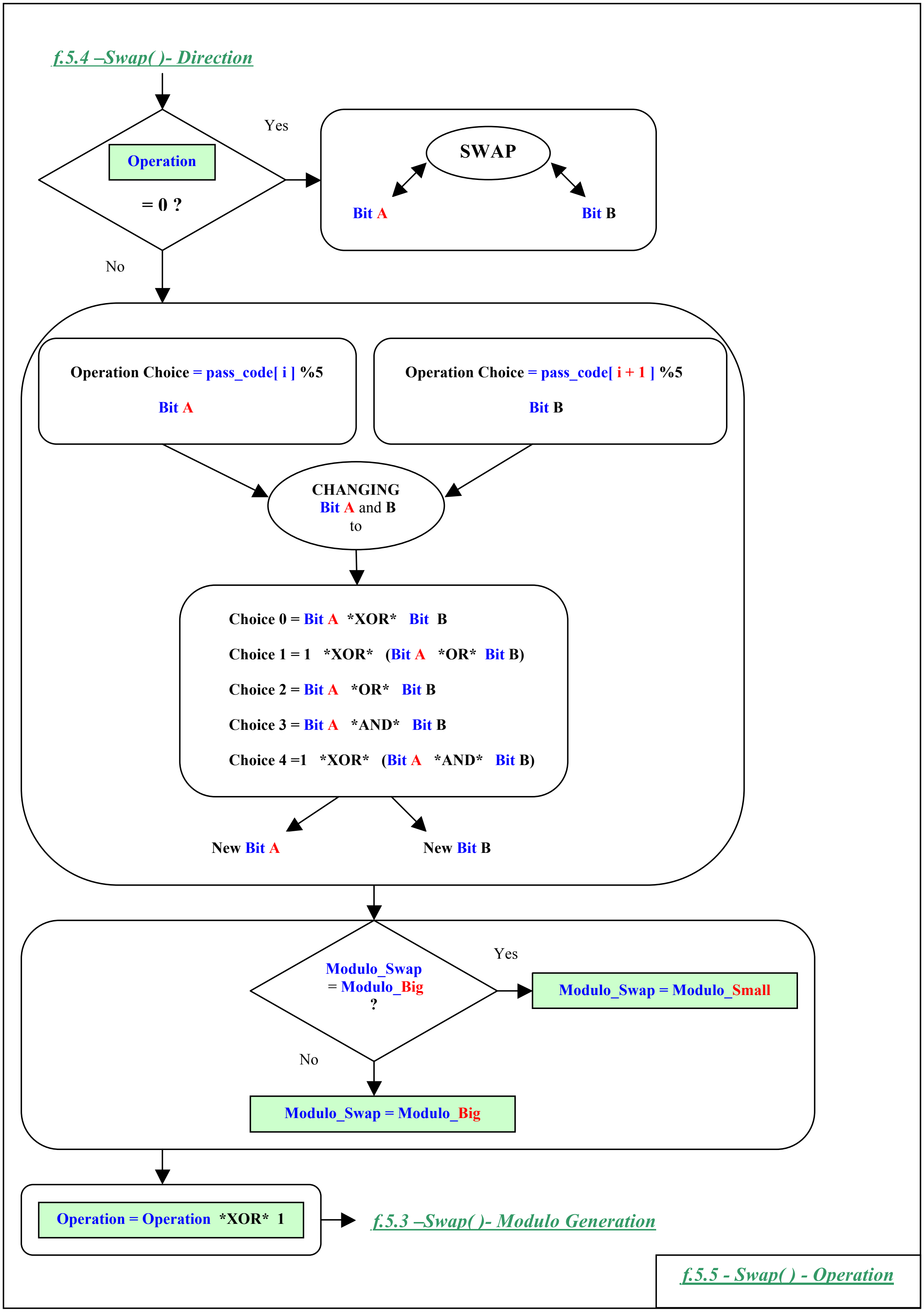
f.5.2 - Swap( ) - Init



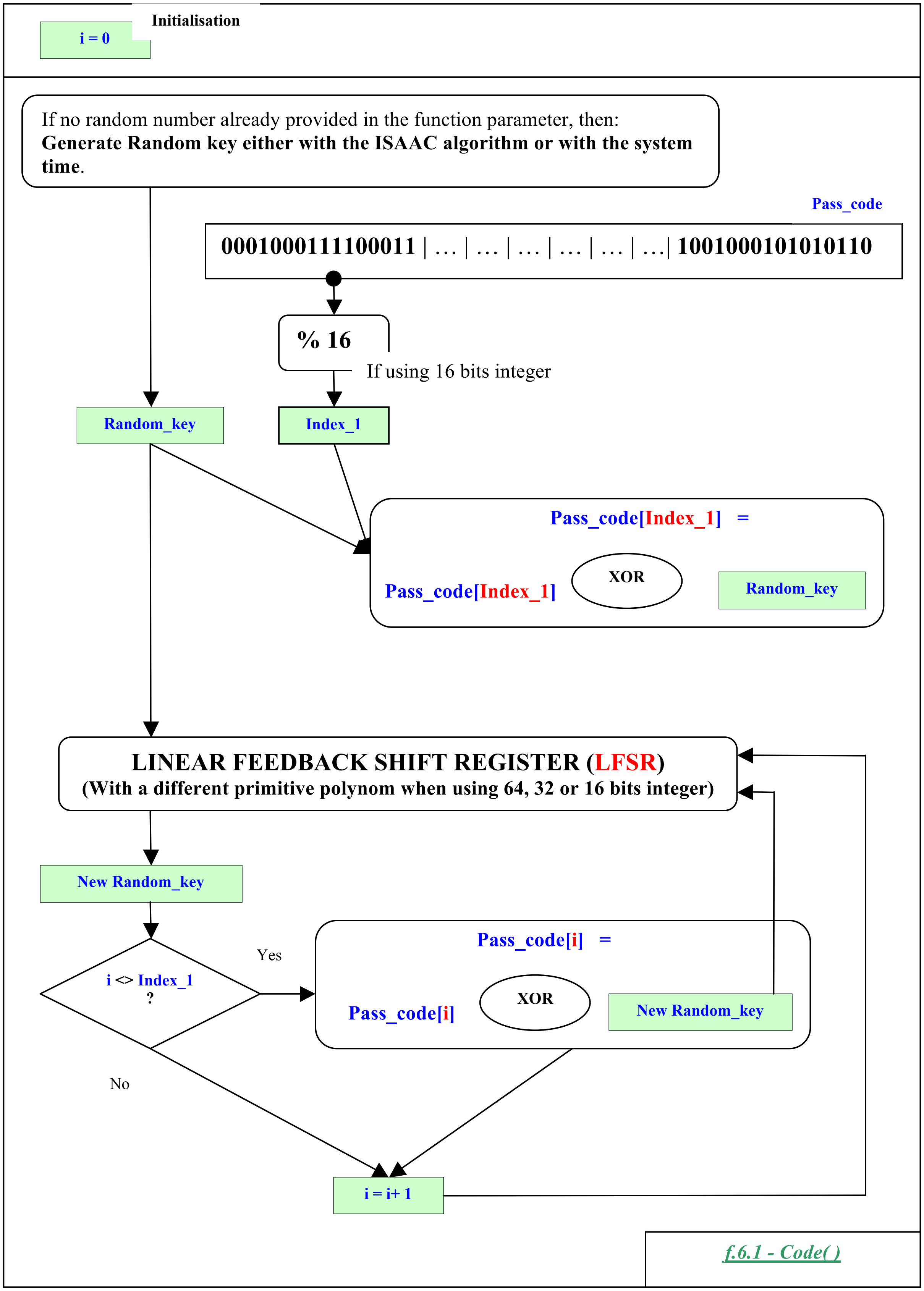
f.5.4 –Swap( )- Direction

f.5.3 - Swap( ) - Modulo Generation


Pass_code
0101100001101000 | … | … | … | … | … | …| 1101110101001011
SW*b*
Bit A = pass_code[i]
Bit B = pass_code[i+SWb+1]
Pass_code
0101100001101000 | … | … | … | … | … | …| 1101110101001011
SW*b*
Bit B = pass_code[i2 - x - 1]
Bit A = pass_code[KEYLENGTH -1 - i]


f.5.4 –Swap( )- Direction
Operation
= 0 ?
Yes
SWAP
Bit A
Bit B
No
Operation Choice = pass_code[i] %5
Bit A
Operation Choice = pass_code[i + 1] %5
Bit B
CHANGING
Bit A and B
to
Choice 0 = Bit A *XOR* Bit B
Choice 1 = 1 *XOR* (Bit A *OR* Bit B)
Choice 2 = Bit A *OR* Bit B
Choice 3 = Bit A *AND* Bit B
Choice 4 =1 *XOR* (Bit A *AND* Bit B)
New Bit A
New Bit B
Modulo_Swap
= Modulo_Big
?
Yes
Modulo_Swap = Modulo_Small
No
Modulo_Swap = Modulo_Big
Operation = Operation *XOR* 1
f.5.3 –Swap( )- Modulo Generation
f.5.5 - Swap( ) - Operation

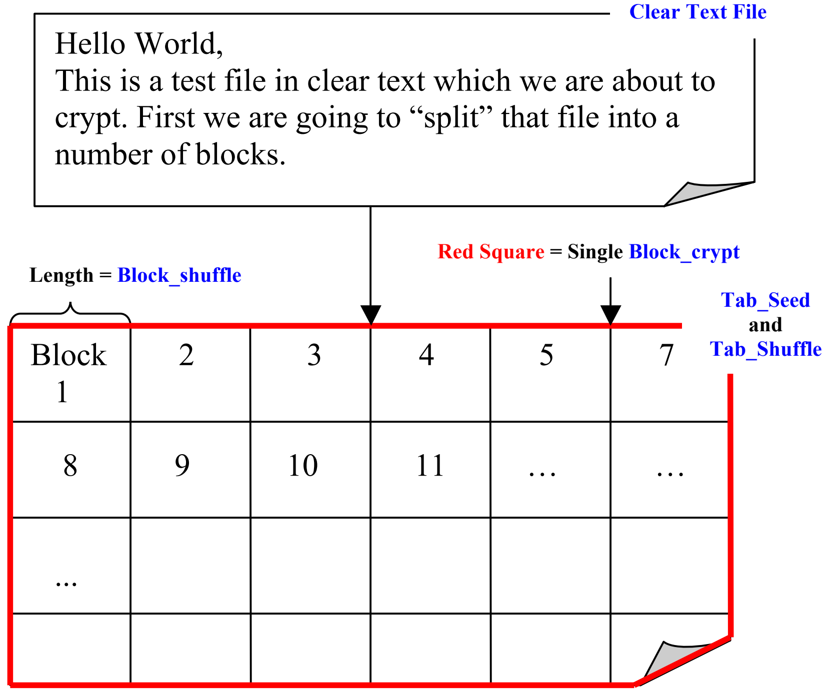
III.1.5 Final Scrambling

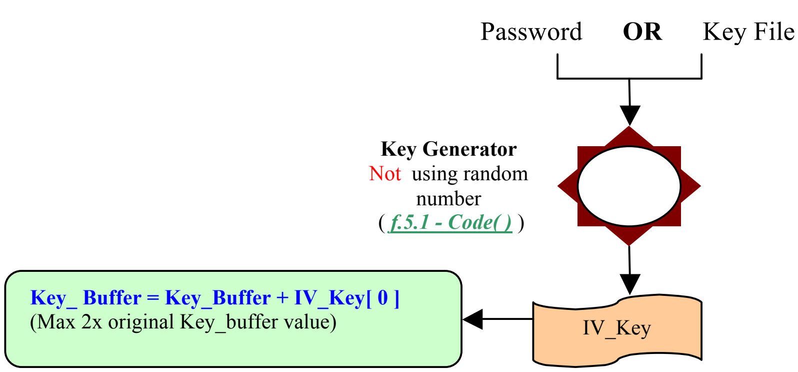


III.2 File Encryption Details

III.2.1 Initialisation

See *f.7.1 – file_crypt() - Init* and *f.7.2.1 – file_crypt() –Seed() - Random Generation*





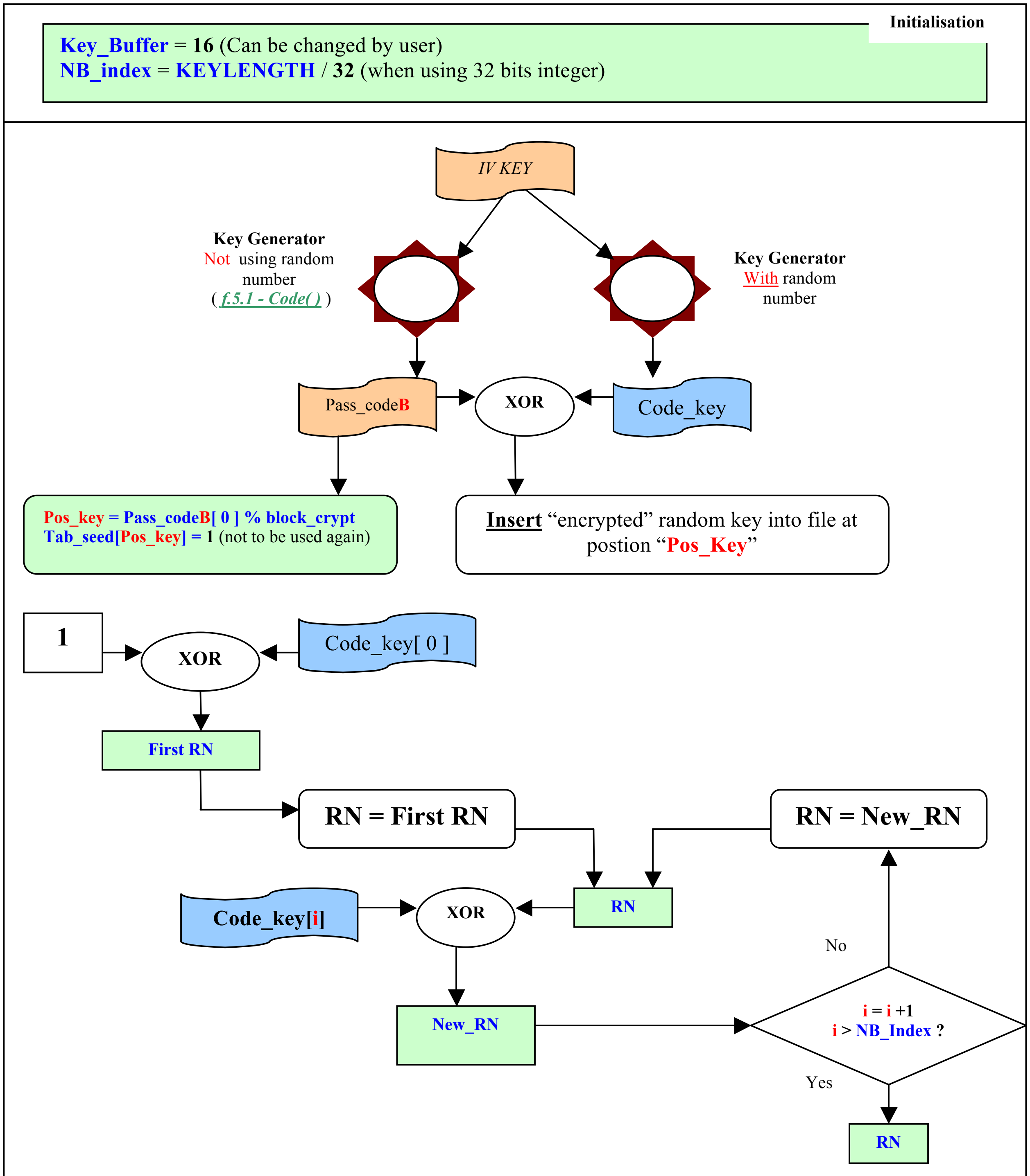
Initialisation

i = **Pass_code[1]**
Dynamic_shuffle = IV_Key[i] % 32 (if using 32 bits integer)
Block_shuffle = Dynamic_shuffle + Block_shuffle
Block_crypt = File length in bytes

Block_shuffle is define with the following rules:

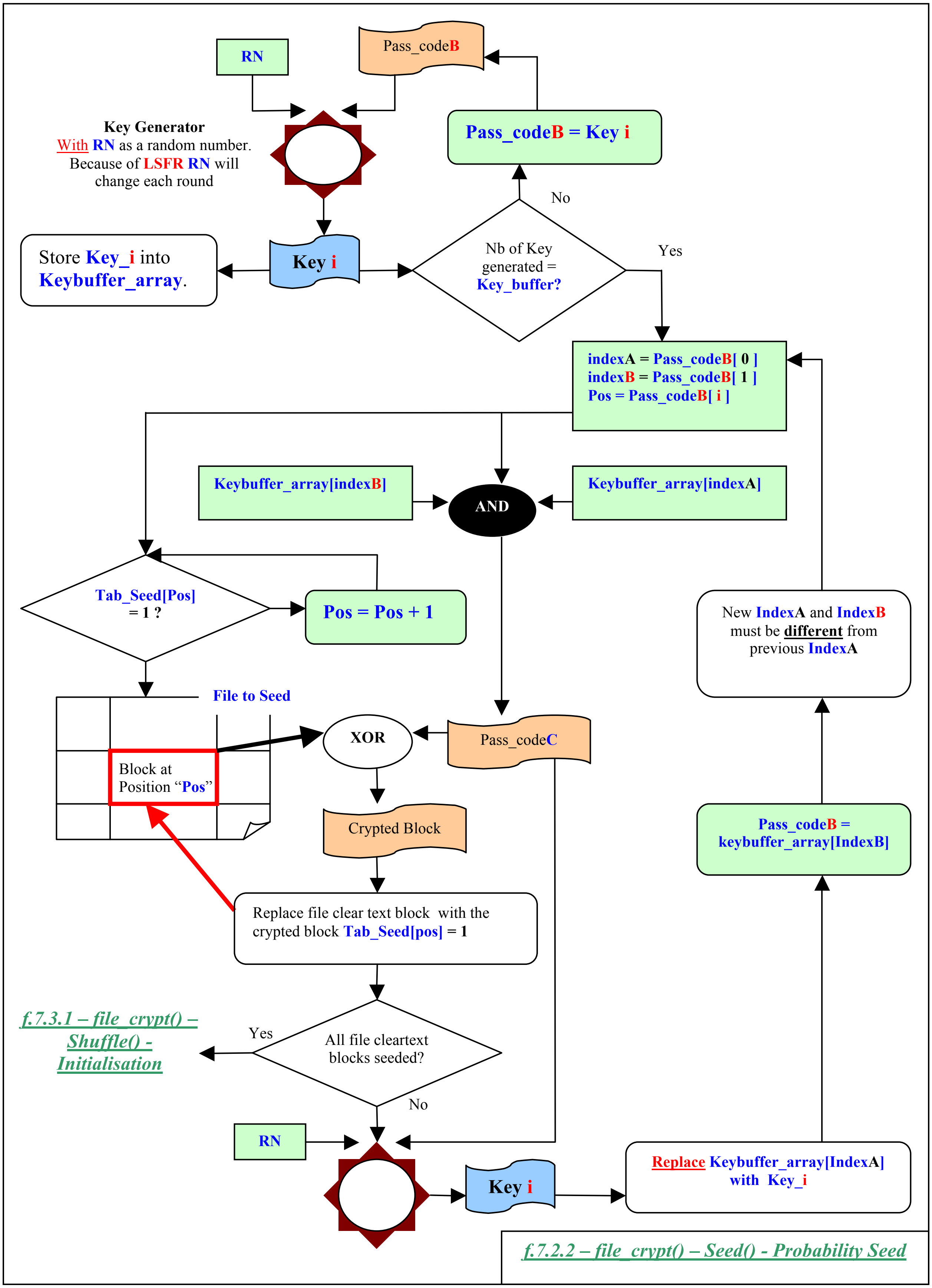
a) It must be a multiple of 4 bytes (if using 32 bits integer)
b) Block_crypt / block_shuffle >= 6

f.7.1.1 – file_crypt() - Init



f.7.2.1 – file_crypt() –Seed() - Random Generation

III.2.2 Seeding



III.2.3 Shuffling

See *f.7.3.1 – file_crypt() – Shuffle() - Initialisation*
And *f.7.3.2 – file_crypt() – Shuffle() - Position & Operation*
And *f.7.3.3 – file_crypt() – Shuffle() –Last 2 blocks*

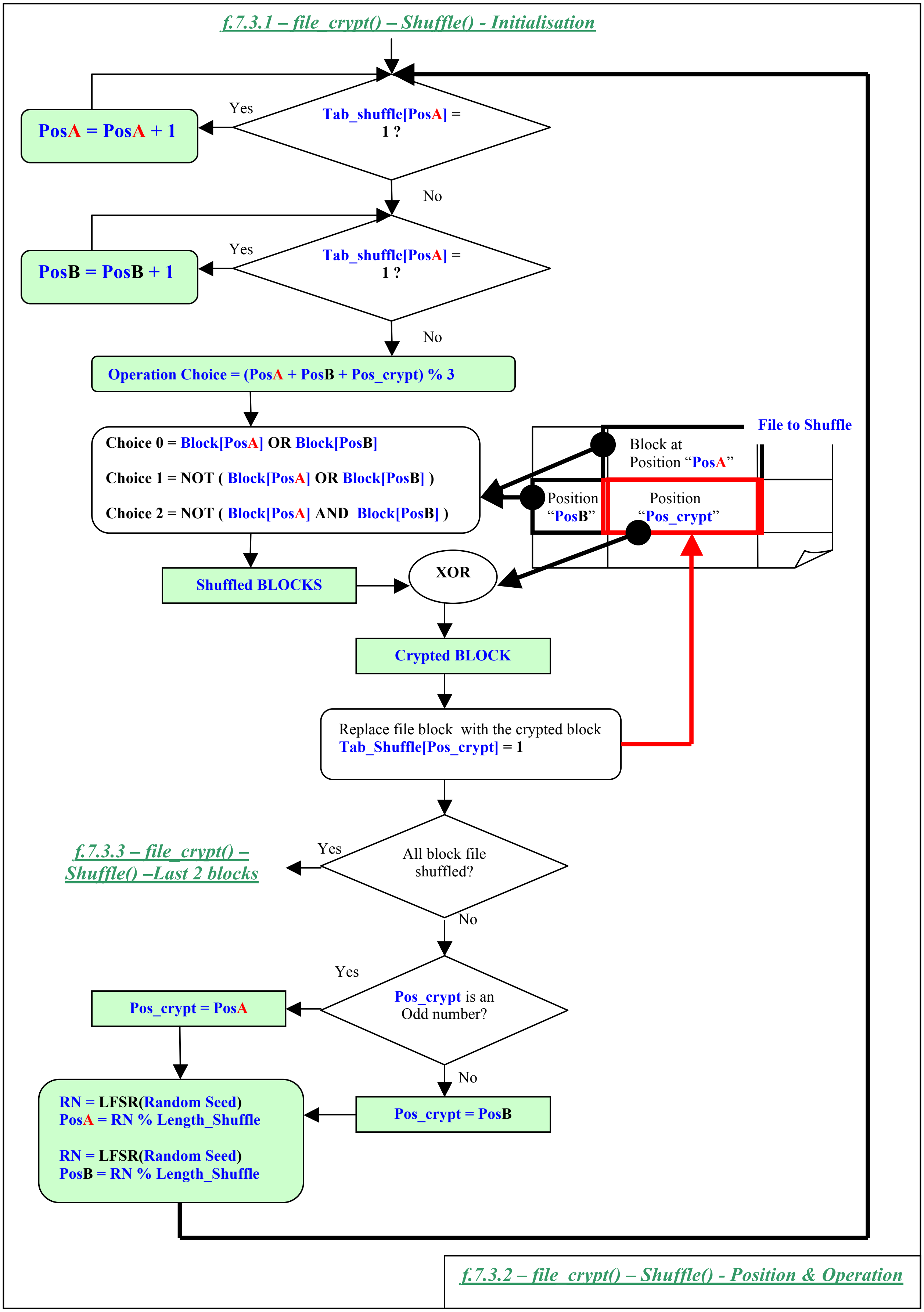
Initialisation

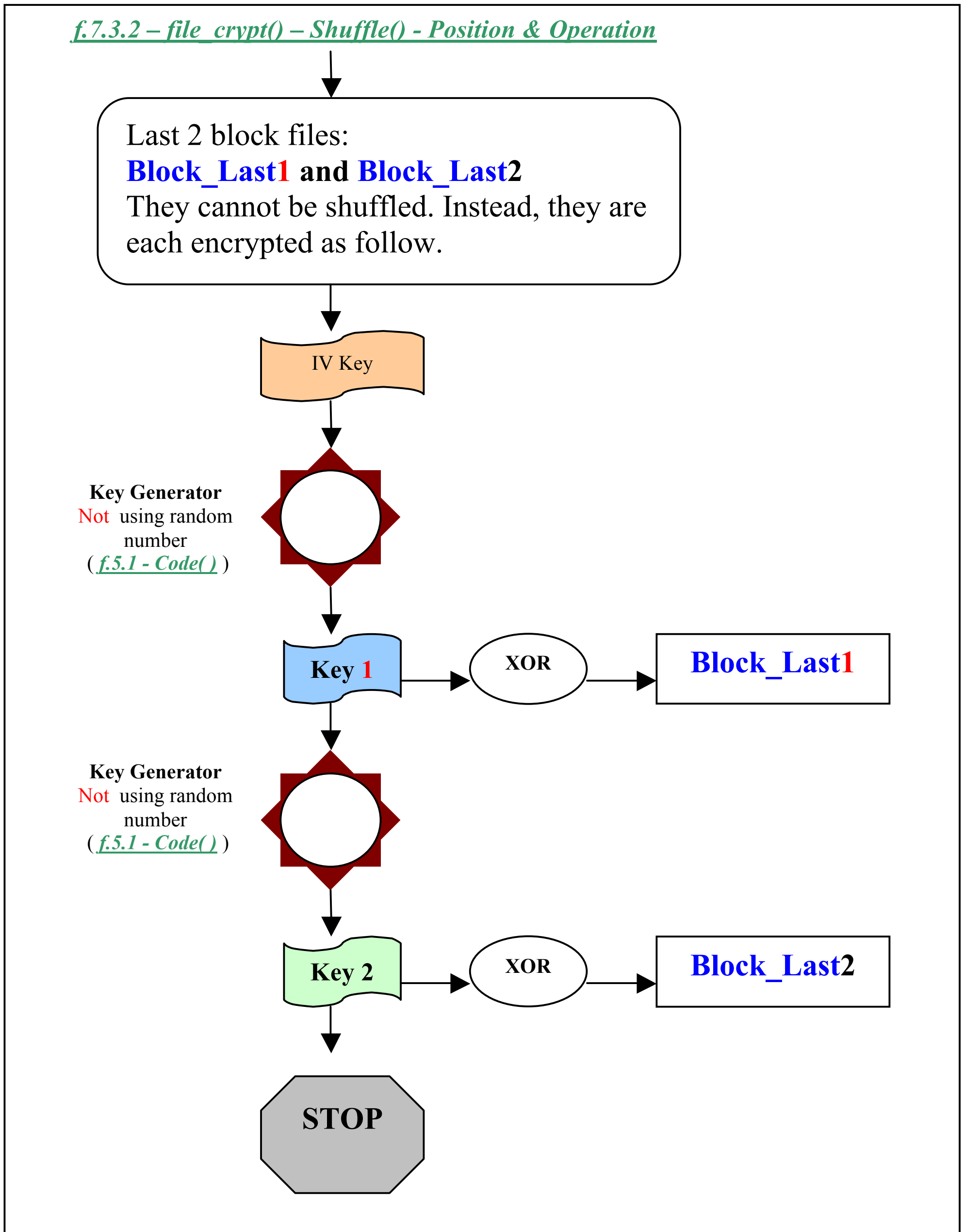
index = IV_Key[0]
PosA = IV_Key [Index]
PosB = IV_Key [Index + 1]
Random_seed = IV_Key [Index + 2]
Pos_crypt = "last block" (this is because the last block may have a variable size)
Length_shuffle = Length_file / Block_shuffle (number of shuffle block in the file to crypt)

↓

f.7.3.2 – file_crypt() – Shuffle() - Position & Operation

f.7.3.1 – file_crypt() – Shuffle() - Initialisation

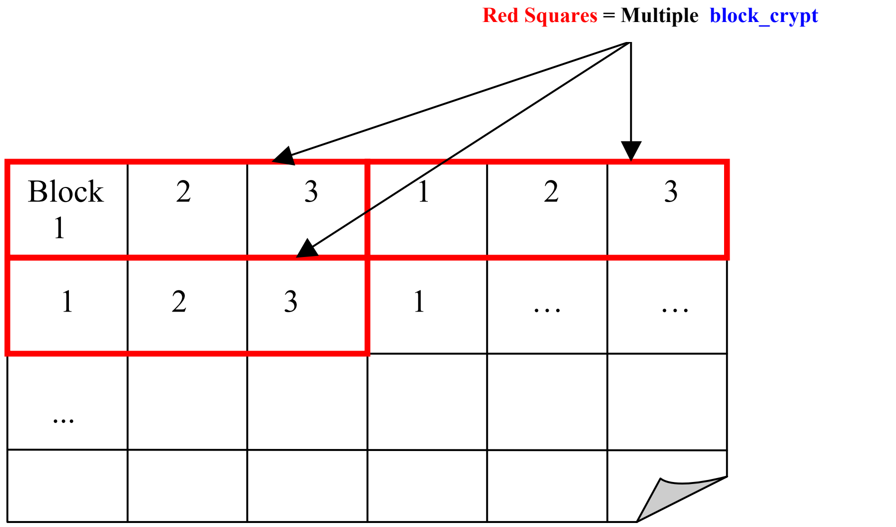

f.7.3.1 – file_crypt() – Shuffle() - Initialisation
Tab_shuffle[PosA] = 1 ?
Yes
PosA = PosA + 1
No
Tab_shuffle[PosA] = 1 ?
Yes
PosB = PosB + 1
No
Operation Choice = (PosA + PosB + Pos_crypt) % 3
Choice 0 = Block[PosA] OR Block[PosB]
Choice 1 = NOT (Block[PosA] OR Block[PosB])
Choice 2 = NOT (Block[PosA] AND Block[PosB])
File to Shuffle
Block at Position "PosA"
Position "PosB"
Position "Pos_crypt"
Shuffled BLOCKS
XOR
Crypted BLOCK
Replace file block with the crypted block
Tab_Shuffle[Pos_crypt] = 1
All block file shuffled?
Yes
f.7.3.3 – file_crypt() – Shuffle() –Last 2 blocks
No
Pos_crypt is an Odd number?
Yes
Pos_crypt = PosA
No
Pos_crypt = PosB
RN = LFSR(Random Seed)
PosA = RN % Length_Shuffle
RN = LFSR(Random Seed)
PosB = RN % Length_Shuffle
f.7.3.2 – file_crypt() – Shuffle() - Position & Operation



f.7.3.3 – file_crypt() – Shuffle() –Last 2 blocks

III.2.4 Alternatives

All the KD highlighted in the above steps can be changed to static values, enable or disable. The same is true for the size of the "block_crypt" as shown in ***f.7.4 – Alternative Block Crypt size***. This means that all the above steps can either be conducted across the entire file or within smaller "working blocks".



f.7.4 – Alternative Block Crypt size