Chapter 1

INTRODUCTION

1.1 eSTREAM Project

eSTREAM, a multi-year project coordinated by ECRYPT, came to an end in April 2008. This project is dedicated to promoting the design of new stream ciphers. The project finished with the publication of a portfolio of new stream ciphers. Four of the proposals in the final portfolio were suited to fast encryption in software which is called Profile-1 [1].

The portfolio was revised in September 2008, after the announcement of cryptanalytic results against one of the algorithms, and since then has been revisited periodically as the algorithms have matured. The current 2012 eSTREAM portfolio contains seven algorithms. Among them, four algorithms are shortlisted as the finalists for software applications and remaining are for hardware applications. They are HC-128, Rabbit, Salsa20/12 and SOSEMANUK (in alphabetical order) for software applications [1].

Stream cipher HC-128 is the simplified version of HC-256 for 128-bit security. HC-128 is a simple, secure, software-oriented cipher and it is freely-available [1]. HC-128 consists of two secret tables, each one with 512 32-bit elements. At each step, we update one element of a table with non-linear feedback function. All the elements of the two tables get updated every 1024 steps. At each step, one 32-bit output is generated from the non-linear output filtering function. HC-128 is suitable for the modern superscalar microprocessors [1].

Rabbit is a stream cipher algorithm that has been designed for high performance in software implementation. Both key setup and encryption are very fast, making the algorithm particularly suited for all applications where large amounts of data or large numbers of data packages have to be encrypted. Examples: - Server-side encryption, Multimedia encryption, Hard-disk encryption, and encryption on limited-resource devices etc. [2, 3].

Salsa20/r is a software-oriented stream cipher by Bernstein. During the operation of the cipher, the key, a 64-bit nonce (unique message number), a 64-bit counter, and four 32-bit constants are used to construct the 512-bit initial state of the cipher [1]. After r iterations of the Salsa20/r round function, the updated state is used as a 512-bit output. Each such output block is an independent combination of the key, nonce, and counter and, since there is no

chaining between blocks, the operation of Salsa20/r resembles the operation of a block cipher in counter mode. This stream cipher is used to do fast encryption for huge amount of data.

SOSEMANUK has a variable key length, ranging from 128 to 256 bits, and takes an initial value of 128 bits. However, for any key length the cipher is only claimed to 128-bit security. SOSEMANUK uses similar design principles to the stream cipher SNOW 2.0 [5] and the block cipher SERPENT [4].

SOSEMANUK aims to overcome potential structural weaknesses in SNOW 2.0 while providing better performance by decreasing the size of the internal state. As for SNOW 2.0, SOSEMANUK has two main components: A linear feedback shift register(LFSR) and a finite state machine (FSM). The LFSR operates on 32-bit words and has length 10. At every clock a new 32-bit word is computed. The FSM has two 32-bit memory registers. This algorithm is also used for fast software encryptions [1].

1.2 Motivation

As being the stream cipher, a branch of symmetric cryptography, I thought that it also should have significant features, results as the block cipher. Keeping this in mind, while researching, finding and studying the papers and the contents related to this field, I came to know about eSTREAM project run by ECRYPT from Europe. In this project, there were several techniques or algorithms participated in the competition held during 2004. Finally, four of them were selected as eSTREAM finalist by the end of 2008 AD [1].

They are HC-128, Rabbit, Salsa20/12 and SOSEMANUK. Among them, Rabbit is the oldest and patented stream cipher. It is seen that the HC-128 has shown the best performance among others. Similarly, high security can be achieved by SOSEMANUK since it has entirely different mechanism for setting IV and getting unique key.

Performances of these algorithms are depending upon number of parameters. One can easily ask the question what the performance of these techniques will be if different sizes of message are input to them considering other factors constant. This question is the sign of motivation in this thesis that the performance of the algorithms is observed by inputting small to large (variable) sizes of textual message.

1.3 Objective

- To implement and analyze the performance of eSTREAM cipher finalists such as HC-128, Rabbit, Salsa20/12 and SOSEMANUK using different parameters like different size of messages.
- To calculate cycle/Byte performance.

1.4 Thesis Organization

The rest of the content in this study is organized into subsequent five chapters.

Chapter 2 provides background study required for dissertation. In this chapter the problem of different eSTREAM cipher algorithms are mentioned, problem statement is formulated and how this study response those issues is mentioned.

Chapter 3 contains previous literature related to this work in detail under literature review. Moreover, it contains details of each algorithms of eSTREAM project.

Chapter 4 provides an implementation overview of different eSTREAM cipher finalists in Java Programming language integrated in NetBeans 8.0.2 version. The implementation details with major coding functions are provided in this chapter.

Chapter 5 includes the analysis of time required for creating key streams and encrypting messages and finally with the help of average time needed for encrypting for all candidates algorithm, cycle per byte is calculated. The result of the study is shown in tabular form as well as in graphs.

Finally, the concluding remarks and further recommendations as well as future works are outlined in chapter 6.

Chapter 2

THESIS BACKGROUND

Today, the internet has virtually become the way of doing business as it offers a powerful widespread medium of commerce and enables greater connectivity of disparate groups throughout the world. So, it may have many risks like loss of privacy, loss of data integrity, denial of service and identify spoofing. To the solution of these threads in internet many secure cryptographic algorithms are needed for providing services such as confidentiality, data integrity and authentication to handle packets which may vary in size over a large range. The size of the message has a significant impact on the performance of such algorithms. Hence the messages have to be prepared by padding the required amount of zero bits to get an integer number of blocks. This process becomes a considerable overhead when the short messages are more dominant in the message stream. In this thesis, for simplicity communicating parties are named as Alice and Bob where as attacker named as Darth.

2.1 Problem Definition

Before eSTREAM project was lunched, there was an another project called NESSIE which stands for "New European Schemes for Signature, Integrity and Encryption". In that project, several techniques and algorithms were submitted for selecting fast stream cipher algorithm. None of the stream ciphers, submitted to NESSIE, were selected because everyone felt to cryptanalysis. This surprising result led to the eSTREAM project [6].

There were eight algorithms submitted in eSTREAM project namely CryptMT, Dragon, HC-128, LEX, NLS, Rabbit, Salsa20/12 and SOSEMANUK [7]. Among them, CryptMT, Dragon, LEX as well as NLS were actually slower as compared to other remaining and performance was not remarkable. Therefore, Rabbit, Salsa20/12, HC-128 and SOSEMANUK were selected in eSTREAM portfolio phases 3. These are stream cipher finalists announced by ECRYPT in eSTREAM project in 2012 [1]. Time is the key factor for encryption as well as decryption in cryptography. So that, the best and the fastest algorithm among these stream cipher finalists with good performance will be purposed in this study.

2.2 Background Study

Since all the study require the basic terms and terminology related to that study in this context, basic study related to this work are outlined in the following sections.

2.2.1 Cryptography

Cryptography is art of protecting information by encrypting it into an unreadable format, called cipher text. Only those who possess a secret key can decipher (or decrypt) the message into plaintext. Encrypted messages can sometimes be broken by cryptanalysis, also called code breaking, although modern cryptography techniques are virtually unbreakable. Cryptography enables one to store sensitive information or transmit it across insecure networks so that it cannot be read by anyone except the intended recipient. While cryptography is the science of securing data, cryptanalysis is the science of analyzing and breaking secure communication. Classical cryptanalysis involves an interesting combination of analytical reasoning, application of mathematical tools, pattern finding, patience, determination, and luck. Cryptanalysts are also called attackers and represented as Darth. Cryptology embraces both cryptography and cryptanalysis. The modern cryptography can be divided into two main branches [8, 21]:

- Symmetric Cryptography, where the same key is used to encrypt a message and decrypt data.
- Asymmetric cryptography, where two different keys are used for encryption and decryption.

2.2.1.1 Symmetric Cryptography

Symmetric cryptography is a form of cryptosystem in which encryption and decryption are performed using the same key. It is also known as private key cryptosystem. Symmetric cryptosystem was the only type of encryption technique in use prior to the development of public key cryptosystem. Which can be defined as: Let M denotes the set of all possible plaintext messages, C the set of all possible cipher text, K the set of all possible keys, k: $M \rightarrow C$ is the encryption function, and k: $C \rightarrow M$, is decryption function, such that $k(k(m)) = m$ for all $m \in M$ and $k \in K$. In this cryptosystem, sender and receiver have to initially agree upon a secret key $k \in K$. After that, whenever sender wishes to send a message $m \in M$ to receiver, sender sends the cipher text $C = k(m)$ to receiver, from which receiver can recover m by applying the decryption function as $m = k(C)$ [27]. The notion of private key cryptosystem is depicted in Figure 2.1.

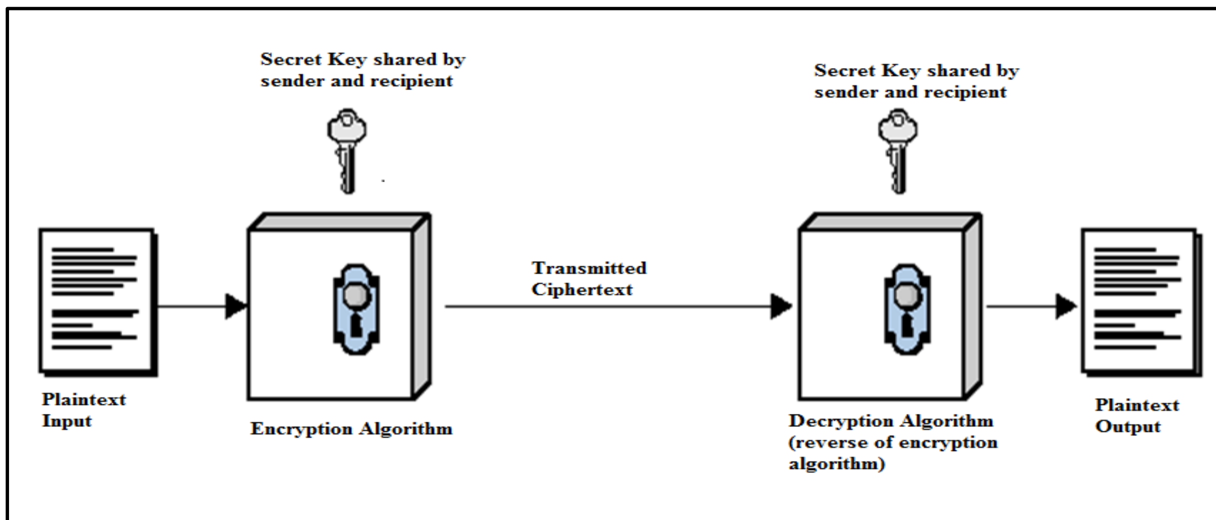

Figure-2.1: Simplified Model of Symmetric Encryption [19].

The effectiveness of private key cryptosystems relies on the requirement of strong encryption algorithm which would be like the algorithm to be such that an opponent who knows the algorithm and has access to one or more cipher texts would be unable to decipher the cipher text or find out the key and another requirement is that sender and receiver must have obtained copies of the secret key in a secure fashion and must keep the key secure. Modern techniques used in private key cryptosystem are XOR Cipher, Rotation Cipher, Substitution Cipher: S-box, Transposition Cipher: Data Encryption Standard (DES), Advanced Encryption Standard (AES) and so on. As mentioned in [19], private key cryptosystems have numerous limitations which are outlined below:

- **Key distribution problem:** Two parties that want to communicate each other need to set up a shared secrete key before starting communicate over an insecure channel.
- **Key management problem:** Every pair of users must share a secret key leading to a total of n*(n-1)/2 keys. Where n be the users in a network. If n is large, then the number of keys become unmanageable and traffic in network may be increased. It makes difficult to manage key.
- **No signatures possible:** A digital signature is an authentication mechanism that enables the creator of the message to attach a special token that acts as a signature. A digital signature allows the receiver of a message to convince any third-party that the message in fact originated from the sender.

2.2.1.2 Asymmetric Cryptography

Asymmetric encryption is a form of cryptosystem in which encryption and decryption are performed using the different key- one a public key and one a private key. It is also known as public-key encryption. Asymmetric encryption transforms plaintext into cipher text using a one of two keys and an encryption algorithm. Using the paired key and a decryption algorithm, the plaintext is recovered from the cipher text.

Asymmetric encryption can be used for confidentiality, authentication, or both. The most widely used public-key cryptosystem is RSA. Public-key algorithms are based on mathematical functions rather than on substitution and permutation. More important, public-key cryptography is asymmetric, involving the use of two separate keys, in contrast to symmetric encryption, which uses only one key. The use of two keys has profound consequences in the areas of confidentiality, key distribution, and authentication [19].

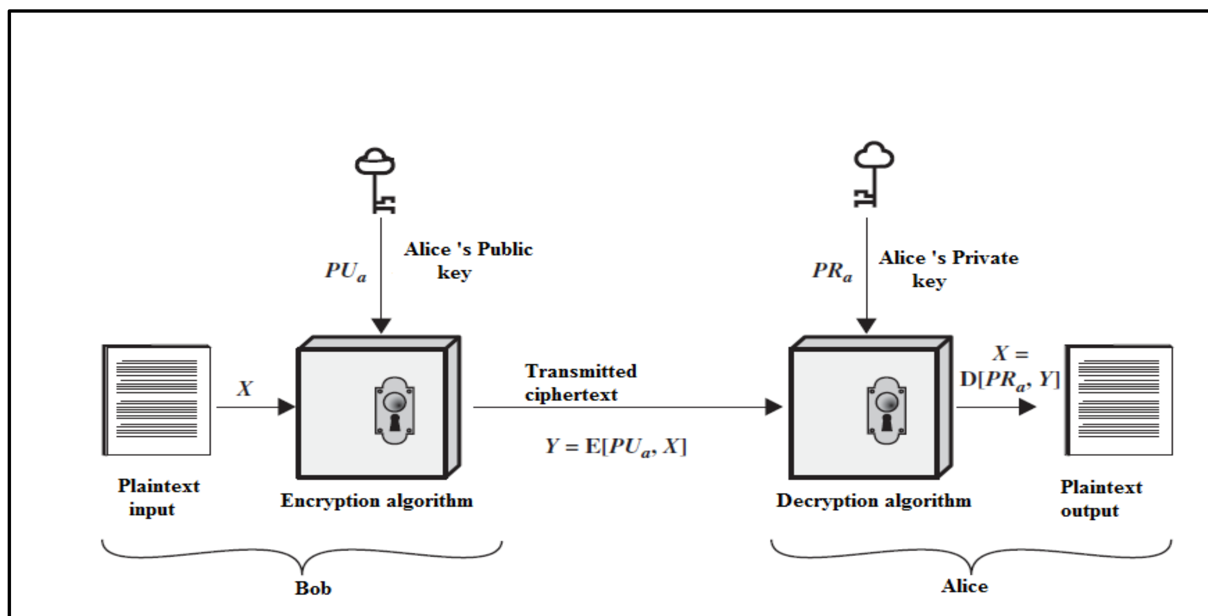

Figure-2.2: Encryption with Public Key [19].

Chapter 3

LITERATURE REVIEW

3.1 Stream Ciphers

Stream ciphers are an important class of encryption algorithms and are defined as the ciphers in which plain texts are encrypted by XORing between secret key and plain texts to obtain ciphers. They encrypt individual characters (usually binary digits) of a plaintext message one at a time. Stream ciphers are generally faster than block ciphers in software as well as hardware applications [14]. They are more appropriate in some telecommunications applications, where buffering is limited or characters must be individually processed as they are received. If it is observed at the types of cryptographic algorithms that exist in a little bit more detail, it can be seen that the symmetric ciphers can be divided into stream ciphers and block ciphers, as shown in Fig-3.1.

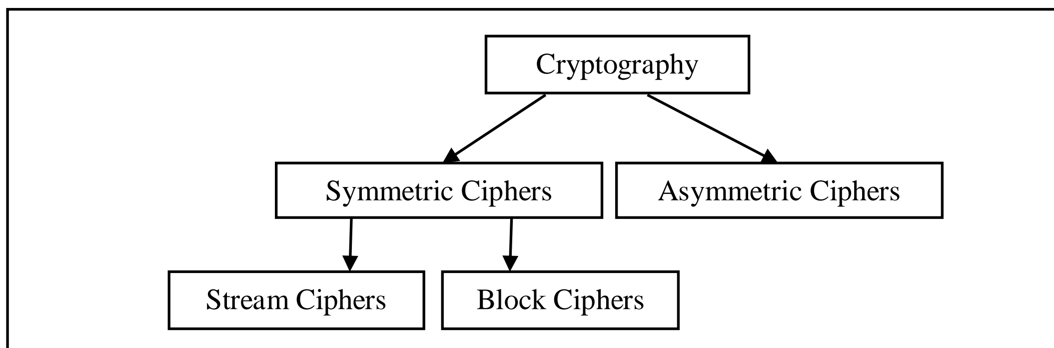

Figure-3.1: Showing Cryptographic Branches [14].

3.1.1 Stream and Block Ciphers

Symmetric cryptography is split into block ciphers and stream ciphers, which are easy to distinguish. Figure 3.2 depicts the operational differences between stream [Fig. 3.2(a)] and block [Fig. 3.2(b)] ciphers when we want to encrypt *b* bits at a time, where *b* is the width of the block cipher.

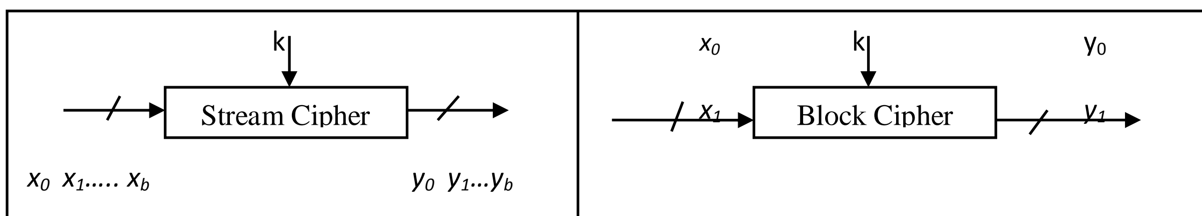

Figure-3.2: Principles of Encrypting *b* bits with a stream (a) and a block (b) cipher [14].

Stream ciphers encrypt bits individually. This is achieved by adding a bit from a key stream to a plaintext bit. There are synchronous stream ciphers where the key stream depends only on the key, and asynchronous ones where the key stream also depends on the cipher text. If the dotted line in Fig. 3.3 is present, the stream cipher is an asynchronous one. Most practical stream ciphers are synchronous ones. An example of an asynchronous stream cipher is the cipher feedback (CFB) mode [14].

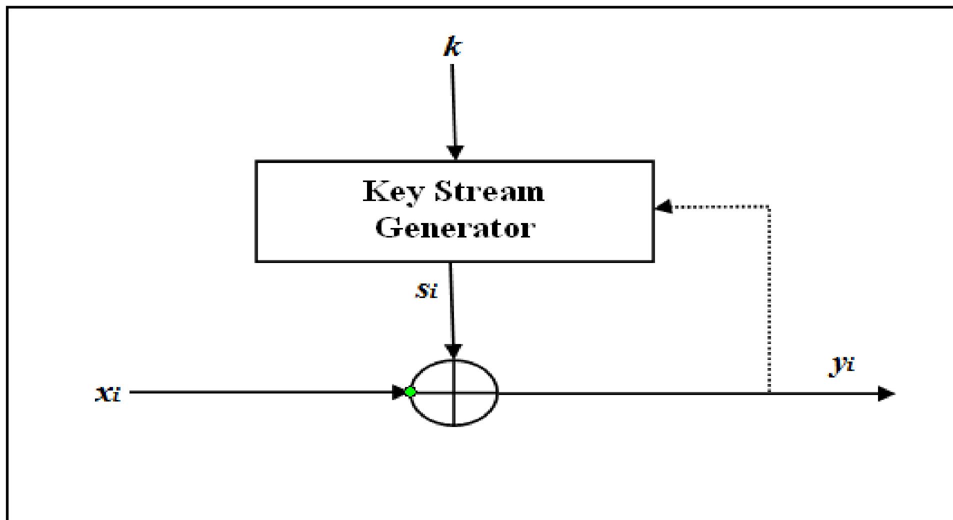

Figure-3.3: Showing Asynchronous Stream Cipher Generation [14].

Block ciphers encrypt an entire block of plaintext bits at a time with the same key. This means that the encryption of any plaintext bit in a given block depends on every other plain text bit in the same block. In practice, the vast majority of block cipher, either have a block length of 128 bits (16 bytes) such as the advanced encryption standard (AES), or a block length of 64 bits (8 bytes) such as the data encryption standard (DES) or triple DES (3DES) algorithm. Some differentiation points between stream ciphers vs. block ciphers are as below:

1. In practice, particularly for encrypting computer communication on the Internet, block ciphers are used more often than stream ciphers.

2. Because stream ciphers tend to be small and fast, they are particularly relevant for applications with little computational resources, e.g., for cell phones or other small embedded devices.

A prominent example for a stream cipher is the A5/1 cipher, which is part of the GSM mobile phone standard and is used for voice encryption. However, stream ciphers are sometimes also used for encrypting Internet traffic, especially the stream cipher RC4.

3. It is assumed that stream ciphers tended to encrypt more efficiently than block ciphers. Efficient for software-optimized stream ciphers means that they need fewer processor instructions (or processor cycles) to encrypt one bit of plaintext.

3.1.2 Encryption and Decryption in Stream Ciphers

Example: Alice wants to encrypt the letter A, where the letter is given in ASCII code. The ASCII value for A is $65_{10} = 1000001_2$. Let's furthermore assume that the first key stream bits are $(s0, \ldots, s6) = 0101100$.

Sender: - Alice, Receiver: - Bob, Attacker: - Darth

$x0, \ldots, x6 = 1000001 = A$

$\oplus$

$s0, \ldots, s6 = 0101100$

$y0, \ldots, y6 = 1101101 = m$

.

.

.

.−−−−m−=−1−10−1−10−1−−→

$y0, \ldots, y6 = 1101101$

$\oplus$

$s0, \ldots, s6 = 0101100$

$x0, \ldots, x6 = 1000001 = A$

According to above mentioned example, bitwise XORing between message and key is carried out for encryption as well as decryption.

3.1.3 Random Numbers, Nonce and OTP in Stream Cipher

Random numbers and its generation play very important role in stream ciphers since generating key is major thing and encryption and decryption is simply the XOR operation. Random number can be generated by three ways: TRNG, PRNG and CSPRNG [14].

True random number generators (TRNGs) are characterized by the fact that their output cannot be reproduced. For instance, if we flip a coin 100 times and record the resulting sequence of 100 bits, it will be virtually impossible for anyone on Earth to generate the same 100 bit sequence. The chance of success is $1/2^{100}$, which is an extremely small probability. TRNGs are based on physical processes. Examples include coin flipping, rolling of dice etc [14].

Pseudorandom number generators (PRNGs) generate sequences which are computed from an initial seed value. Often they are computed recursively in the following way:

$$s0 = seed$$

$$s_{i+1} = f(s_i),\ i = 0,1,\ldots$$

A widely used example is the *rand()* function used in ANSI C. A common requirement of PRNGs is that they possess good statistical properties, meaning their output approximates a sequence of true random numbers. There are many mathematical tests, e.g., the chi-square test for PRNG [14].

Cryptographically secure pseudorandom number generators (CSPRNGs) are a special type of PRNG which possess the following additional property: A CSPRNG is PRNG which is unpredictable. Informally, this means that given n output bits of the key stream s_i, s_{i+1}, . . . , s_{i+n-1}, where n is some integer, it is computationally infeasible to compute the subsequent bits s_{i+n}, s_{i+n+1}, ... A more exact definition is that given n consecutive bits of the key stream, there is no polynomial time algorithm that can predict the next bit s_{n+1} with better than 50% chance of success.

Another property of CSPRNG is that given the above sequence, it should be computationally infeasible to compute any preceding bits s_{i-1}, s_{i-2}, ... Note that the need for unpredictability of CSPRNGs is unique to cryptography. Virtually, in all other situations where pseudorandom numbers are needed in computer science or engineering, unpredictability is not needed [14].

Many aspects of cryptography require random numbers, for example: Key generation, nonce, one-time pads etc. The "quality" of the randomness required for these applications varies. For example, creating a nonce in some protocols needs only uniqueness. On the other hand, generation of a master key requires a higher quality. In cryptography, a nonce is an arbitrary number that may only be used once. It is similar in spirit to a nonce word. They can also be useful as initialization vectors and in cryptographic hash function [14].

In cryptography, the one-time pad (OTP) is an encryption technique that cannot be cracked if used correctly. In this technique, a plaintext is paired with a random secret key (also referred to as *a one-time pad*). Then, each bit or character of the plaintext is encrypted by combining it with the corresponding bit or character from the pad using modular addition. If the key is truly random, is at least as long as the plaintext, is never reused in whole or in part, and is kept completely secret, then the resulting cipher text will be impossible to decrypt or break. It has also been proved that any cipher with the perfect secrecy property must use keys with effectively the same requirements as OTP keys [14].

3.2 Candidate Algorithms

3.2.1 HC-128

Stream cipher HC-128 is the simplified version of HC-256 for 128-bit security [10]. HC-128 is a simple, secure, software-efficient cipher and it is freely-available. HC-128 consists of two secret tables, each one with 512 32-bit elements. At each step, we update one element of a table with non-linear feedback function. All the elements of the two tables get updated every 1024 steps.

At each step, one 32-bit output is generated from the non-linear output filtering function. HC-128 is suitable for the modern superscalar microprocessors. The dependency between operations in HC-128 is very small: three consecutive steps can be computed in parallel. At each step, the feedback and output functions can be computed in parallel. The high degree of parallelism allows HC-128 to run efficiently on the modern processor [10, 15]. An entire diagram for HC-128 algorithm is given as below in Fig. 3.4.

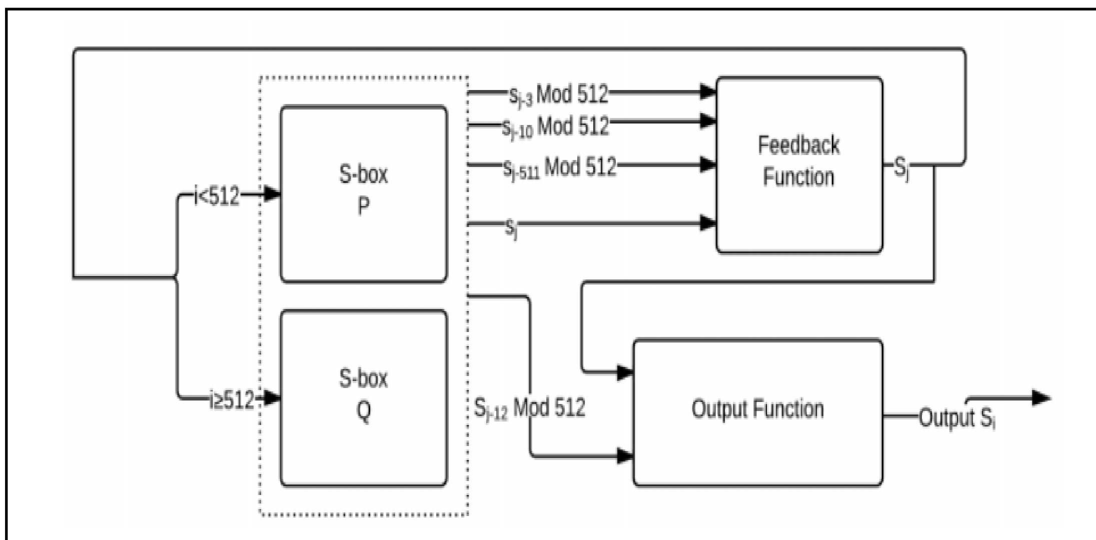

Figure-3.4: Showing Block Diagram for HC-128 Algorithm [15].

3.2.1.1 Cipher Specification

Stream Cipher consists of two secret tables. Each table contains 32 bit elements. After each step, one element from one table gets updated from a nonlinear feedback function. After 1024 steps, all of the elements will have been updated once.

Definitions needed to explain the algorithm:

- K will be our 128 bit key.
- IV will be an 128 bit initialization vector.
- s_i will be the 32 bit output generated by step i.
- P and Q will be the two tables containing 512 32bit elements each. Each element will be referenced as P[i] and Q[i].

Functions used in the algorithm are given as below.

- $f_1(x) = (x >>> 7) \text{ XOR } (x >>> 18) \text{ XOR } (x >> 3)$
- $f_2(x) = (x >>> 17) \text{ XOR } (x >>> 19) \text{ XOR } (x >> 10)$
- $g_1(x, y, z) = ((x >>> 10) \text{ XOR } (z >>> 23)) + (y >>> 8)$
- $g_2(x, y, z) = ((x <<< 10) \text{ XOR } (z <<< 23)) + (y <<< 8)$
- $h_1(x) = Q[x_0] + Q[256 + x_2]$
- $h_2(x) = P[x_0] + P[256 + x_2]$
- $x = x_3 \| x_2 \| x_1 \| x_0$

The initialization is broken into a few steps. These contribute to expanding K and IV into the P and Q tables. For the first step, we expand into an array W of size 1280 for W[i] as follows:

{Step-1: Expanding key and IV into an array W_i ($0 \leq i \leq 1279$)...}

For $i = 0 \rightarrow 7$ do

$W_i \leftarrow K_i$

end for

for $i = 8 \rightarrow 15$ do

$W_i \leftarrow IV_{i-8}$

end for

for $i = 16 \rightarrow 1279$ do

$W_i \leftarrow f_2(W_{i-2}) + W_{i-7} + f_1(W_{i-15}) + W_{i-16} + i$

end for

{Step-2: Update the tables P and Q with the array W}

For $i = 0 \rightarrow 511$ do

$P[i] \leftarrow W_{i+256}$

$Q[i] \leftarrow W_{i+768}$

end for

{Step-3: Run the cipher for 1024 steps and use the outputs to replace the table elements....}

for $i = 0 \rightarrow 511$ do

$P[i] \leftarrow (P[i] + g_1(P[i \boxminus 3], P[i \boxminus 10], P[i \boxminus 511])) \oplus h_1(p[i \boxminus 12])$

$Q[i] \leftarrow (Q[i] + g_2(P[i \boxminus 3], P[i \boxminus 10], P[i \boxminus 511])) \oplus h_2(p[i \boxminus 12])$

end for

3.2.1.2 Key Stream Generation

{Assume N bits are required....}

For $i = 0 \rightarrow N$ do

$j = i \bmod 512$

if $(i \bmod 1024) < 512$ then

$P[j] \leftarrow (P[j] + g_1(P[j \boxminus 3], P[j \boxminus 10], P[j \boxminus 511]))$

$s_i = h_1(P[j \boxminus 12] \oplus P[j])$

else

$Q[j] \leftarrow (Q[j] + g_2(Q[j \boxminus 3], Q[j \boxminus 10], Q[j \boxminus 511]))$

$s_i = h_2(Q[j \boxminus 12] \oplus Q[j])$

end if

$i \leftarrow i + 1$

End for

Starting with the far left as in figure-3.4, we have i's value, based on this, we either go to the P or Q S-box. If i is less than 512 we go to the S-box P, if i is greater than or equal to 512 we go to the S-box Q. Once we know which box we are using we grab 4 elements out of the box by;

S_{j-3} Mod 512;

S_{j-10} Mod 512;

S_{j-511} Mod 512;

S_j Mod 512.

These elements go into the Feedback function below.

3.2.1.3 Feedback Function

In the Feedback Function we use the elements s_{j-3} Mod 512; s_{j-10} Mod 512; s_{j-511} Mod 512, from the table, as parameters in the g function which bit shifts the elements and XORs them together, after the output of the g function is then added to s_j Mod 512 and the result is put back into the correct box at s_j. In the above picture it demonstrates the use of the P S-box, if 'i' would have been greater than or equal to 512 we would have grabs the elements from the Q S-box, replace the function g_1 with g_2 and instead put the output back into the Q S-box. The output of this function also goes into the Output Function described below figure-3.5.

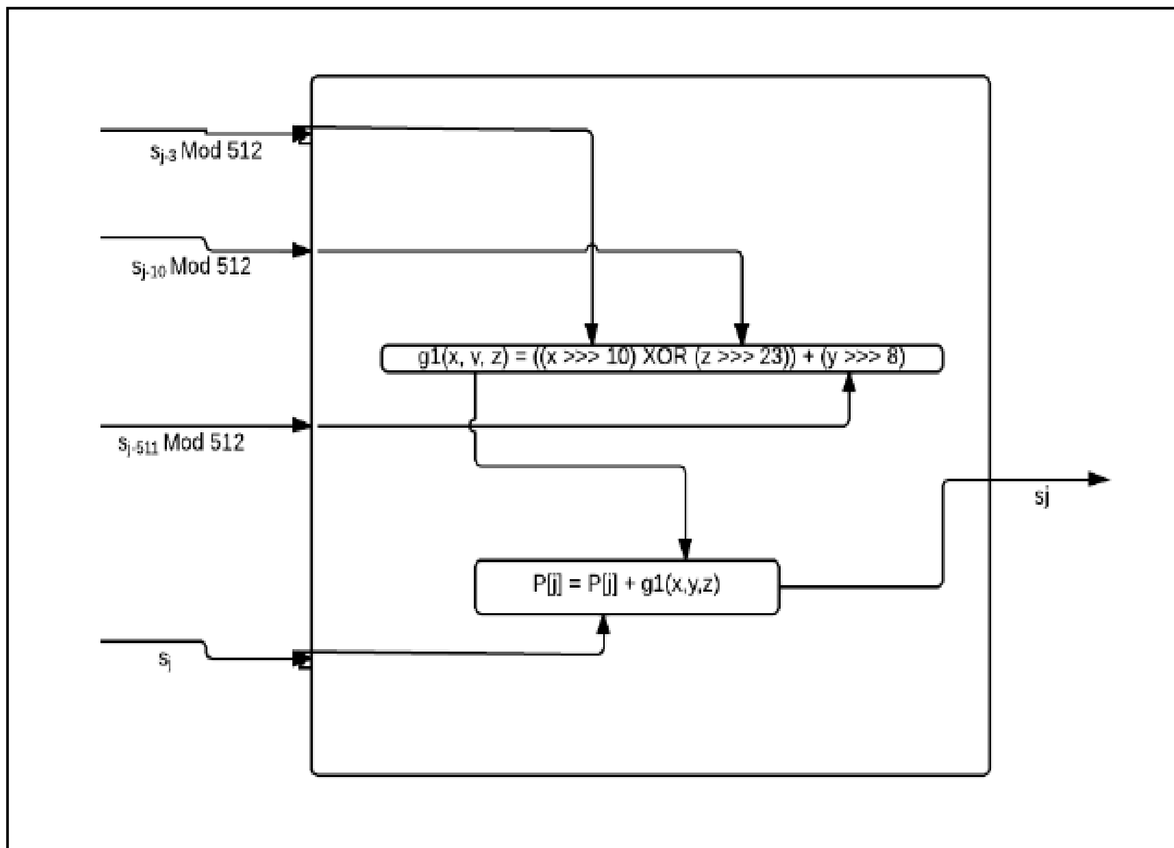

Figure: - 3.5: Showing Feedback Function [15].

3.2.1.4 Output Function

In the Output Function shown above, we use the elements s_j and s_{j-12} Mod 512 in the function $h_1$and then the output of that function goes into the final function $s_i = h_1(x)$ XOR p[j], the output of this function is the key stream.

In the above picture it demonstrates the use of the P S-box, if i would have been greater than or equal to 512, the elements from the Q S-box would have been grabbed and the function h_1 with h_2 would be replaced.

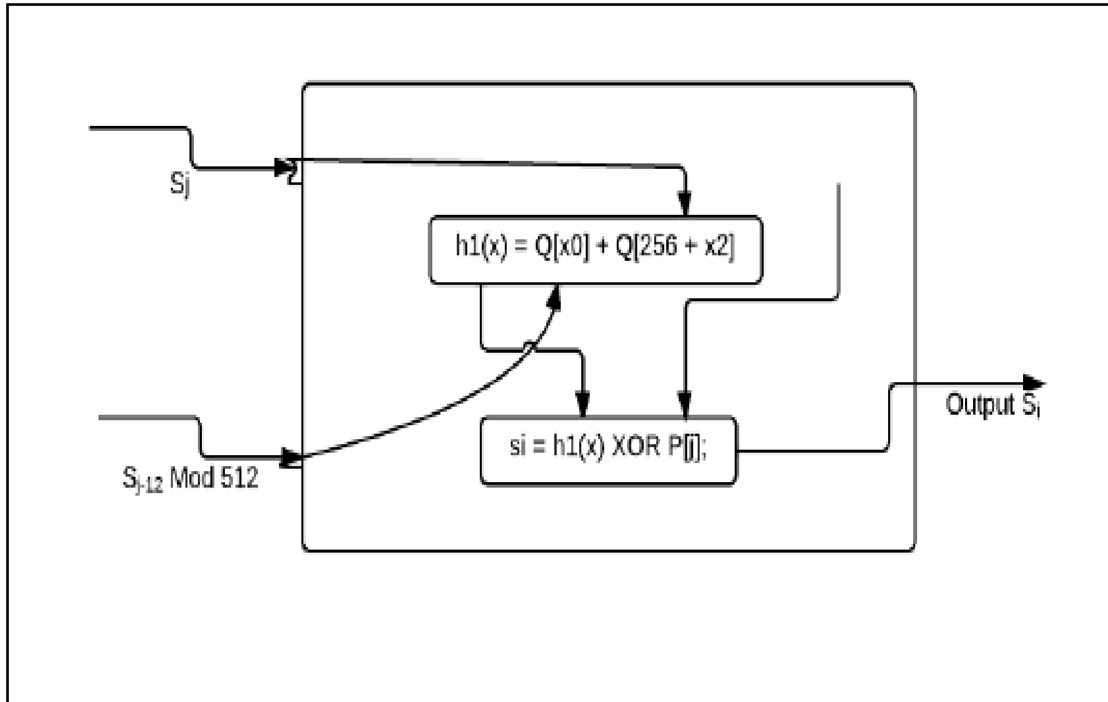

Figure:-3.6: Showing Output Function [15].

3.2.2 Rabbit

Rabbit is a synchronous stream cipher introduced in Fast Software Encryption in 2003. It is one of the most potent candidates of the eSTREAM project. The designers of the cipher targeted to use it in both software and hardware environments. The design is very strong as the designers provided the security analysis considering several possible attacks viz. algebraic, differential, guess-and-determine, and statistical attacks. The Rabbit Algorithm takes 128-bit key and if necessary 64-bit IV as input [11].

In each iteration, it generates 128-bit output. The output is pseudo-random in the natural sense that they cannot be distinguished from random strings of 128-bit with non negligible probability. The core of this cipher consists of 513 internal state bits. Obviously, the output generated in each iteration is some combination of these state-bits. The 513 bits are divided into eight 32-bit state variable, eight 32-bit counter and one counter carry bit. The state functions which update these state variables are non-linear and thus build the basic of security provided by this cipher [11].

The design of rabbit enables faster implementation than common ciphers. Mostly bitwise operations like concatenation, bitwise XOR, shifting are involved which explains its faster performances. A few costly operations like squaring are necessary to enhance the amount of non-linearity. A key of size 128-bit can be used for encrypting up to 2^{64} blocks of plain-text.

3.2.2.1 Specifications of Rabbit

Most of the notation used here are well-known. However, Necessary notations are provided here below.

$\oplus$ Logical Exclusive OR.

& Logical AND.

<<</>> >Left / Right Rotation.

$\ll$ / $\gg$ Left / Right Shift.

◊ Concatenation.

$A^{[g..h]}$ Bit number g through h of A.

In addition, while numbering the bits, the least significant bit is denoted by 0 and hexadecimal numbers are prefixed conventionally by “0x”.

The internal state of the stream cipher consists of 513 bits as stated earlier. 512 bits are divided between eight 32-bit state variables $x_{j,i}$ and eight 32-bit counter variables $c_{j,i}$, where $x_{j,i}$ is the state variable of subsystem j at iteration i, and $c_{j,i}$ denotes the corresponding counter variable. There is one counter carry bit via $\varphi_{7,i}$, which needs to be stored between iterations. Basically it stores the carry output of a summation which updates counter in each iteration. This counter carry bit is initialized to zero. The eight state variables and the eight counters are derived from the key at initialization.

3.2.2.2 Key-Setup Scheme

The Key-Setup scheme consists of three main parts. It takes the key as input and initializes them. Then it interacts with Next-State function several times. Finally to prevent key recovery by inversion of the counter system, it re-initializes the counter system.

The goal of the algorithm used in this step is to expand the input key (128-bit) into both the eight state variables and the eight counters such that there is a one-to-one correspondence between the key and the initial state variables $x_{j,0}$ and the initial counters $c_{j,0}$. The key, $K^{[127..0]}$, is divided into eight sub-keys: $k_0 = K^{[15..0]}$, $k_1 = K^{[31..16]}$, . . ., $k_7 = K^{[127..112]}$.The state and counter variables are initialized from the sub-keys as follows:

$$x_{j,0} = k_{(j+1 \bmod 8)} \lozenge k_j \qquad \text{for } j \text{ even}$$

$$x_{j,0} = k_{(j+5 \bmod 8)} \lozenge k_{(j+4 \bmod 8)} \qquad \text{for j odd}$$

and

$$C_{j,0} = k_{(j+4 \bmod 8)} \lozenge k_{(j+5 \bmod 8)} \qquad \text{for j even}$$

$$C_{j,0} = k_j \lozenge k_{(j+1 \bmod 8)} \qquad \text{for j odd}$$

Then, the system is iterated four times, according to the Next-state function, to diminish correlations between bits in the key and bits in the internal state variables. Finally, the counter variables are re-initialized according to:

$$c_{j,4} = c_{j,4} \oplus x_{((j+4) \bmod 8),4}$$

for all j, to prevent recovery of the key by inversion of the counter system.

A block diagram for Rabbit Stream cipher algorithm is given as below in Figure-3.7.

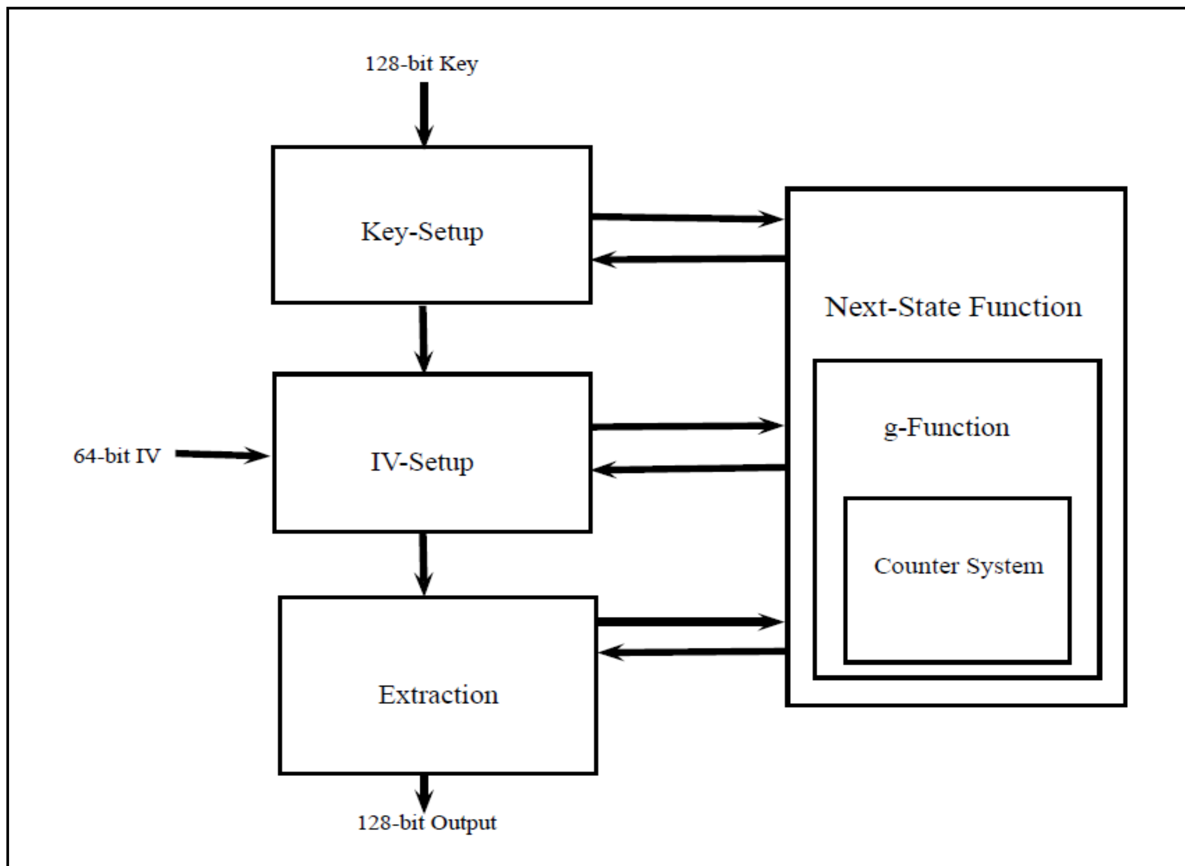

Figure-3.7: Showing an Entire Block Diagram of Rabbit [13].

Key Setup Algorithm is given as below in high level pseudo code:

{Step-1: Initializing System....}

For $j = 0 \rightarrow 7$ do

if (j is even) then

$x_{j,0} \leftarrow \text{CONCAT}(k_{(j+1) \bmod 8}, k_j)$

$c_{j,0} \leftarrow \text{CONCAT}(k_{(j+4) \bmod 8}, k_{(j+5) \bmod 8})$

else

$x_{j,0} \leftarrow \text{CONCAT}(k_{(j+5) \bmod 8}, k_{(j+4) \bmod 8})$

$c_{j,0} \leftarrow$ CONCAT(k_j , $k_{(j+1) \bmod 8}$)

end if

end for

{Step-2: Iterating System by Next-State Function....}

For i= 0 → 3 do

State[$x_{j,i+1}$, $c_{j,i+1}$] ← NEXT-STATE(State[$x_{j,i}$, $c_{j,i}$]) $\forall j \in \{0, . . . , 7\}$

end for

{Step-3: Re-initializing Counters....}

For j = 0 → 7 do

$c_{j,4} \leftarrow$ XOR($c_{j,4}$, $x_{((j+4) \bmod 8),4}$)

end for

3.2.2.3 IV-Setup Scheme

Now, after completion of Key-Setup, one can optionally run IV-setup scheme. The input of this part is the output from Key-Setup and a 64-bit IV. The internal states after key-setup, is called the Master State. In this scheme, a copy of that Master State is modified. The IV-setup scheme works by modifying the counter state as function of the IV. This is done by XORing the 64-bit IV on all the 256 bits of the counter state. The 64 bits of the IV are denoted $IV^{[63..0]}$.

The counters are modified as:

$c_{0,4} = c_{0,4} \oplus IV^{[31..0]}$ $\quad$ $c_{1,4} = c_{1,4} \oplus IV^{[63..48]} \oplus IV^{[31..16]}$

$c_{2,4} = c_{2,4} \oplus IV^{[63..32]}$ $\quad$ $c_{3,4} = c_{3,4} \oplus IV^{[47..32]} \oplus IV^{[15..0]}$

$c_{4,4} = c_{4,4} \oplus IV^{[31..0]}$ $\quad$ $c_{5,4} = c_{5,4} \oplus IV^{[63..48]} \oplus IV^{[31..16]}$

$c_{6,4} = c_{6,4} \oplus IV^{[63..32]}$ $\quad$ $c_{7,4} = c_{7,4} \oplus IV^{[47..32]} \oplus IV^{[15..0]}$

The system is then iterated four times to make all state bits non-linearly dependent on all IV bits. This is essential to incorporate non-linearity in this scheme. Like the previous scheme, this is done by calling the Next-state Function 4 times. The modification of the counter by the IV guarantees that all 2^{64} different IV vectors will lead to unique key-streams. The scheme has been summarized by a high-level pseudo-code as below.

{Step-1: Modifying counters by input IV....}

For j= 0 → 7 do

If j= 0 mod 4 then

$c_{j,4} \leftarrow$ XOR($c_{j,4}$, $IV^{[31..0])}$

```
            end if
            if j = 1 mod 4 then
                  c_{j,4} ← XOR(c_{j,4},CONCAT(IV^[63..48], IV^[31..16]))
            end if
            if  j = 2 mod 4 then
                  c_{j,4} ← XOR(c_{j,4}, IV^[63..32])
            end if
            if  j = 3 mod 4 then
                  c_{j,4} ← XOR(c_{j,4}, CONCAT(IV^[47..32], IV^[15..0]))
            end if
      end for
{Step-2: Iterating System by Next-State Function....}
      For i= 0 → 3 do
            State[x_{j,i+1}, c_{j,i+1}] ← NEXT-STATE(State[x_{j,i}, c_{j,i}]) ∀j ∈ {0, . . . , 7}
      end for
```

3.2.2.4 Extraction Scheme

The Extraction Scheme takes the output from IV-Setup scheme whenever the later is used. Otherwise, it takes the output of Key-setup scheme as its input. In this scheme again the input state variable is iterated using Next-state function. And, after each iteration, the 128-bit output key-stream s_i is extracted from 128-bit internal state variable i.e. x_i as follows:

$s_i^{[15..0]} = x_{0,i}^{[15..0]} \oplus x_{5,i}^{[31..16]}$ $\quad$ $s_i^{[31..16]} = x_{0,i}^{[31..16]} \oplus x_{3,i}^{[15..0]}$

$s_i^{[47..32]} = x_{2,i}^{[15..0]} \oplus x_{7,i}^{[31..16]}$ $\quad$ $s_i^{[63..48]} = x_{2,i}^{[31..16]} \oplus x_{5,i}^{[15..0]}$

$s_i^{[79..64]} = x_{4,i}^{[15..0]} \oplus x_{1,i}^{[31..16]}$ $\quad$ $s_i^{[95..80]} = x_{4,i}^{[31..16]} \oplus x_{7,i}^{[15..0]}$

$s_i^{[111..96]} = x_{6,i}^{[15..0]} \oplus x_{3,i}^{[31..16]}$ $\quad$ $s_i^{[127..112]} = x_{6,i}^{[31..16]} \oplus x_{1,i}^{[15..0]}$

Consequently the high-level pseudo-code has been given below:

```
for i = 0 → SIZE do {Iterate the system...}
      State[x_{j,i+1}, c_{j,i+1}] ← NEXT-STATE(State[x_{j,i}, c_{j,i}]) ∀j ∈ {0, . . . , 7}
      {Generate Key-Stream as in Extraction Scheme ...}
            s_i ←COMPUTE-KEY-STREAM(x_{j,i})∀j ∈ {0, . . . , 7}
```

end for

3.2.2.5 Next-State Function

Now, next step in this cipher is Next-state Function. Actually there are three steps which are performed in this function. First counters are updated according to the counter function, then the g-values are computed from the old state-variable and updated counter-variable. Then the state variables are updated from the newly computed g-values. For better modularity, the implementation can be thought of as the cascading call of three different functions which are doing those different tasks. The Next-state function calls the subroutine g-function which again calls the counter-updating function [13].

The counter-variables are updated by following equations:

$$c_{0,i+1} = c_{0,i} + a_0 + \varphi_{7,i} \bmod 2^{32}$$
$$c_{1,i+1} = c_{1,i} + a_1 + \varphi_{0,i+1} \bmod 2^{32}$$
$$c_{2,i+1} = c_{2,i} + a_2 + \varphi_{1,i+1} \bmod 2^{32}$$
$$c_{3,i+1} = c_{3,i} + a_3 + \varphi_{2,i+1} \bmod 2^{32}$$
$$c_{4,i+1} = c_{4,i} + a_4 + \varphi_{3,i+1} \bmod 2^{32}$$
$$c_{5,i+1} = c_{5,i} + a_5 + \varphi_{4,i+1} \bmod 2^{32}$$
$$c_{6,i+1} = c_{6,i} + a_6 + \varphi_{5,i+1} \bmod 2^{32}$$
$$c_{7,i+1} = c_{7,i} + a_7 + \varphi_{6,i+1} \bmod 2^{32}$$

where the counter carry bit is given by the following equation:

$\varphi_{j,i+1} = 1$ if $c_{0,i} + a_0 + \varphi_{7,i} \geq 2^{32}$ and $j = 0$

$\varphi_{j,i+1} = 1$ if $c_{j,i} + a_j + \varphi_{j-1,i+1} \geq 2^{32}$ and $j > 0$

$\varphi_{j,i+1} = 0$ Otherwise.

The a_j are constants having following values:

$a_0 = \text{0x4D34D34D}$ $a_1 = \text{0xD34D34D3}$

$a_2 = \text{0x34D34D34}$ $a_3 = \text{0x4D34D34D}$

$a_4 = \text{0xD34D34D4}$ $a_5 = \text{0x34D34D34}$

$a_6 = \text{0x4D34D34D}$ $a_7 = \text{0xD34D34D3}$

In the next step, the g-values are computed with the updated counter values and the old state-variables. They are computed as:

$$g_{j,i} = ((x_{j,i} + c_{j,i+1})^2 \oplus ((x_{j,i} + c_{j,i+1})^2 \gg 32)) \bmod 2^{32}$$

Finally, the internal state variables ($x_{'j,i}$s) are computed as follows:

$$x_{0,i+1} = g_{0,i} + (g_{7,i} \lll 16) + (g_{6,i} \lll 16) \bmod 2^{32}$$
$$x_{1,i+1} = g_{1,i} + (g_{0,i} \lll 8) + g_{7,i} \bmod 2^{32}$$
$$x_{2,i+1} = g_{2,i} + (g_{1,i} \lll 16) + (g_{0,i} \lll 16) \bmod 2^{32}$$
$$x_{3,i+1} = g_{3,i} + (g_{2,i} \lll 8) + g_{1,i} \bmod 2^{32}$$
$$x_{4,i+1} = g_{4,i} + (g_{3,i} \lll 16) + (g_{2,i} \lll 16) \bmod 2^{32}$$
$$x_{5,i+1} = g_{5,i} + (g_{4,i} \lll 8) + g_{3,i} \bmod 2^{32}$$
$$x_{6,i+1} = g_{6,i} + (g_{5,i} \lll 16) + (g_{4,i} \lll 16) \bmod 2^{32}$$
$$x_{7,i+1} = g_{7,i} + (g_{6,i} \lll 8) + g_{5,i} \bmod 2^{32}$$

3.2.2.6 Encryption / Decryption Scheme

The extracted bits are XOR'ed with plaintext / cipher text to encrypt/decrypt.

$$c_i = p_i \oplus s_i$$
$$p_i = c_i \oplus s_i$$

where c_i and p_i are the ith 128-bit cipher text and plaintext blocks respectively.

3.2.3 Salsa20/12

Salsa20/12 is a stream cipher submitted to eSTREAM by Daniel Bernstein. It is built on a pseudorandom function based on 32-bit addition, bitwise addition (XOR) and rotation operations, which maps a 256-bit key, a 64-bit nonce, and a 64-bit stream position to a 512-bit output (a version with a 128-bit key also exists). This gives Salsa20 the unusual advantage that the user can efficiently seek to any position in the output stream [12].

It is not patented, and Bernstein has written several public domain implementations optimized for common architectures. The version selected in the eSTREAM profile has 12 rounds. So, it is called Salsa 20/12. Here it is focused on the stream cipher Salsa20 in general.There are a lot of active researches going on around the crypto-community to break this very popular eSTREAM cipher [12]. A complete diagram of Salsa20/12 is given as below in Figure- 3.8.

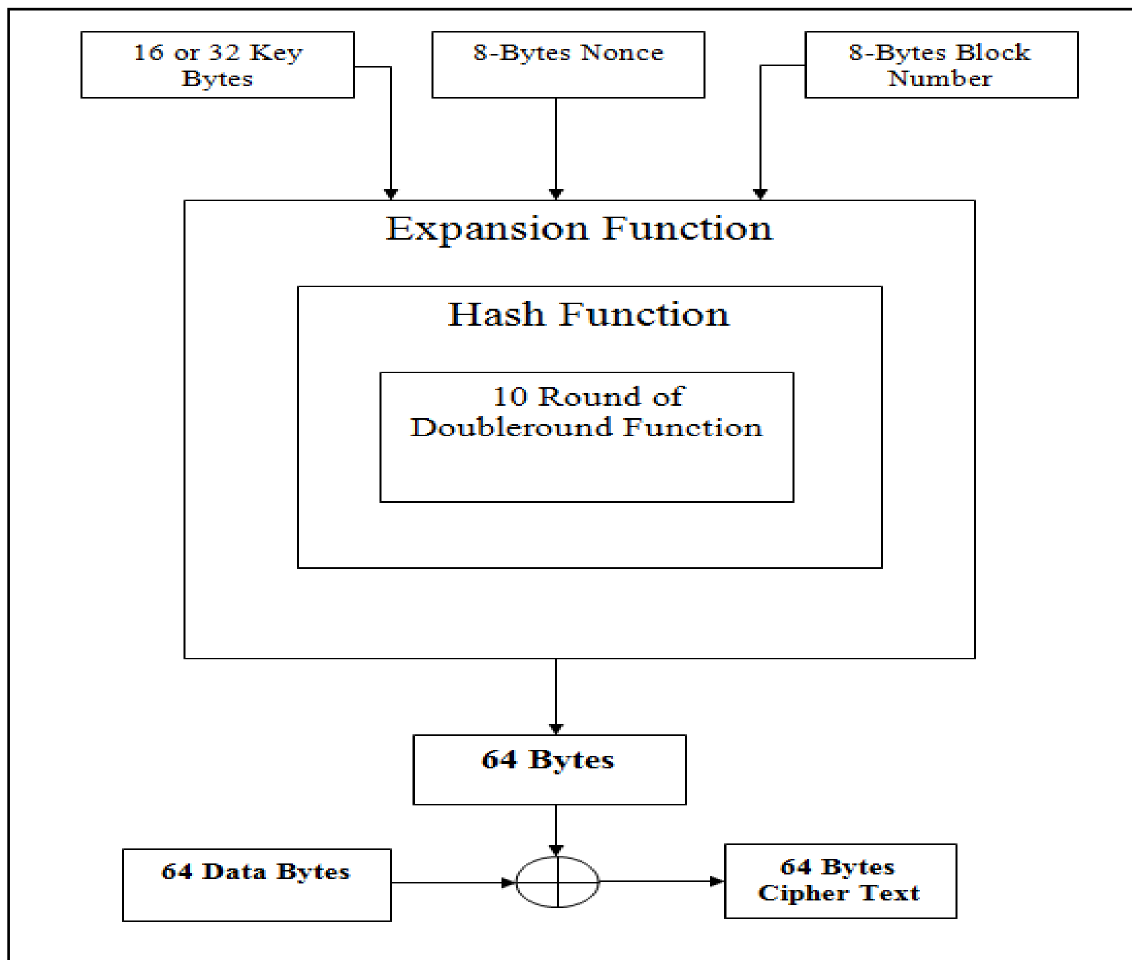

Figure-3.8: Showing a block Diagram of Salsa20/12 Algorithm [26].

3.2.3.1 Specifications of Salsa20/12

The core of Salsa20 is a hash function with 64-byte input and 64-byte output. The hash function is used in counter mode as a stream cipher. Salsa20 encrypts a 64-byte block of plain text by hashing the key, nonce, and block number and XORing the result with the plain text [12]. A word is an element of the set $\{0, 1, . . . , 2^{32} - 1\}$. They are generally represented in hexadecimal notation. The sum of two words u and v is defined as $(u + v) \bmod 2^{32}$. Now, step by step functions as well as algorithm steps are given as below in general form.

Input: - 64 Bytes   Output: - 64 Bytes

Step-1: Quarterround Function.

Input: - $y = \{y_0, y_1, y_2, y_3\}$, then quarterround(y) = z = $\{z_0, z_1, z_2, z_3\}$ is defined as follows:

$z_1 = y_1 \oplus ((y_0 + y_3) \lll 7),$

$z_2 = y_2 \oplus ((z_1 + y_0) \lll 9),$

$z_3 = y_3 \oplus ((z_2 + z_1) \lll 13),$

$z_0 = y_0 \oplus ((z_3 + z_2) \lll 18).$

Step-2: Rowround Function.

Input: - $y = \{y_0, y_1, \ldots, y_{15}\}$, then rowround(y) = $z = \{z_0, z_1, \ldots, z_{15}\}$ is defined as follows:

$(z_0, z_1, z_2, z_3) = \text{quarterround}(y_0, y_1, y_2, y_3)$

$(z_4, z_5, z_6, z_7) = \text{quarterround}(y_5, y_6, y_7, y_4)$

$(z_8, z_9, z_{10}, z_{11}) = \text{quarterround}(y_{10}, y_{11}, y_8, y_9)$

$(z_{12}, z_{13}, z_{14}, z_{15}) = \text{quarterround}(y_{15}, y_{12}, y_{13}, y_{14}).$

Step-3: Columnround Function.

Input: - $y = \{y_0, y_1, \ldots, y_{15}\}$, then columnround(y) = $z = \{z_0, z_1, \ldots, z_{15}\}$ is defined as follows:

$(z_0, z_4, z_8, z_{12}) = \text{quarterround}(y_0, y_4, y_8, y_{12})$

$(z_5, z_9, z_{13}, z_1) = \text{quarterround}(y_5, y_9, y_{13}, y_1)$

$(z_{10}, z_{14}, z_2, z_6) = \text{quarterround}(y_{10}, y_{14}, y_2, y_6)$

$(z_{15}, z_3, z_7, z_{11}) = \text{quarterround}(y_{15}, y_3, y_7, y_{11}).$

Step-4: Doubleround Function.

Input: - $y = \{y_0, y_1, \ldots, y_{15}\}$, then doubleround(y) = $z = \{z_0, z_1, \ldots, z_{15}\}$ and defined as below.

For i=1 to 10

doubleround(y) = rowround(columnround(y))

End for

Step-5: Salsa20 Hash Function.

Input & Output: - 64 bytes

In short, $Salsa20(x) = x + doubleround^{10}(x)$ where, x is a 4-byte word.

In detail, it can be given as below.

(Let x = x[0], x[1], ..., x[63])

x_0 = littleendian(x[0], x[1], x[2], x[3]),

x_1 = littleendian(x[4], x[5], x[6], x[7]),

.

.

x_{15} = littleendian(x[60], x[61], x[62], x[63])

where, littleendian is a function as littleendian(b) = $b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3$ and b is 4-byte sequence of b=(b_0, b_1, b_2, b_3).

Step-6: Salsa 20 Expansion Function

If k is a 32-byte or 16-byte sequence and n is a 16-byte sequence then $Salsa20_k(n)$ is a 64-bytesequence defined as follows:

Let

σ_0 = (101, 120, 112, 97),

σ_1 = (110, 100, 32, 51),

σ_2 = (50, 45, 98, 121),

σ_3 = (116, 101, 32, 107)

and,

τ_0 = (101, 120, 112, 97),

τ_1 = (110, 100, 32, 49),

τ_2 = (54, 45, 98, 121),

τ_3 = (116, 101, 32, 107).

If k_0, k_1, n are 16-byte sequences, then

$Salsa20_{k0,k1}(n) = Salsa20(\sigma_0, k_0, \sigma_1, n, \sigma_2, k_1, \sigma_3)$.

Else if k, n are 16-byte sequences, then

$Salsa20_k(n) = Salsa20(\tau_0, k, \tau_1, n, \tau_2, k, \tau_3)$.

Encryption is performed as byte wise with the plain text. Finally, it is also bit wise XORed operation for encryption as well as decryption.

4.2.4 SOSEMANUK

SOSEMANUK is another new synchronous software-oriented stream cipher selected in eSTREAM project. It uses both basic design principles from the stream cipher SNOW 2.0[5] and transformations derived from the block cipher SERPENT [4]. It is well-known that snow snakes do not exist since snakes either hibernate or move to warmer climes during the winter. Instead Sosemanuk is a popular sport played by the Eastern Canadian tribes. It consists in throwing a wooden stick along a snow bank as far as possible. Its name means snowsnake in the Cree language, since the stick looks like a snake in the snow [9, 13].

The Sosemanuk stream cipher is a new synchronous stream cipher dedicated to software applications. Its key length is variable between 128 and 256 bits. Any key length is claimed to achieve 128-bit security. It is inspired by the design of SNOW 2.0 which is very elegant and achieves a very high throughput. Sosemanuk aims at improving SNOW 2.0 from two respects.

First, it avoids some structural properties which may appear as potential weaknesses, even if the SNOW 2.0 cipher with a 128-bit key resists all known attacks. Second, efficiency is improved on several architectures by reducing the internal state size, thus allowing for a more direct mapping of data on the processor registers [9].

Sosemanuk also requires a reduced amount of static data; this lower data cache pressure yields better performance on several architectures. Another strength of Sosemanuk is that its key setup procedure is based on a reduced version of the well-known block cipher SERPENT, improving classical initialization procedures both from an efficiency and a security point of view.

An entire Block Diagram of SOSEMANUK is given as below in figure-3.9.

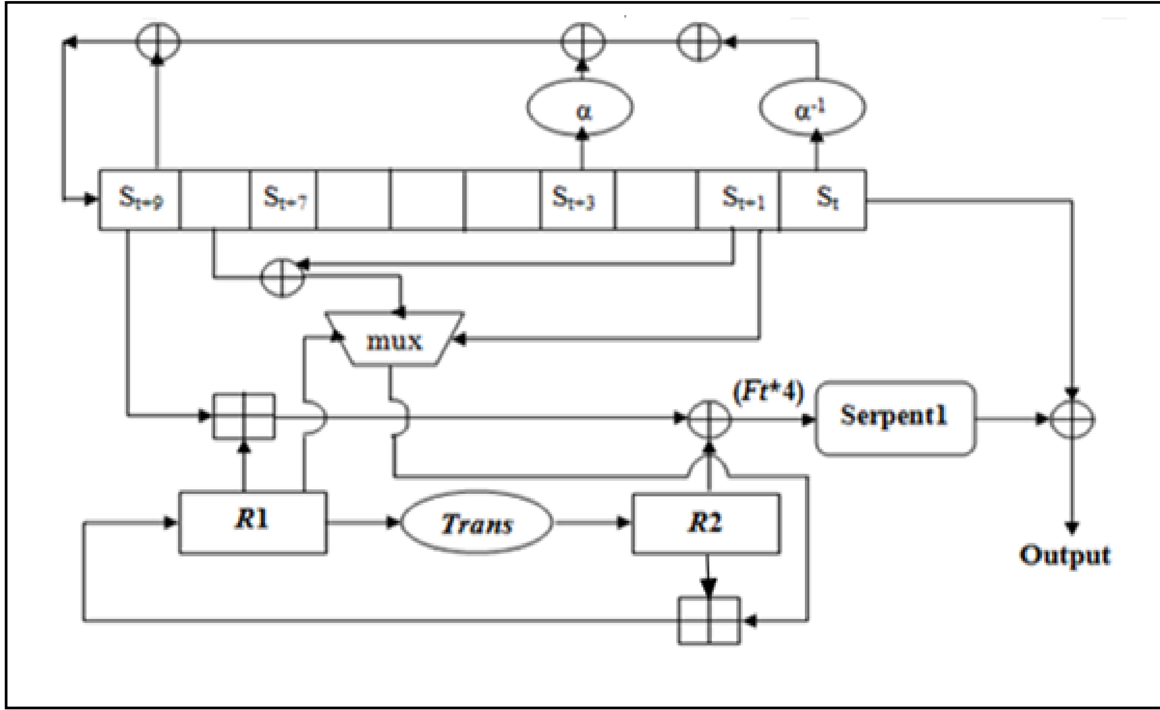

Figure-3.9: Showing Block Diagram of SOSEMANUK [9, 13].

Now, in algorithmic way, step by step functions and steps of SOSEMANUK algorithm is given as below.

Step-1: Key Initialization Process.

It takes SERPENT secret key initialization process where user fed keys are first divided into 8 32-bits pre keys w_{-8} to w_{-1}. After this, 132 intermediate keys are generated as

For i = 0 to 131

$$W_i = (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \Phi \oplus i) <<< 11$$

End For

Where w_i is an intermediate key, $\oplus$ is Exclusive OR operation and Φ is golden fractional ratio of $(\sqrt{5}+1)/2$ or 0x9e3779b9 hexadecimal number. After this, 33 round keys are generated from these intermediate keys by running them in S-box as below.

$\{k_0, k_1, k_2, k_3\} := S_3(w_0, w_1, w_2, w_3)$

$\{k_4, k_5, k_6, k_7\} := S_2(w_4, w_5, w_6, w_7)$

……………………………………

……………………………………

$\{k_{128}, k_{129}, k_{130}, k_{131}\} := S_3(w_{128}, w_{129}, w_{130}, w_{131})$

Step-2: IV Injection (128 bit Value).

SERPENT block cipher consists here of 24 rounds and outputs of 12th, 18th, and 24th rounds are used for IV as below.

Output of 12th round = $(Y_3^{12}, Y_2^{12}, Y_1^{12}, Y_0^{12})$

Output of 18th round = $(Y_3^{18}, Y_2^{18}, Y_1^{18}, Y_0^{18})$

Output of 24th round = $(Y_3^{24}, Y_2^{24}, Y_1^{24}, Y_0^{24})$

Now SOSEMANUK internal states are initialized as below.

$(S_7, S_8, S_9, S_{10}) = (Y_3^{12}, Y_2^{12}, Y_1^{12}, Y_0^{12})$

$(S5, S6) = (Y_1^{18}, Y_3^{18})$

$(S_1, S_2, S_3, S_4) = (Y_3^{24}, Y_2^{24}, Y_1^{24}, Y_0^{24})$

$R1_0 = Y_0^{18}$

$R2_0 = Y_2^{18}$

Step-3: - LFSR definition.

The LFSR operates over elements of F_2^{32}. At time t=0 i.e. initial state, there are 10 32 bit values s1 to s10. After that, new value is computed as below.

$S_{t+10} = S_{t+9} \oplus \alpha^{-1} S_{t+3} \oplus \alpha S_t, \vee t >= 1$

Step-4: - Finite State Machine

At each step, the FSM (Finite State Machine) takes as inputs some words from LFSR state and it updates the memory bits and produces output as below at time t.

$FSM_t : (R1_{t-1}, R2_{t-1}, S_{t+1}, S_{t+8}, S_{t+9}) \longrightarrow (R1_t, R2_t, f_t)$

Where

$R1_t = (R2_{t-1} + mux(lsb(R1_{t-1}), s_{t+1}, s_{t+1} \oplus s_{t+8})) \bmod 2^{32}$

$R2_t = \mathrm{Trans}(R1_{t-1})$

$f_t = (s_{t+9} + R1_t \bmod 2^{32}) \oplus R2_t$

Where lsb(x) is least significant bit of x, mux(c, x, y) is equal to x if c=0 or y if c=1. Similarly, Trans (z) = $(M*z \bmod 2^{32}) <<< 7$, where M is constant value 0x54655307 hexadecimal number.

Step-5: - Output Transformation.

The outputs of the FSM are grouped by four, and *Serpent1* is applied to each group; the result is then combined by XOR with the corresponding dropped values from the LFSR, to produce the output values z_t.

$$(z_{t+3}, z_{t+2}, z_{t+1}, z_t) = Serpent1(f_{t+3}, f_{t+2}, f_{t+1}, f_t) \oplus (s_{t+3}, s_{t+2}, s_{t+1}, s_t)$$

Finally, message will be enciphered with z_t value which is obtained by XORing Serpent1 and words dropped by LFSR as shown in the figure-3.9 above.

Chapter 4

IMPLEMENTATION & TESTING

4.1 Java Implementation

Java was conceived at Sun Microsystems, in 1991. This language is initially called "OAK" but it was renamed as java in 1995 with the Virtual Machine being known as the Java Virtual Machine (JVM). At that time, the use of the World Wide Web was starting to become widespread. The web involved the communication between all sorts of processors and systems; just the sort of situation for which Sun Micro system had developed Java. Hence Java became the preferred language for Web programming [19].

Java compiles the source file (.java) and converts into intermediate file called byte code (.class) which can be run on several architectures with the help of java virtual machine (JVM). This beauty of the java programming language motivates to use of java anywhere or in any type of application development. This makes software developed in java platform independent.

4.2 Choice of the Programming Language: Java

Most of other language likes C, C++ are designed to be compiled for a specific target machine. Although it is possible to compile a C++ program for any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time consuming to create, solution was needed, and to find a solution, java was created which could be used to produce code that can run on a variety of CPUs under different environment.

The Java security APIs spans a wide range of areas. For developing secure application, Cryptographic and public key infrastructure (PKI) interfaces are used. Interfaces for performing authentication and access control enable applications to protect against unauthorized access to protected resources. The APIs allow for multiple interoperable implementations of algorithms and other security services. Services are implemented in providers, which are plugged into the Java platform via a standard interface that makes it easy for applications to obtain security services without having to know anything about their implementations. This allows developers to focus on how to integrate security into their applications, rather than on how to actually implement complex security mechanisms. The Java platform includes a number of providers that implement a core set of security services. It

also allows for additional custom providers to be installed. This enables developers to extend the platform with new security mechanisms [20].

4.3 Netbeans

NetBeans is an integrated development environment (IDE) for developing primarily with Java, but also with other languages, in particular PHP, C/C++, and HTML5. It is developed at Charles University as a student project in 1996. In 1997, it was produced as commercial versions and bought by Sun Microsystems in 1999.

It is also an application platform framework for Java desktop applications and others. The NetBeans IDE is written in Java and can run on Windows, OS X, Linux, Solaris and other platforms supporting a compatible JVM. The NetBeans Platform allows applications to be developed from a set of modular software components called modules.

Different versions of Netbeans IDE are introduced in last few years. NetBeans IDE 7.0 was released in April 2011. On August 1, 2011, the NetBeans Team released NetBeans IDE 7.0.1, which has full support for the official release of the Java SE 7 platform. As passing versions from NetBeans IDE 6.5 to currently developing version NetBeans IDE 8.0 many more features are added in newer versions. NetBeans IDE 7.4 was released in October 15, 2013. NetBeans IDE 8.0 is currently in development. NetBeans IDE 8.0.2 is used for implementing in this thesis [16, 20].

4.4 Research Methodology

Research Study, here is to find out the technique which is of very high speed i.e. the algorithm which can encrypt the given message (plaintext) with very high speed than others. For this, random size of data that is of any kinds, will be collected to feed to individual algorithm and time taken to encrypt will be calculated.

4.4.1 Data Collection

Various sample messages of different sizes will be fed to the different modules. Messages may be either text or number or images/graphics etc. But, for easy, plaintext data of different sizes are taken to input for all algorithms. Secondary data collection method is used here. Because, no primary data is required in this study.

4.5 Implementation Details of Candidate Algorithms

Four algorithms / techniques are made to run by feeding same size of message at once. Different classes are created for each algorithm and related coding, functions are kept in class. Finally, all functions and classes are accessed from the main class. Time taken to

encrypt the message file is calculated individually. The main thing in stream cipher is to design and develop key stream which then will be XORed with the message. After generating key stream, 128 bit size key gets encrypted by XORing with same size of message. Important Java coding and functions for each algorithm are given as below.

4.5.1 HC-128

Main Java coding/functions for HC-128 algorithm are given as below. Initial Vector and initial key is directly given in code level. Message encrypting as well as functions converting into byte code are also given below. Details about coding are mentioned in Appendix section.

- void hc128(String msg) throws java.lang.Exception
- private void init()
- private byte getByte()
- public static byte[] encrypt(HC128 hc, byte[] data)

4.5.2 Rabbit

We can see important functions required in Rabbit Algorithm below. Important functions involving in this algorithm are key setup function, function for enciphering message etc. are given. They are as follows. Details about coding are mentioned in Appendix section.

- public void keySetup(byte[] p_key)
- public void cipher(byte[] p_src, byte[] p_dest, long data_size)
- public String encryptMsg(String msg) throws UnsupportedEncodingException etc.

4.5.3 Salsa20/12

In this Algorithm, Java inbuilt functions and classes are used mostly for implementation. However, some functions like salsa20Encryption and inbuilt classes are given as below. Details about coding are mentioned in Appendix section.

- void salsa20Encrypt(String filename).
- import org.bouncycastle.crypto.StreamCipher;
- import org.bouncycastle.crypto.engines.Salsa20Engine;
- import org.bouncycastle.crypto.params.KeyParameter;
- import org.bouncycastle.crypto.params.ParametersWithIV;

4.5.4 SOSEMANUK

It contains number of important functions as it actually is integrated by two algorithms SNOW2.0 (stream cipher) and SERPENT (block cipher). We can see here below some

important functions like soSemanuk(msg), SetKey() and SetIV() etc. Details about coding are mentioned in Appendix section.

- public void soSeManuk(String msg) throws UnsupportedEncodingException
- public void setKey(byte[] key)
- public void setIV(byte[] iv)

4.6 Sample Test Cases

For testing data input, the different size of text file is taken as input message. Size of 30 bytes file has been taken as smallest size and 10KB as big message. Constant keys as well as IV given in program, sample of input message and generated ciphers are as follows:

4.6.1 Key

I. HC -128

K = "AAAAAAAAqweAAAAT"

II. Rabbit

```
Byte Key[] ={
(byte)0xa0, (byte)0x33, (byte)0xd6, (byte)0x78,
(byte)0x6b, (byte)0x05, (byte)0x14,
(byte)0xac, (byte)0xfc, (byte)0x3d, (byte)0x8e,
(byte)0x2d, (byte)0x6a, (byte)0x2c, (byte)0x27, (byte)0x1d
}
```

III. Salsa20/12

K= "gdsfkhalfjjsfvvh"

IV. SOSEMANUK

```
byte key[]= {
        (byte)0xA7, (byte)0xC0, (byte)0x83, (byte)0xFE, (byte)0xB7
}
```

4.6.2 Input Message (30 Bytes)

“eSTREAM Project, ENCRYPT, EU”

4.6.3 Cipher After Encryption

```
HC128 Cipher :
***************
□  9%□ □□□<4MK□

Salsa20  cipher :
*****************
A&_  |□□J{ e□ 3Lh a□□LX□Q□B□ □□;*□□Ʀ□□0□e □HW□$v□ □  Ʒ□$<□□C\□PA□□ 'ǵ□□R/□□{□ %*□
[□(□'□□Y□□}4□□P□□Pn□□T□-□}x'□g □H)□ □    □□□□6□□(□□>□□ □□; x□a□9 □□□□ □□□ □:□D>□k□□□C□1□□□□
Y|Y□@ f□p□:?□Ḳ =□?Yr#R□3!□□ {□y□□0□□.□□@□r □□T□□□□ □ι□□ □/□ □j□□KS;ʎ□□□ aY□□ □□□bBJ□□□□D,□□

Rabbit Cipher:
**************
□ □□OJ@□%j□□\□2T□ □□OJ@□%j□□\□2T□ □□OJ@□%j□□\□2T□ □□OJ@□%j□□\□2T□ □□OJ@□%j□□\□2T□ □□OJ@□%j□

Sosemanuk Cipher :
******************
5ZEX□S~|□□□□□□□s □ "□□□□1□□6□□□□□<6n #7□□□□□ vO□ □□q□□□Wo_□N□□JF□ □□F□X*□□□□□□□□□>□c□□□□T□
```

4.6.4 Input Message (100 Bytes)

"Performance Analysis of eSTREAM Cipher Finalists: HC-128, Salsa20/12, Rabbit and SOSEMANUK" by BDD.

4.6.5 Cipher After Encryption

```
HC128 Cipher :
***************
□¸□*4□M□%□□ /□ "□□□K978X□.□□ж□ +-^α5□□□□;□> T= @□□aJ□□□ ,□□o□□□w□ □□i□u□]□□. U %>B□;T□p□ы7□

Salsa20  cipher :
*****************
□E□a□□□K9□□   □
'□L *B_K □}S(E□xa e $□□(□

Rabbit Cipher:
**************
□*□□FWK□+b□□ □=□□*□□FWK□+b□□ □=□□*□□FWK□+b□□ □=□□*□□FWK□+b□□ □=□□*□□FWK□+b□□ □=□□*□□FWK□+b□□ □=□

Sosemanuk Cipher :
******************
 □□□□]□R□h□□d□_>b'(□w□}n c□□□□z?□o~□Y□_□1 ,□ □□□1□□ □□□□□□ □□ }□□□□S(□□□□□f3□> □]□□JoP  C□BFw□□}=Zk /!
```

Chapter 5

RESULT & ANALYSIS

This chapter presents an overview of comparison of the eSTREAM ciphers in terms of performance and cost. Target Architecture and Specifications are described in this chapter. Time in system nanosecond needed for encrypting all eSTREAM cipher algorithms implemented in java is calculated and performance is analyzed as cycle/byte.

5.1 Target Architectures

The main goal of this thesis is to measure the performance of the all eSTREAM cipher finalists. These candidate algorithms are tested on desktop system. The following system is used:

- A PC with an Intel Core i3 Processor 2.53GHz having 2GB RAM .the operating system is Windows7 Ultimate running in 32-bit mode. The system is running the java VM 22.0 –b10, Java HotSpot(TM) with NetBeans IDE 8.

5.2 Measuring Cost

There is some extra cost which may be added to the absolute cost for encrypting the different size of messages but this is equally affected to all candidates algorithm on the execution. The system time in nanosecond is taken just before the execution of code segment for generating key stream and encrypting the message in each algorithm and the completion of the execution. The time spending for encrypting message is calculated by subtracting start time taken before execution from completion time taken after completing execution of specific code segment [18, 20]. The time required for each algorithm is calculated as follows:

```
long    startTime = System.nanoTime();
//  createKEY and encrypt function call
long    timeRequired  =  System.nanoTime() -  startTime;
```

Various processes may be run in background of system so absolute measurement may not be carried out. Due to this reason, time needed for encrypting given message in all algorithm may not be observe same in every run of program. Therefore at least 5 times, the program implemented in java is run in architecture described as above section and finally average required time observed in every run is calculated as:

Average required Time = $\sum_{i=1}^{5} \frac{T_i}{5}$ where T_i represent time obtained in i^{th} run of execution.

This average calculated time is used to calculate cycle per byte.

5.3 Measuring Performance

Timing cryptographic primitive is useful when analyzing the performance of multiple algorithms on a single machine. But, it may vary on other machine therefore, cryptographers prefer to measure how many cycle it takes to process each byte. Different cycle/byte is calculated in the same box also because of background other process. So to optimize such extra cost, average is taken running multiple times in same machine for each candidate algorithms.

In this thesis, Cycle/byte calculation with the following parameters: time in second spent generating key and encrypting the message (T_s), frequency of the CPU in Hz(F) and message input in bytes(L). The formula for creating cycle/byte suggested by [19] is:

$$\text{Cycle/byte} = \frac{T_s * F}{L}$$

5.4 Analysis

This section will present the result of the performance tests for various input sizes of each algorithm. A simple multiple histograms for each candidate algorithms will be presented each for cycle per byte calculation.

Following table and corresponding charts show the overall performance in the different encryption algorithm, HC-128, Rabbit, Salsa20/12 and SOSEMANUK. Different sizes of data like 30 bytes, 100 bytes, 1KB, 5KB, 10KB, 30KB and 60KB are taken by every candidate algorithms as below. Small size of message is considered to be 30 bytes or less than that and big size of data means 60KB or bigger than that size of message.

Message Size = 30 Bytes

Table 5.1: Performance of Candidate Algorithms for Small Message Size (30 Bytes)

Candidate Algorithms	1st Run	2nd Run	3rd Run	4th Run	5th Run	Average
HC-128	105	250	253	603	367	315.6
Rabbit	100	187	213	500	301	260.2
Salsa20/12	593	859	570	783	645	690
SOSEMANUK	5670	2311	1254	1123	2109	2493.4

Table 5.2: Performance of all the Candidate Algorithms for Small Message Size (30 bytes) Calculated in Cycle/Byte.

Candidate Algorithms	Cycle/Byte
HC-128	26.62
Rabbit	21.94
Salsa20/12	58.19
SOSEMANUK	210.28

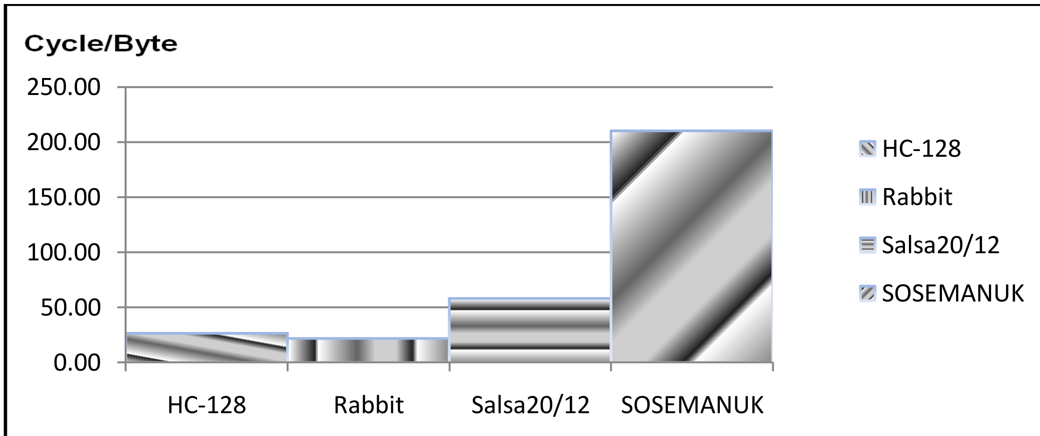

Figure 5.1: Performance of Candidate Algorithms for small Message Size (30 Bytes) shown in Bar Diagram.

Message Size = 100 Bytes

Table 5.3: Performance of all the Candidate Algorithms for Message Size (100 Bytes)

Candidate Algorithms	1st Run	2nd Run	3rd Run	4th Run	5th Run	Average
HC-128	1134	1578	1504	1375	1301	1378.4
Rabbit	2001	1040	1180	1956	1297	1494.8
Salsa20/12	10523	3843	8891	5361	3769	6477.4
SOSEMANUK	13115	12280	10106	6247	3367	9023

Table 5.4: Performance of Candidate Algorithms for Message Size (100 bytes) in Cycle/Byte.

Candidate Algorithms	Cycle/Byte
HC-128	116.25
Rabbit	126.06
Salsa20/12	546.26
SOSEMANUK	760.94

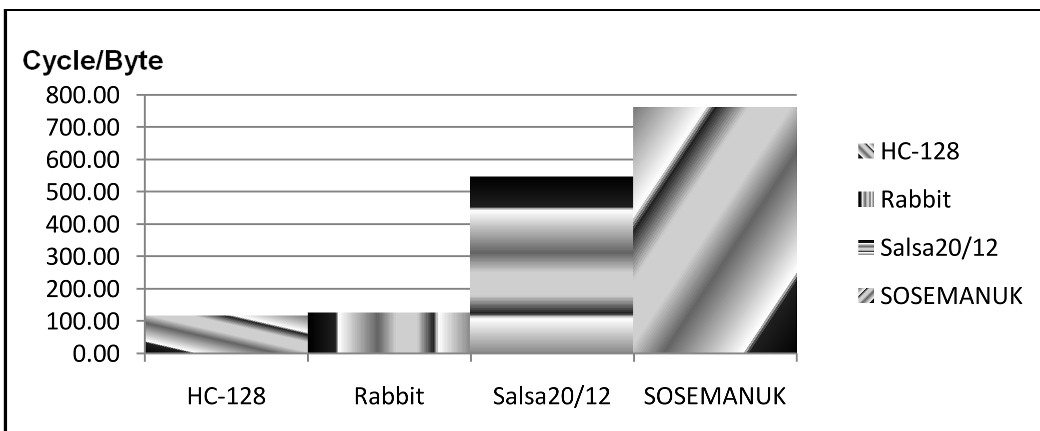

Figure 5.2: Performance of Candidate Algorithms for Message Size (100 bytes) shown in Bar Diagram

Message Size = 1KB

Table 5.5: Performance of all the Candidate Algorithms for Message Size (1KB)

Candidate Algorithms	1st Run	2nd Run	3rd Run	4th Run	5th Run	Average
HC-128	3366	3298	1009	1284	2347	2260.8
Rabbit	7899	4500	3392	1558	1807	3831.2
Salsa20/12	24865	24897	16621	4534	5417	15266.8
SOSEMANUK	56745	56814	32435	25033	23456	38896.6

Table 5.6: Performance of Candidate Algorithms for Message Size (1KB) in Cycle/Byte.

Candidate Algorithms	Cycle/Byte
HC-128	190.66
Rabbit	323.10
Salsa20/12	1287.50
SOSEMANUK	3280.28

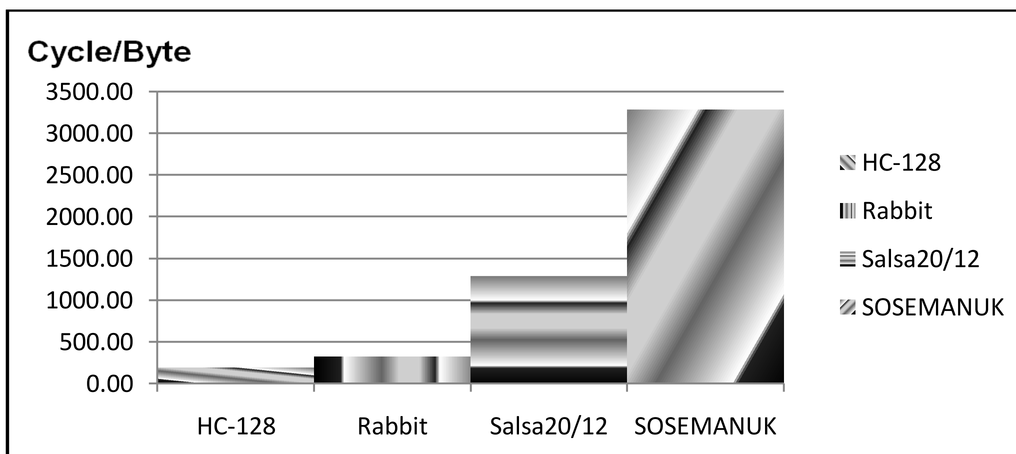

Figure 5.3: Performance of Candidate Algorithms for Message Size (1KB) shown in Bar Diagram.

Message Size = 5KB

Table 5.7: Performance of all the Candidate Algorithms for Message Size (5KB)

Candidate Algorithms	1st Run	2nd Run	3rd Run	4th Run	5th Run	Average
HC-128	23053	21691	15244	11009	10688	16337
Rabbit	62985	137701	69665	93203	39571	80625
Salsa20/12	51624	33131	76044	45206	47655	50732
SOSEMANUK	432531	194098	242806	124238	191017	236938

Table 5.8: Performance of Candidate Algorithms for Message Size (5KB) in Cycle/Byte.

Candidate Algorithms	Cycle/Byte
HC-128	1377.75
Rabbit	6799.38
Salsa20/12	4278.40
SOSEMANUK	19981.77

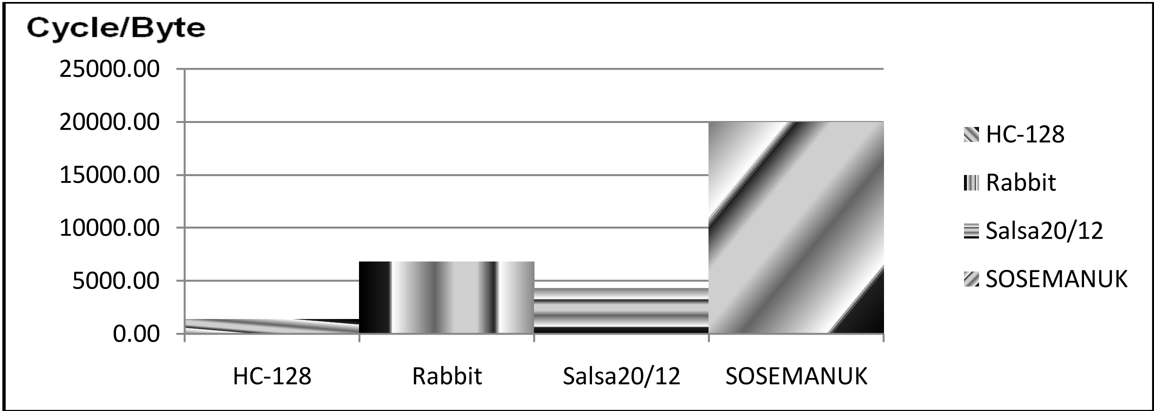

Figure 5.4: Performance of Candidate Algorithms for Message Size (5KB) shown in Bar Diagram.

Message Size = 10KB

Table 5.9: Performance of all the Candidate Algorithms for Message Size (10KB)

Candidate Algorithms	1st Run	2nd Run	3rd Run	4th Run	5th Run	Average
HC-128	27226	9297	9048	17314	10082	14593.4
Rabbit	59248	48334	74097	86689	96834	73040.4
Salsa20/12	49138	30099	11927	29437	33605	30841.2
SOSEMANUK	178104	98988	320633	136681	211990	189279.2

Table 5.10: Performance of Candidate Algorithms for Message Size (10KB) in Cycle/Byte.

Candidate Algorithms	Cycle/Byte
HC-128	1230.71
Rabbit	6159.74
Salsa20/12	2600.94
SOSEMANUK	15962.55

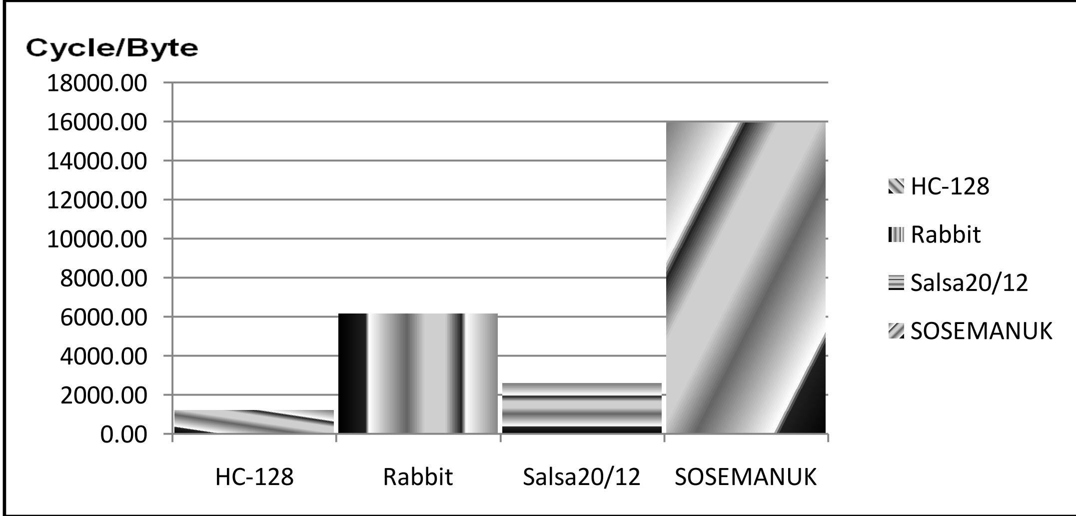

Figure 5.5: Performance of Candidate Algorithms for Message Size (10KB) shown in Bar Diagram

Message Size = 30KB

Table 5.11: Performance of all the Candidate Algorithms for Message Size (30KB)

Candidate Algorithms	1st Run	2nd Run	3rd Run	4th Run	5th Run	Average
HC-128	337981	125141	45742	657432	563458	345950.8
Rabbit	95501	99466	240642	765431	4365877	1113383
Salsa20/12	50985	62712	46018	86542	76599	64571.2
SOSEMANUK	7918750	8459876	8996713	5764532	4673423	7162659

Table 5.12: Performance of Candidate Algorithms for Message Size (30KB) in Cycle/Byte.

Candidate Algorithms	Cycle/Byte
HC-128	29175.18
Rabbit	93895.33
Salsa20/12	5445.50
SOSEMANUK	604050.89

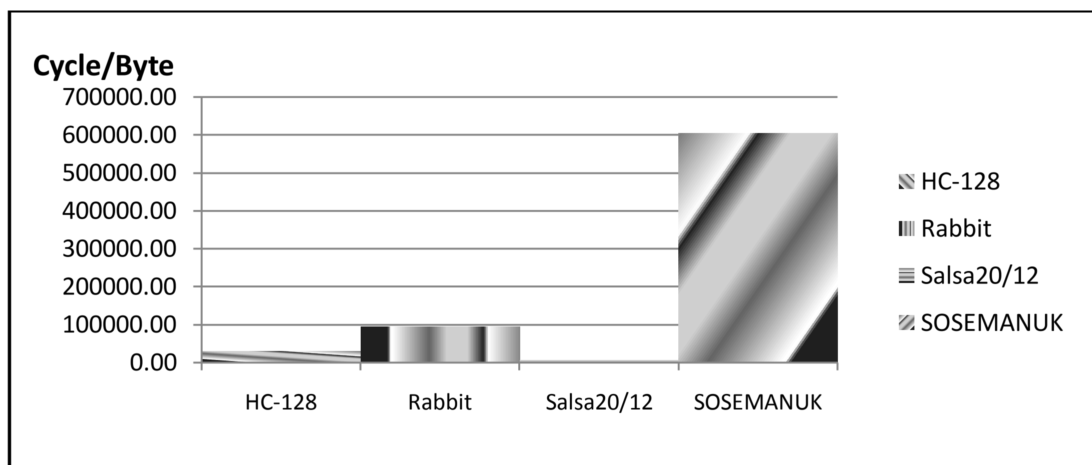

Figure 5.6: Performance of Candidate Algorithms for Message Size (30KB) shown in Bar Diagram

Message Size = 60KB

Table 5.13: Performance of all the Candidate Algorithms for Message Size (60KB)

Candidate Algorithms	1st Run	2nd Run	3rd Run	4th Run	5th Run	Average
HC-128	322307	230003	465700	313542	140082	294326.8
Rabbit	337806	346900	386750	212321	936834	444122.2
Salsa20/12	19226	120560	50385	29437	83605	60642.6
SOSEMANUK	19387329	2477085	9963301	136681	2211990	6835277

Table 5.14: Performance of Candidate Algorithms for Message Size (60KB) in Cycle/Byte.

Candidate Algorithms	Cycle/Byte
HC-128	24821.56
Rabbit	37454.31
Salsa20/12	5114.19
SOSEMANUK	576441.71

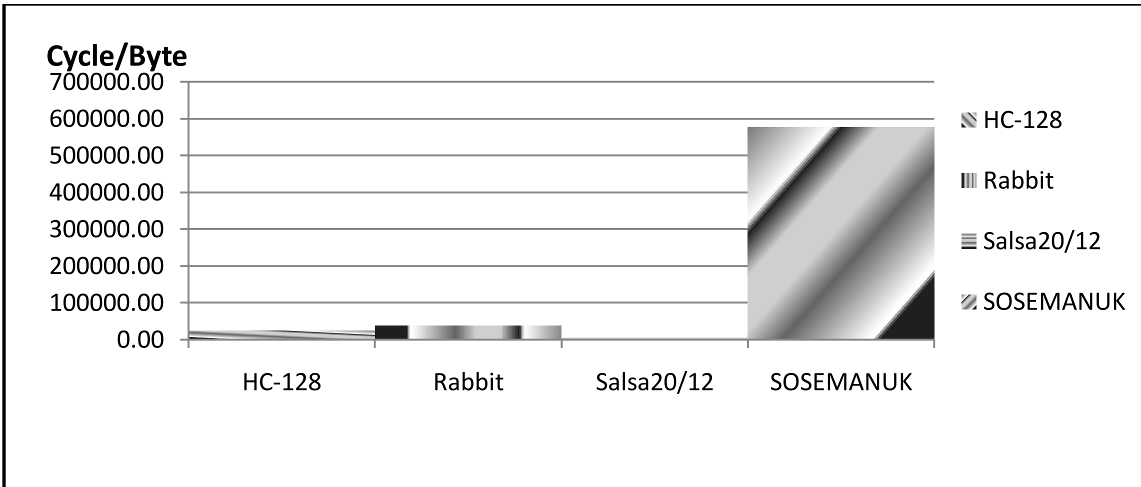

Figure 5.7: Performance of Candidate Algorithms for Message Size (60KB) shown in Bar Diagram

5.5 Result

By the help of above mentioned measuring criteria in the targeted architecture as given in section-5.1 to 5.3, thus obtained result is analyzed in section-5.4. Cycle/byte measuring unit is the best way to check the performance of any algorithms. After running the each algorithm with different sized message, it is observed that performance of Rabbit is better than other algorithms if the message size is very small.

With the increment of size of message, performance of Salsa20/12 is found to be better. Rabbit yields 17.58% to 89.28% better performance in small size of message. HC-128 yields 46.37% to 92.39% better performance in some other cases (bigger message size). But finally, the performance of Salsa20/12 is found to be the best algorithm amongst others for big size of message. It yields 38.32% to 93.67% better performance in all the huge message size of data. Therefore, Salsa20/12 eSTREAM cipher algorithm among all finalists is the best algorithm purposed in this study.

Chapter 6

CONCLUSION & FUTURE WORK

6.1 Conclusions

In this thesis, those eSTREAM cipher finalists (algorithms) are studied, discussed and implemented using most popular and highly accurate programming language Java. After implementation, different size messages were feed to encrypt to all the candidate algorithms and results were observed and analyzed. The result of empirical performance comparison shows that Salsa20/12 is the best one among other algorithms. But, Rabbit algorithm also seems to be better in small size of message i.e. in less than 30 bytes message. While message size gets increased, performance of HC-128 seems to be better in some cases. But, Salsa20/12 seems to be the best for big data size. Hence, Salsa20/12 algorithm shows the best performance among all other eSTREAM cipher finalists.

6.2 Future Works

Actually, considering and keeping in mind that security as well as all the other parameters are constant, the thesis work is carried out here. In this thesis, performance analysis among eSTREAM cipher finalists is done. Selecting Algorithm which can encrypt with high speed only is not matter but security is also. Security has been a great thread and challenge for entire field of cryptography. Similarly, there are some other parameters and cases as well which should be also counted while analyzing these algorithms. So, in future work, it can be the study to optimize and find better algorithm by improving security issue.

REFERENCES

[1] Carlos Cid (RHUL) and Matt Robshaw (FTRD), The eSTREAM Portfolio in 2012. Version 1.0, 16 January 2012.

[2] M. Robshaw and O. Billet, editors. New Stream Cipher Designs: The eSTREAM Final-ists. LNCS 4986, pp. 267293. Springer 2008.

[3] ECRYPT Network of Excellence. The eSTREAM project, available via http://www.ecrypt.eu.org/stream/.

[4] E. Biham, L.R. Knudsen, and R.J. Anderson. Serpent: A New Block Cipher Proposal. In S. Vaudenay, editors, Proceedings of FSE 1998, LNCS, volume 1372, pp. 222-238, Springer Verlag.

[5] PatrikEkdahl and Thomas Johansson, A New Version of the Stream Cipher SNOW.

[6] D.J. Bernstein. Salsa20 page. http://cr.yp.to/snuffle.html.

[7] C. Berbain1, O. Billet1, A. Canteaut2, N. Courtois3, H. Gilbert1,L. Goubin4, A. Gouget5, L.Granboulan6, C. Lauradoux2, M. Minier2,T. Pornin7 and H. Sibert5 "Sosemanuk, a fast software-oriented stream cipher".

[8] Hongjun Wu, "The Stream Cipher HC-128", Katholieke Universiteit Leuven, ESAT/SCD-COSIC Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgiumwu.hongjun@esat.kuleuven.be

[9] Martin Boesgaard, Mette Vesterager, Thomas Christensenn and Erik Zenner," The Stream Cipher Rabbit", CRYPTICOA/S Fruebjergvej 3, 2100 Copenhagen, Denmark, info@cryptico.com

[10] Daniel J. Bernstein, Department of Mathematics, Statistics, and Computer

Science(M/C 249), "Salsa20 speci_cation", The University of Illinois at Chicago, Chicago, IL 60607-7045, snuffle@box.cr.yp.to

[11] Pratyay Mukherjee, "An Overview of eSTREAM Ciphers", Centre of Excellence in Cryptology, Indian Statistical Institute.

[12] C. Paar, J. Pelzl, *Understanding Cryptography*, DOI 10.1007/978-3-642-04101-3 2,_c Springer-Verlag Berlin Heidelberg 2010.

[13] "HC-128 Stream Cipher", A-Team Deliverables (Michael Burns, Brian Baum).

[14] Netbeans ide 7.1 features , May 2010. http://netbeans.org/features/index.html

[15] Petrank, E., Rackoff, C., *CBC MAC for real-time data sources*. J.Cryptology, vol. 13, no. 3, pp.315–338, Springer-Verlag, 2000.

[16] Rogaway, P., Black, J., *A block- cipher mode operation for parallelizable message Authentication.* Advances in cryptology – EUROCRYPTO 2002 ,LNCS 2332, pp . 384-397, Springer – Verlag ,2002.

[17] http://java.sun.com/javame Sun Microsystems Inc.,

[18] *Timing Cryptographic Primitives*, http:// etutorials.org

[19] William Stallings *Cryptography And Network Security Principles and Practice*, Prentice all, Fifth Edition, 2010.

[20] "Performance Analysis of Cipher Block Chaining Message Authentication Code (CBC MAC) and its Variants" dissertation by CBC, page no 39 to 41.

[21] H.C.A.V. Tilborg, *Fundamentals of Cryptology*, Kluwer Academic Publisher Boston , 1988.

APPENDIX

Code of Implementation

HC-128

Important functions and java coding are given as below for HC-128 algorithm.

```
void hc128(String msg) throws java.lang.Exception
	{
		String iv_srt = "@#$$54214AEFDCAE";
		 String key_srt = "AAAAAAAAqweAAAAT";
		 HC128 hc_enc = new HC128(iv_srt.getBytes(), key_srt.getBytes());
		String s =msg ;

		byte[] ed = encrypt(hc_enc, s.getBytes());
		byte[] ed33 = encrypt(hc_enc, ed);
		System.out.println();
		System.out.println("HC128 Cipher : ");
		System.out.println("***************");
		System.out.println(new String(ed33));
	}

 //To encrypt the message
public static byte[] encrypt(HC128 hc, byte[] data)
	{
		for (int i = 0; i < data.length; i++) {
		data[i] = hc.returnByte(data[i]);
	}
	return data;
	}
```

```
private void init()
{
        if (key.length != 16) {
          throw new java.lang.IllegalArgumentException("The key must be 128 bit long");
        }
        cnt = 0;
        int[] w = new int[1280];
        for (int i = 0; i < 16; i++) {
            w[i >> 3] |= key[i] << (i & 0x7);
        }
        System.arraycopy(w, 0, w, 4, 4);
        for (int i = 0; i < Math.min(16, iv.length); i++) {
            w[(i >> 3) + 8] |= iv[i] << (i & 0x7);
        }
        System.arraycopy(w, 8, w, 12, 4);
         for (int i = 16; i < 1280; i++) {
            w[i] = f2(w[i - 2]) + w[i - 7] + f1(w[i - 15]) + w[i - 16] + i;
        }
         System.arraycopy(w, 256, p, 0, 512);
        System.arraycopy(w, 768, q, 0, 512);
        for (int i = 0; i < 512; i++) {
            p[i] = step();
        }
        for (int i = 0; i < 512; i++) {
            q[i] = step();
        }
         cnt = 0;
    }

     private byte getByte() {
        if (idx == 0)
        {
            int step = step();
            buf[3] = (byte) (step & 0xFF);
            step >>= 8;
            buf[2] = (byte) (step & 0xFF);
            step >>= 8;
            buf[1] = (byte) (step & 0xFF);
            step >>= 8;
            buf[0] = (byte) (step & 0xFF);
        }
        byte ret = buf[idx];
        idx = idx + 1 & 0x3;
        return ret;
    }
```

Rabbit

We can see important functions required in Rabbit Algorithm below. Important functions involving in this algorithm are key setup function, function for enciphering message etc. are given.

```
public void keySetup(byte[] p_key)
   {
      int k0, k1, k2, k3, i;
      k0 = os2ip(p_key, 12);
      k1 = os2ip(p_key, 8);
      k2 = os2ip(p_key, 4);
      k3 = os2ip(p_key, 0);
      x[0] = k0;
      x[2] = k1;
      x[4] = k2;
      x[6] = k3;

      x[1] = ( k3 << 16 ) | ( k2 >>> 16 );
      x[3] = ( k0 << 16 ) | ( k3 >>> 16 );
      x[5] = ( k1 << 16 ) | ( k0 >>> 16 );
      x[7] = ( k2 << 16 ) | ( k1 >>> 16 );

      c[0] = rotL(k2, 16);
      c[2] = rotL(k3, 16);
      c[4] = rotL(k0, 16);
      c[6] = rotL(k1, 16);

      c[1] = (k0 & 0xffff0000) | (k1 & 0x0000ffff);
      c[3] = (k1 & 0xffff0000) | (k2 & 0x0000ffff);
      c[5] = (k2 & 0xffff0000) | (k3 & 0x0000ffff);
      c[7] = (k3 & 0xffff0000) | (k0 & 0x0000ffff);
      carry = 0;
      for( i = 0; i < 4; i++)
      {
         next_state();
      }
      for( i = 0; i < 8; i++)
      {
         c[ (i + 4) & 7 ] ^= x[i];
      }
   }
```

```
public void cipher(byte[] p_src, byte[] p_dest, long data_size)
  {
    int i, j, m;
    int[] k = new int[4];
    byte[] t = new byte[4];
    for( i = 0; i < data_size; i+=16)
    {
      next_state();
      k[0] = os2ip(p_src, i * 16 + 0) ^ x[0] ^ ( x[5] >>> 16 ) ^ ( x[3] << 16 );
      k[1] = os2ip(p_src, i * 16 + 4) ^ x[2] ^ ( x[7] >>> 16 ) ^ ( x[5] << 16 );
      k[2] = os2ip(p_src, i * 16 + 8) ^ x[4] ^ ( x[1] >>> 16 ) ^ ( x[7] << 16 );
      k[3] = os2ip(p_src, i * 16 + 12) ^ x[6] ^ ( x[3] >>> 16 ) ^ ( x[1] << 16 );
      for( j = 0; j < 4; j++)
      {
        t = i2osp(k[j]);
        for( m = 0; m < 4; m++)
        {
          p_dest[ i * 16 + j * 4 + m ] = t[m];
        }
      }
    }
  }
```

```
public String encryptMsg(String msg) throws UnsupportedEncodingException
    {
         byte[] key = {
          (byte)0xa0, (byte)0x33, (byte)0xd6, (byte)0x78,
          (byte)0x6b, (byte)0x05, (byte)0x14, (byte)0xac,
          (byte)0xfc, (byte)0x3d, (byte)0x8e, (byte)0x2d,
          (byte)0x6a, (byte)0x2c, (byte)0x27, (byte)0x1d
        };
        byte[] message = msg.getBytes();
        byte[] ciphertext = new byte[16];
        Rabbit rtest = new Rabbit();
        rtest.keySetup(key);
        rtest.cipher(message, ciphertext, 16);
        Rabbit rtest2 = new Rabbit();
        rtest2.keySetup(key);
        byte[] szT = new byte[16];
        for( int i = 0; i < 16; i++)
        {
            szT[i] = 0;
        }
        rtest2.cipher(ciphertext, szT, 16);
          String s = new String(ciphertext, "US-ASCII");
         System.out.print(s);
        String ms = ciphertext.toString();
        return ms;
    }
}
```

Salsa20/12

In this Algorithm, Java inbuilt functions and classes are used mostly for implementation. However, some functions like salsa20Encryption are given as below.

```
void salsa20Encrypt(String filename) {
        String key = "gdsfkhalfjjsfvvh";
        if (key.length() != 16 && key.length() != 32) {
            System.out.println("\n  *** key must be 16 or 32 bytes, so it will *** ");
            System.out.println(  "  *** be padded or truncated as appropriate  *** \n"); }
        key = padnulls(key, 16);
        byte [] nonce = str2byt("abcdefyt");
        KeyParameter     keyparam = new KeyParameter(str2byt(key));
        ParametersWithIV params   = new ParametersWithIV(keyparam, nonce);
        byte[] content = loadfmfile(filename+".txt");
        StreamCipher salsa = new Salsa20Engine();
        salsa.init(true, params);
        byte[]  ciphertext = new byte[content.length];
        salsa.processBytes(content, 0, content.length, ciphertext, 0);
        String newfilename = "cipher " + filename +".txt";
        savetofile(newfilename, concat(nonce, ciphertext));
        byte[] content2 = loadfmfile(newfilename);
        nonce = extract(content2, 0, 8);
        ciphertext = extract(content2, 8, content2.length);
        KeyParameter     keyparam2 = new KeyParameter(str2byt(key));
        ParametersWithIV params2   = new ParametersWithIV(keyparam2, nonce);
        StreamCipher salsa2 = new Salsa20Engine();
        salsa2.init(true, params2);
        byte[]  plaintext = new byte[ciphertext.length];
        salsa2.processBytes(ciphertext, 0, ciphertext.length, plaintext, 0);
        System.out.println();
        System.out.println("Salsa20  cipher :");
        System.out.println("*****************");
        System.out.println(byt2str(ciphertext));}
}
```

SOSEMANUK

It contains number of important functions as it actually is integrated by two algorithms SNOW2.0 (stream cipher) and SERPENT (block cipher). We can see here below some important functions like soSemanuk(msg), SetKey() and SetIV() etc.

```
public void soSeManuk(String msg) throws UnsupportedEncodingException
        {
                byte[] key = {
                        (byte)0xA7, (byte)0xC0, (byte)0x83, (byte)0xFE,
                        (byte)0xB7
                };

                byte[] iv = {

                        (byte)0x00, (byte)0x11, (byte)0x22, (byte)0x33,
                        (byte)0x44, (byte)0x55, (byte)0x66, (byte)0x77,
                        (byte)0x88, (byte)0x99, (byte)0xAA, (byte)0xBB,
                        (byte)0xCC, (byte)0xDD, (byte)0xEE, (byte)0xFF
                };

                SosemanukFast sf = new SosemanukFast();
                sf.setKey(key);
                sf.setIV(iv);
                byte[] tmp = new byte[160];
                sf.makeStream(tmp, 0, tmp.length);
           String keybin ="";
                        for (int i = 0; i < 10; i ++) {
                        for (int j = 0; j < 16; j ++) {
                                int v = tmp[i * 16 + j] & 0xFF;
                        }
                }

 byte[] bytes = msg.getBytes();
 StringBuilder binarymsg = new StringBuilder();
 for (byte b : bytes)
 {
   int val = b;
   for (int i = 0; i < 8; i++)
   {
     binarymsg.append((val & 128) == 0 ? 0 : 1);
     val <<= 1;
   }
   binarymsg.append(' ');
 }
```

```
int sizekey = keybin.length();
      int sizemsg = binarymsg.length();
      int rem = sizemsg%sizekey;

      int padsize = sizekey-rem;
      binarymsg.append(1);
      for(int i=0;i<padsize-1;i++)
      {
       binarymsg.append('0');
      }
     String cipherbinary="";
        int loop = binarymsg.length()/sizekey;
        int k=0;
        for(int j=0 ;j<loop ;j++)
        {
         for (int i = 0; i < keybin.length(); i++)
            {

           if(keybin.charAt(i)!=binarymsg.charAt(k))
           {
           cipherbinary  += '1';
           k++;
           }
           else
           {
               cipherbinary += '0';
               k++;
           }
        }
        }
          byte[] bval = new BigInteger(cipherbinary, 2).toByteArray();
          String s = new String(bval, "US-ASCII");
            System.out.println();

            System.out.println("Sosemanuk Cipher :");
            System.out.println("*****************");
            System.out.println(s);
    }

}
```

```
public void setKey(byte[] key)
	{
		if (key.length < 1 || key.length > 32)
			throw new Error("bad key length: " + key.length);
		byte[] lkey;
		if (key.length == 32) {
			lkey = key;
		} else {
			lkey = new byte[32];
			System.arraycopy(key, 0, lkey, 0, key.length);
			lkey[key.length] = 0x01;
			for (int i = key.length + 1; i < lkey.length; i ++)
				lkey[i] = 0x00;
		}……………………..

public void setIV(byte[] iv)
	{
		if (iv == null)
			iv = new byte[0];
		if (iv.length > 16)
			throw new Error("bad IV length: " + iv.length);
		byte[] piv;
		if (iv.length == 16) {
			piv = iv;
		} else {
			piv = new byte[16];
			System.arraycopy(iv, 0, piv, 0, iv.length);
			for (int i = iv.length; i < piv.length; i ++)
				piv[i] = 0x00;
		}…………………………….
//A long coding part is not shown in above (…………..) blank part.
```