CHAOS BASED BLOCK ENCRYPTOR

By

FARHANA BT SARIN

FINAL PROJECT REPORT

Submitted to the Electrical & Electronics Engineering Programme
in Partial Fulfillment of the Requirements
for the Degree
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Universiti Teknologi Petronas
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

CHAOS BASED BLOCK ENCRYPTOR

by

Farhana bt Sarin

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Approved:

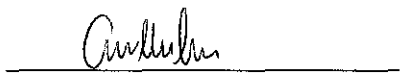
AP. Dr. Varun Jeoti
Project Supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

June 2008

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.



FARHANA BT SARIN

ABSTRACT

This project is basically a block encryption to protect block of data. In this project we will propose encryption algorithm which offer less computational complexity and also provide securing data not only during transmission but also during storage. For this project, the studies focus on Advanced Encryption Standard (AES) and H-CES (Hybrid Chaotic Encryption Scheme) algorithms which are to be used in order to implement this project. The coding of encryption and decryption algorithms is created using C language. The coding compilation is using Borland C++ program. This report is the documentation about the work that has been carried out on the project. This report is divide into several parts which is the introduction, literature review, methodology, result and discussion and conclusion.

ACKNOWLEDGEMENTS

Greatest praise to Allah for His blessing that I am able to complete my final year project within the time frame.

I would like to express my sincere thanks to my supervisor, Assoc. Prof. Dr. Varun Jeoti, Lecturer of Electrical and Electronic Engineering School, Universiti Teknologi Petronas, Malaysia for his continuous help and guidance throughout the course of this project. Indeed, his particular attention and detailed supervision are valuable for the completion of this project.

A special thanks also to Muhammad Asim, master student from department Electrical and Electronic Engineering, Universiti Teknologi Petronas. I also would like to thank Mrs. Mazeyanti Mohd Ariffin, Lecturer of Information Technology School, Universiti Teknologi Petronas. I am very thankful for their assistants and support that was truly helpful for me.

I also wish to thank and convey my affection to my family and friends for their constant support and encouragement throughout this two semester duration of final year project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AES	Advanced Encryption Standard
Bit	A binary digit having a value of 0 or 1
H-CES	Hybrid Chaotic Encryption Scheme
IC	Initial Condition
Key - K	A set of numbers/parameters that determine the specific encryption mechanism.
PLCM	Piecewise Linear Chaotic Map
S-box	Non-linear substitution table used in several byte substitutions transformations and in the Key Expansion routine to perform a one for one substitution of a byte value.
Cryptography	The study of how to design good encryption algorithms for converting information from its normal, comprehensible form into an incomprehensible format, rendering it unreadable without secret key.
Encryption	In cryptography, encryption is the process of obscuring information to make it unreadable format with secret key.
Decryption	In cryptography, decryption is the process of converting back the obscured information into a readable format with secret key.
Cipher	In cryptography, cipher is an algorithm for performing encryption/decryption. Accordingly, the encryption algorithm is called encipher, while the decryption algorithm is called decipher.

CHAPTER 1
INTRODUCTION

1.1 Background of Study

During the last decade, there has been an explosive growth of the using computers, networks, communications and multimedia application [7]. In area of business, military or politics, it is very important to make sure that all sensitive data being safely store and protected. The data encryption has become an important and high profile issue because it is a powerful technique for preventing unwanted interception and viewing of our valuable data. It also can provide security and privacy needs in various applications to avoid any attack from unauthorized parties [2].

Encryption is the process of changing data into a form that can be read only by the intended receiver. To decipher the message, the receiver of the encrypted data must have the proper decryption key (password). This is the basic process that been used to provide security of any transmission data.

In the last decade many encryption schemes based on chaos have been proposed and have attracted due attention for encrypting the information. Chaos and cryptography have some common features – the most prominent being sensitivity to small changes in variables and parameters [1]. However, an important difference between chaos and cryptography lies in the fact that systems used in a chaos are defined on real numbers, while cryptography deals with systems defined on finite sets of integers [1].

1.2 Problem Statement

A look at most chaos based cryptosystems suggests that the apparently random-like chaotic sequence masking the plain data can not hide the data very well. The data can be easily recovered with relatively low computational load by the master of cryptography. Chaos-based cryptographic schemes using chaotic masking is still in its infancy with many algorithms and cryptanalysis methodologies still under development [1].

It is seen that the security of the chaotic ciphers proposed by various researchers is suspect against the standard techniques of cryptanalysis [6]. Most of the chaotic ciphers prove weak against the current prevailing cryptanalytic techniques and do not possess the S-box equivalent security that the traditional ciphers can provide [1].

1.3 Objective and Scope of Study

The objective of this project is to entails securing block of data not only during transmission but also during storage. The scope of study includes two encryption schemes – Advanced Encryption Standard (AES) and Hybrid Chaotic Encryption Scheme (H-CES). The H-CES scheme that been proposed in this project use AES S-box as one of the method in the encryption techniques.

CHAPTER 2
LITERATURE REVIEW

In this chapter, some introduction about AES is described to understand about how the standard data encryption is done. This chapter is dividing into 3 part; first part: description about AES, second part: AES cipher operation and last part: AES inverse cipher operation.

2.1 Advanced Encryption Standard (AES)

AES stands for Advanced Encryption Standard. AES is a symmetric key encryption technique which will replace the commonly used Data Encryption Standard (DES). It was developed by two Belgian cryptologists, Vincent Rijmen and Joan Daemen who win a worldwide call for the submissions of encryption algorithms issued by the US Government's National Institute of Standards and Technology (NIST) in 1997 and completed in 2000. AES provides strong encryption and secure enough to protect information that disclosed to the public [2].

The AES algorithm uses one of three cipher key strengths: a 128, 192 or 256-bits encryption key (password) to encrypt blocks with a length of 128, 192 or 256-bits [3]. Both block length and key length can be extended very easily to multiples of 32-bits. Each encryption key size causes the algorithm to behave slightly differently, so the increasing key sizes not only offer a larger number of bits with which you can scramble the data, but also increase the complexity of the cipher algorithm [2].

2.2 AES Cipher

For encryption, AES cipher consists of the following transformations which are applied repeatedly [2]:

- Substitution Bytes Transformation
- Shift Rows Transformation
- Mix Columns Transformation
- Add Round Key Transformation

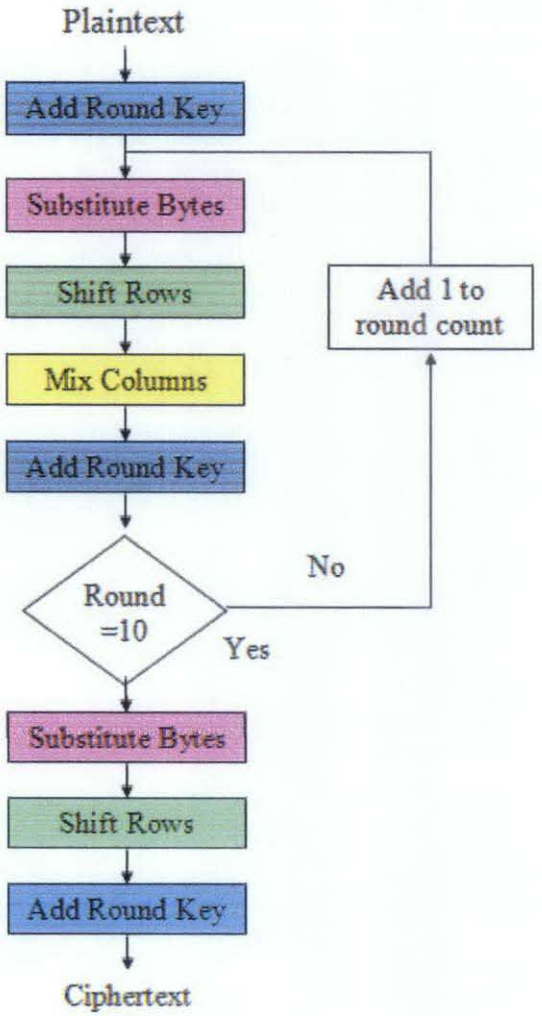


Figure 2.1: Flow of encryption process

2.2.1 Substitution Byte Transformation (SubBytes)

Use S-box 16 x 16 matrixes that contain a permutation of all possible 256 8-bit values. Each individual byte is mapped into a new byte by the leftmost 4-bit of the byte are used as a row value and the rightmost 4-bit are used as a column value [2].

The pseudo-C looks like this:

a (i, j) = SBox [a (i, j)]

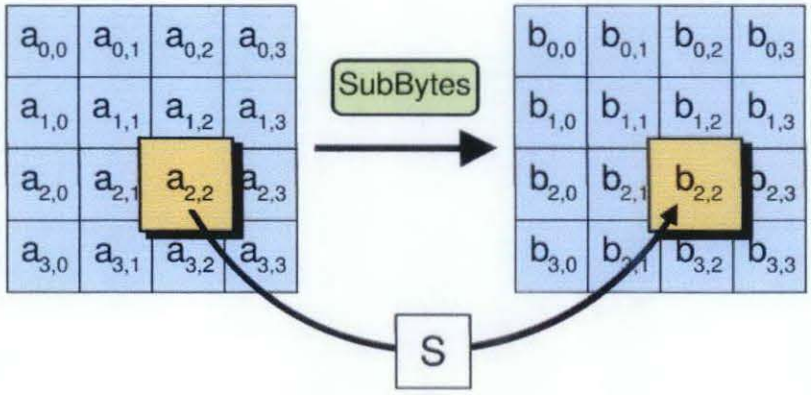


Figure 2.2: Illustration of substitution byte transformation [3]

Table 2.1: AES S-Box Data [2]

	Column															
Row		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table 2.2: Example of substitution byte transformation

Input Data				Substitution Bytes Transformation			
EA	04	65	85	87	F2	4D	97
83	45	5D	96	EC	6E	4C	90
5C	33	98	B0	4A	C3	46	E7
F0	2D	AD	C5	8C	D8	95	A6

2.2.2 Shift Rows Transformation (ShiftRows)

It has pattern to shift the rows. The first row of state is not altered. For the second row, a 1-byte circular left shift is performed. For the third row, a 2-byte circular left shift is performed. For the fourth row, a 3-byte circular left shift is performed [2].

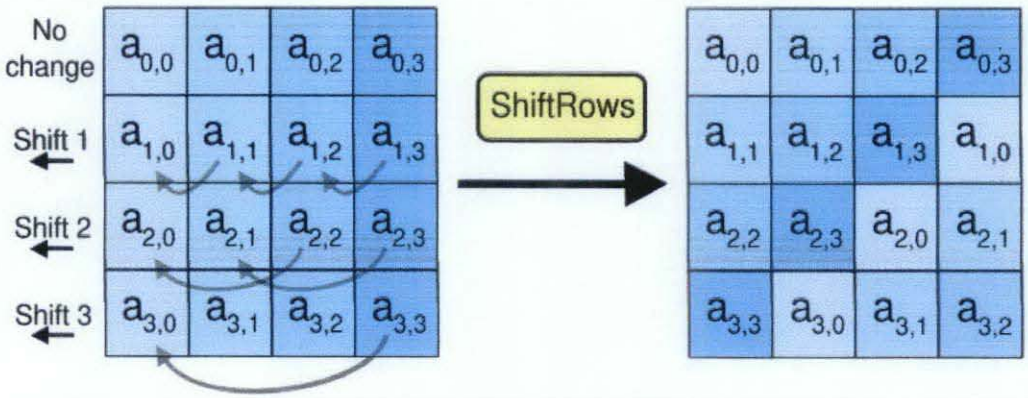


Figure 2.3: Illustration of shift rows transformation [3]

2.2.3 Mix Columns Transformation (MixColumns)

This transformation operates on each column individually. Each byte of a column is mapped into a new value that is function of all four byte in the column. The input data will be XOR with fixed polynomial matrix, c(x) to get the mix columns values.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Figure 2.4: Fix polynomial matrix c(x) values [2]

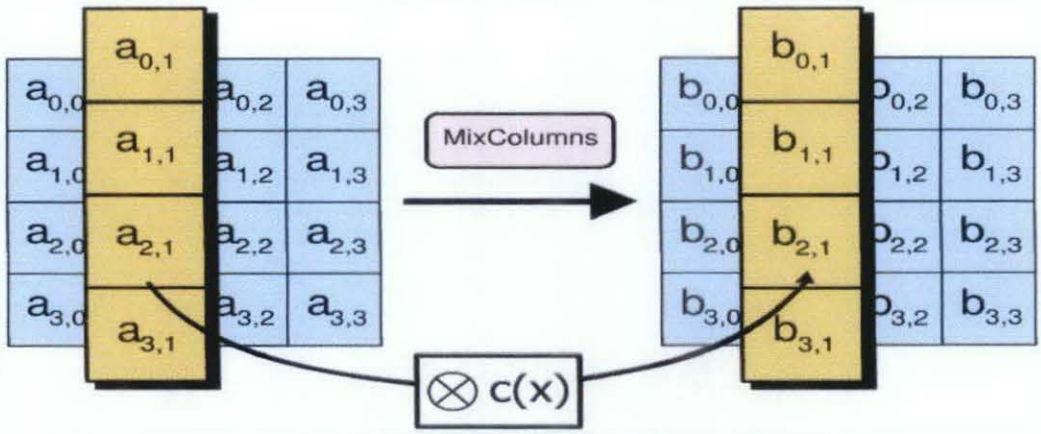


Figure 2.5: Illustration of mix columns transformation [3]

2.2.4 Add Round Key Transformation

The key schedule is responsible for expending a short key into a larger key, whose parts are used during the different iterations. Each key size is expanded to a different size [2]:

A 128 bit key is expanded to a 176 byte key.

A 192 bit key is expanded to a 208 byte key.

A 256 bit key is expanded to a 240 byte key.

There is a relation between the cipher key size, the number of rounds and the expanded key size. For a 128 bit key, there is one initial AddRoundKey operation plus there are 10 rounds and each round need a new 16 byte key, therefore it require 10+1 RoundKey of 16 byte, which equals 176 byte.

The general formula is [2]:

ExpandedKeySize = (numberRound+1) * BlockSize

The 128bits of state are bitwise XORed with the 128bits of the round key. The first matrix is State and the second matrix is the round key.

2.3 AES Inverse Cipher

For decryption, the inverse cipher consists of the following individual transformations [2]:

- Inverse Shift Row Transformation
- Inverse Substitution Bytes Transformation
- Inverse Mix Columns Transformation
- Add Round Key Transformation

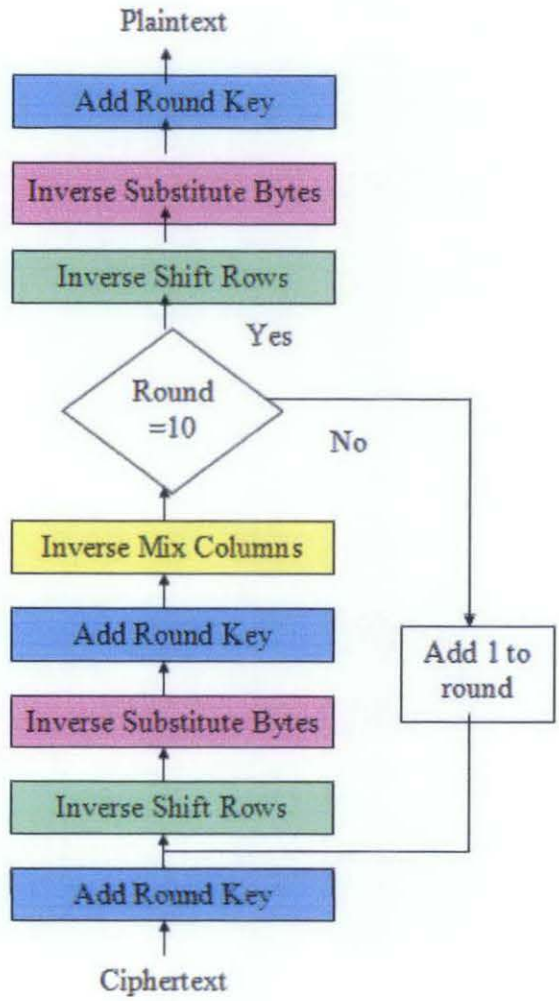


Figure 2.6: Flow of decryption process

2.3.1 Inverse Shift Rows Transformation

This transformation performs the circular shifts in the opposite direction from the forward shift rows transformation of AES cipher of each last three rows.

2.3.2 Inverse Substitution Bytes Transformation

The process is similar to forward substitution but using different table which called inverse S-box.

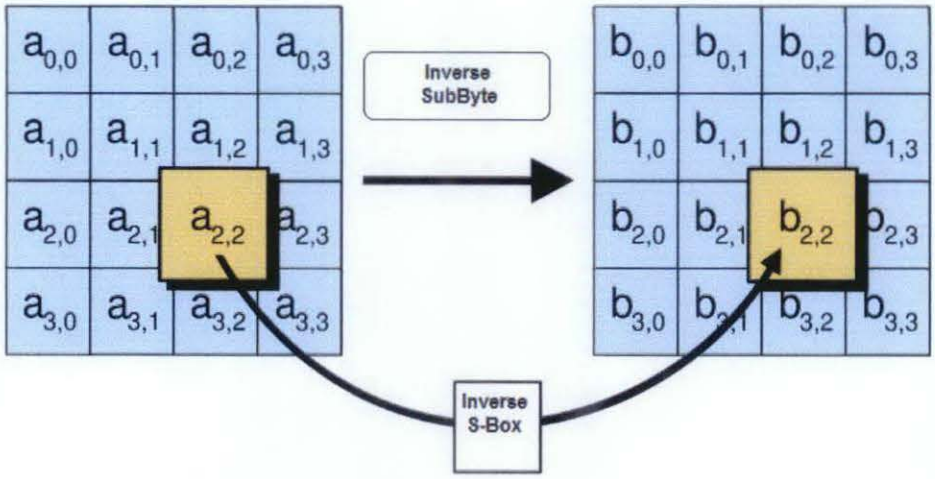


Figure 2.7: Illustration of inverse substitution byte transformation [3]

Table 2.3: Inverse AES S-Box Data [2]

		Column															
Row		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Table 2.4: Example of inverse substitution byte transformation

Input Data				Inverse Substitution Bytes Transformation			
87	F2	4D	97	EA	04	65	85
EC	6E	4C	90	83	45	5D	96
4A	C3	46	E7	5C	33	98	B0
8C	D8	95	A6	F0	2D	AD	C5

2.3.3 Inverse Mix Columns Transformation

This is reverse process of forward mix column. Fixed polynomial for inverse mix column is different from forward mix column.

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$$

Figure 2.9: Inverse fix polynomial matrix c'(x) values [2]

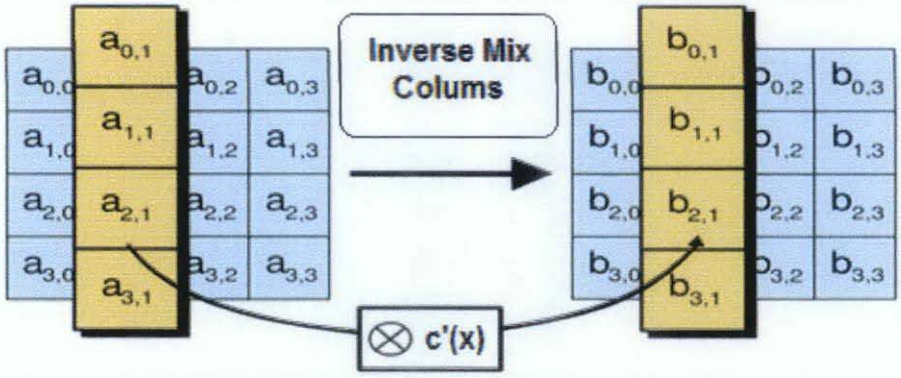


Figure 2.10: Illustration of inverse mix columns transformation [3]

2.3.4 Add Round Key Transformation

This formation is identical to the forward add round key transformation because the XOR operation is its own inverse.

CHAPTER 3
IMPLEMENTATION H-CES

In this chapter, the proposed encryption technique hybrid - Hybrid Chaotic Encryption Scheme (H-CES) is described. H-CES works as a 128 bit block cipher similar to AES. H-CES is a cryptosystem that employs multiple PLCMs, a generalized logistic map, an AES S-box and ciphertext feedback [1]. However, before going into detailed discussion of H-CES, some introduction about PLCM and generalized logistic map is described.

3.1 Piecewise Linear Chaotic Map (PLCM)

PLCM is a 1-D chaotic map that has been widely explored and used. As we have already mentioned, chaotic systems operate on real numbers. Generally, given a real interval, $X = [\alpha, \beta] \subset R$, a piecewise linear chaotic map $F: X \rightarrow X$ is a multi-segmented map such that for $i = 1$ to m, $C_i = F_i(x) = a_i x + b_i$. Any $\{C_i\}_{i=1}^{m}$ constitutes a partition of X, and $\cup_{i=1}^{m} C_i = X$. Also $Ci \cap Cj = \emptyset$, $\forall i \neq j$. Such a map has the following statistical properties [1]:

- It is chaotic since its Lyapunov exponent is positive
- It is exact
- It has the property of mixing
- It is ergodic
- It has uniform invariant density function $f(x) = 1/(\beta - \alpha)$. The uniform invariant density function means that uniform density input will generate uniform density output, and the chaotic orbit from almost every initial condition will lead to the same uniform distribution $f(x) = 1/(\beta - \alpha)$.

3.2 Generalized Chaotic Logistic Map

Generalized logistic map is a discrete map, suitably modified from its continuous counterpart popularly known as logistic map. The modification is made so as to obtain integer values ranging from 0 to 255 [1].

$$f(y_i) = \begin{cases} \left\lfloor \left[\dfrac{y_i(256 - y_i)}{64} \right] \right\rfloor & \tilde{y}_i < 256 \\ 255 & \tilde{y}_i = 256 \end{cases} \tag{3.1}$$

Here $f(y_i)$ is the output of logistic map, y_i is the input to the logistic map given by [1]

$$\tilde{y}_i = \left\lfloor \left[\frac{y_i(256 - y_i)}{64} \right] \right\rfloor. \tag{3.2}$$

3.3 AES S-Box

The proposed encryption technique H-CES, will use the substitution bytes transformation as one of the implementation methods. Each individual byte is mapped into a new byte by the leftmost 4-bit of the byte are used as a row value and the rightmost 4-bit are used as a column value by referring to the AES S-box table [2]. In the proposed decryption technique also will used the inverse substitution bytes transformation using inverse S-box table to get back the original plaintext.

3.4 H-CES Encryption Algorithm

There are 4 steps for the encryption part. Firstly, a unique initial condition is been extract from the external 16 bits of secret key. Secondly, 16 chaotic key stream is generated by using piecewise linear chaotic map (PLCM). Thirdly, substitution bytes transformation is done using AES S-Box values. Lastly, masking and mixing operation is done based on generalized logistic map and ciphertext feedback to produce encryption data. Below is the brief explanation about each step that involved in encryption algorithms.

3.4.1 Extracting a unique Initial Condition from External Secret Key

First step, the initial condition is derived from the external secret key of 16 characters. The secret key is denoted by [1]:

$$K = K_1 \ K_2 \ K_3 \cdots K_{16} \tag{3.1}$$

The K_i is denotes as the 8-bit block of secret key. By using K_i, the i^{th} initial condition IC_i is calculated [1]:

$$IC_i = K_i / 256 \tag{3.2}$$

3.4.2 Generating Chaotic Key Stream using Piecewise Linear Chaotic Map (PLCM)

Second step, the initial condition IC is derived using PLCM condition. Where p is refer to control parameter lies between $0 < p < 1/2$, and x serves as the initial condition [5].

$$CM(x,p) = \begin{cases} x/p & 0 \le x \le p \\ (x-p)/(1/2-p) & p \le x < 1/2 \\ CM(1-x,p) & 1/2 < x \le 1 \end{cases}, \tag{3.3}$$

The values of 16 control parameters p_i that are used in this analysis are:

$$\begin{bmatrix} p1 & p2 & p3 & p4 \\ p5 & p6 & p7 & p8 \\ p9 & p10 & p11 & p12 \\ p13 & p14 & p15 & p16 \end{bmatrix} = \begin{bmatrix} 0.15 & 0.17 & 0.188 & 0.199 \\ 0.222 & 0.444 & 0.143 & 0.23 \\ 0.235 & 0.444 & 0.48 & 0.222 \\ 0.123 & 0.456 & 0.33 & 0.35 \end{bmatrix} \quad (3.4)$$

This will result 16 PLCMs for masking the input plain block of 16 bytes as shown below [1]:

$$CM_1 = CM_1(IC, p_1), \cdots\cdots, CM_{16} = CM_{16}(IC, p_{16}) \quad (3.5)$$

The chaotic key stream is form based on the m-PLCMs that have been generated earlier. The ϕ_i is represented in 8-bits as the key stream from i[th] PLCM [1].

$$\phi_i(k) = \lfloor CM_i * (10^7) \rfloor (\text{mod } 256) \quad where\ i = 1,2,3,\cdots,16 \quad (3.6)$$

All together 128-bit chaotic key stream denoted by ø that is represented in the matrix form as shown below [1]:

$$\phi(k) = [\phi_1(k) \quad \phi_2(k) \quad \cdots \quad \phi_{16}(k)] \quad (3.7)$$

3.4.3 Substitution by AES S-Box

Third step, each byte in the 128 bit input plain block is substituted by S-box, similar to AES. Let the input to CES cipher be donated as x where xi represents an 8-bit block in the input 128 bit plain block. The output of the substitution S-box transformation is denoted as sx [1].

$$x = [x_1, x_2, x_3, \cdots\cdots, x_{16}] \quad (3.8)$$

$$sx = [sx_1, sx_2, sx_3, \ldots\ldots, sx_{16}] \quad (3.9)$$

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3.1: AES Forward S-Box Values [2]

3.4.4 Masking and Mixing: based on generalized logistic map and ciphertext Feedback

This is the final step for H-CES encryption. The output of the generalized logistic map is XORed with the output of substituted by the S-box. The input to the generalized logistic map is chaotic key XORed with ciphertext feedback [1].

$$\underbrace{sx_1 \oplus f(\phi(1) \oplus x'_{16})}_{x'_1}, \underbrace{sx_2 \oplus f(\phi(2) \oplus sx_1)}_{x'_2}, \underbrace{sx_3 \oplus f(\phi(3) \oplus x'_2)}_{x'_3}, \cdots\cdots$$

$$, \underbrace{sx_N \oplus f(\phi(16)) \oplus x'_{15})}_{x'_{16}}, \tag{3.10}$$

3.5 H-CES Decryption Algorithm

For the decryption part, there are 2 steps involved. Firstly, do the inverse masking and mixing operation. Then, do the inverse substitution by AES S-Box. Below are the detail explanations on each step in decryption algorithms.

3.5.1 H-CES Inverse Cipher

First step, let the input to H-CES inverse cipher be denoted as x_i' where it represents 8-bit block in the input 128 bit cipher block [1].

$$x_i' = [x_1', x_2', x_3', \ldots, x_{16}'] \tag{3.11}$$

3.5.2 Inverse of Masking and Mixing

Second step, reverse the order of key stream where the ϕ^n is used in the first round of decryption, then ϕ^{n-1} and ϕ^1 is used in the last round of decryption. A is represent the reverse order of extracted key streams from 16-PLCM [1].

$$A = [\phi^n \ \phi^{n-1} \ \cdots \phi^2 \ \phi^1] \tag{3.12}$$

During decryption process, each byte in the input matrix is XORed with the output of the generalized logistic map and then inverse substituted by the inverse S-box. The inverse of masking and mixing transformation is denoted as [1]:

$$\underbrace{x_1' \oplus f(\phi^n(1) \oplus x_N')}_{sx_1}, \underbrace{x_2' \oplus f(\phi^n(2) \oplus sx_1)}_{sx_2}, \underbrace{x_3' \oplus f(\phi^n(3) \oplus x_2')}_{sx_3}, \cdots\cdots$$
$$, \underbrace{x_N' \oplus f(\phi^n(16)+1) \oplus x_{N-1}')}_{sx_{16}}, \tag{3.13}$$

3.5.3 Inverse of substitution by S-box

The final step is the inverse of the byte substitution by S-box. The inverse S-box is shown as below:

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
10	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
20	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
30	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
40	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
50	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
60	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
70	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
80	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
90	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a0	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b0	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c0	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d0	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e0	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f0	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 3.2: AES Inverse S-Box Values [2]

CHAPTER 4
METHODOLOGY

In this chapter, the flow of coding implementation will be discussed. Start with H-CES encryption coding, followed by H-CES decryption coding and lastly H-CES coding implementation.

4.1 H-CES Encryption Coding

For the H-CES encryption part, firstly, user is required to enter 16-bits integers as original input (plaindata). If the program detect complete 16 bits integer, than the program will read the define values such as external key, P control value and S-box value. Next, program will read the define function such as generalized logistic map and piecewise linear chaotic map (PLCM). Based on PLCM function, chaotic key streams will be generated. The final encryption steps is masking and mixing operation to produce encryption data and the encryption result will be display. Refer figure 4.1 for the flow of encryption coding.

4.2 H-CES Decryption Coding

For the H-CES decryption part, the program will read encryption data as input. If the data is readable then the program will read the define values such as inverse S-box and chaotic key streams. Next, the program will read the define function generalized logistic map and get inverse S-box value. The final operation is inverse masking and mixing to get back the original data. This program also will display the decryption data. Refer figure 4.2 for the flow of decryption coding.

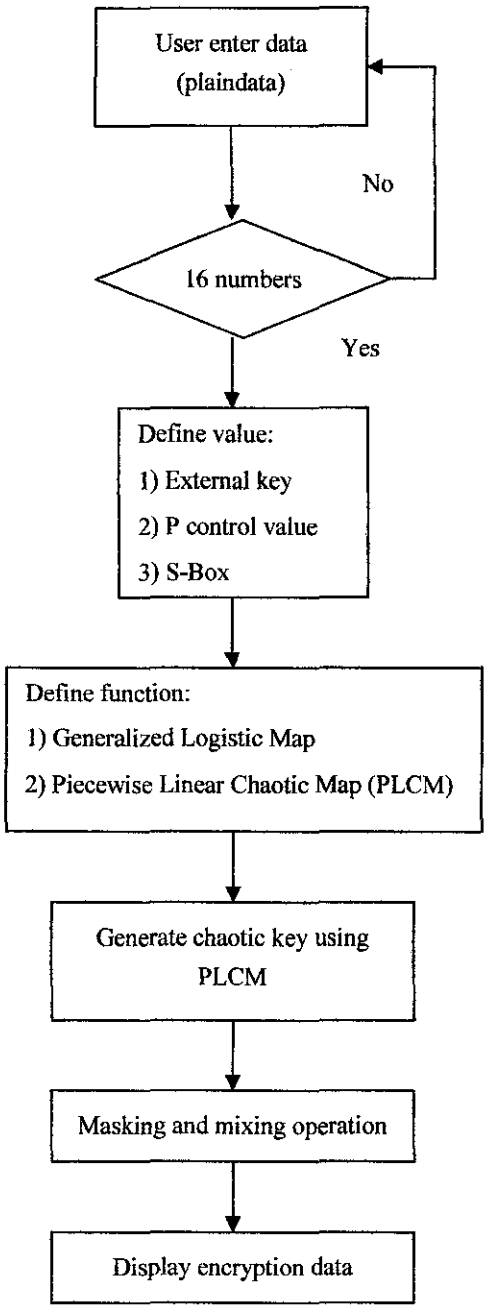


Figure 4.1: Flow of encryption coding

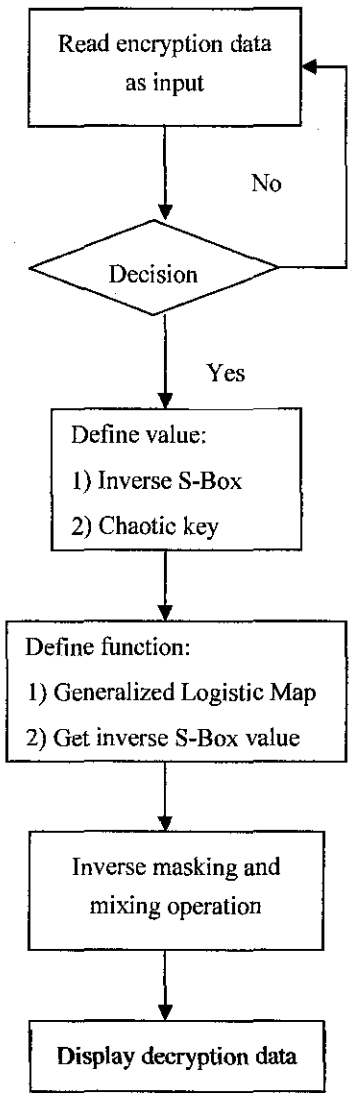


Figure 4.2: Flow of decryption coding

4.3 H-CES Coding Implementation

In this project, all the operation involved in encryption and decryption parts are created using C-language programming. Compilation for the complete coding that has been created is using Borland C++ program. Refer at Appendix A for complete project coding.

Figure 4.3: Symbol Borland C++ program

CHAPTER 5
RESULTS AND DISCUSSION

In the previous chapter, each and every step of the proposed H-CES has been described in detail. The methodology that been chosen can provide enough security protection for our original data not only during transmission but also during storage. In this chapter, the result that been obtain from the coding compilation will be show and discuss.

5.1 Results

If the coding compilation has no error, then result window will shown as figure 5.1 that is asking user to enter set of data. After user finished key in the set of data, result window will change the original data into encryption data and then back to the original data. We have test the coding with 16, 20 and 24 sets of plaintext. For each set we do 4, 5, 6, and 7 iterations of encryption/decryption techniques.

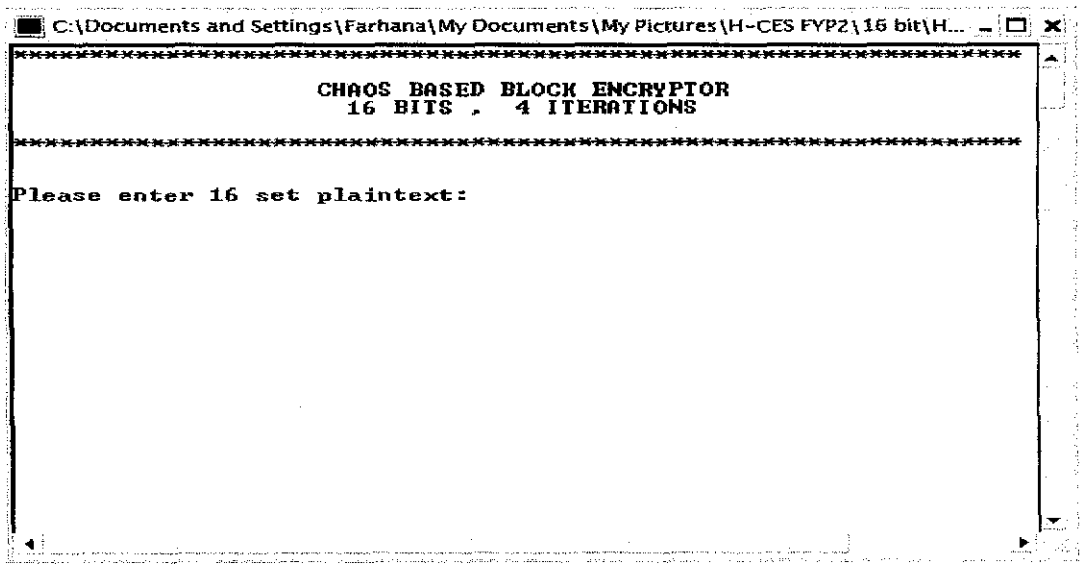


Figure 5.1: Result 1

A) 16 SET OF PLAINTEXT

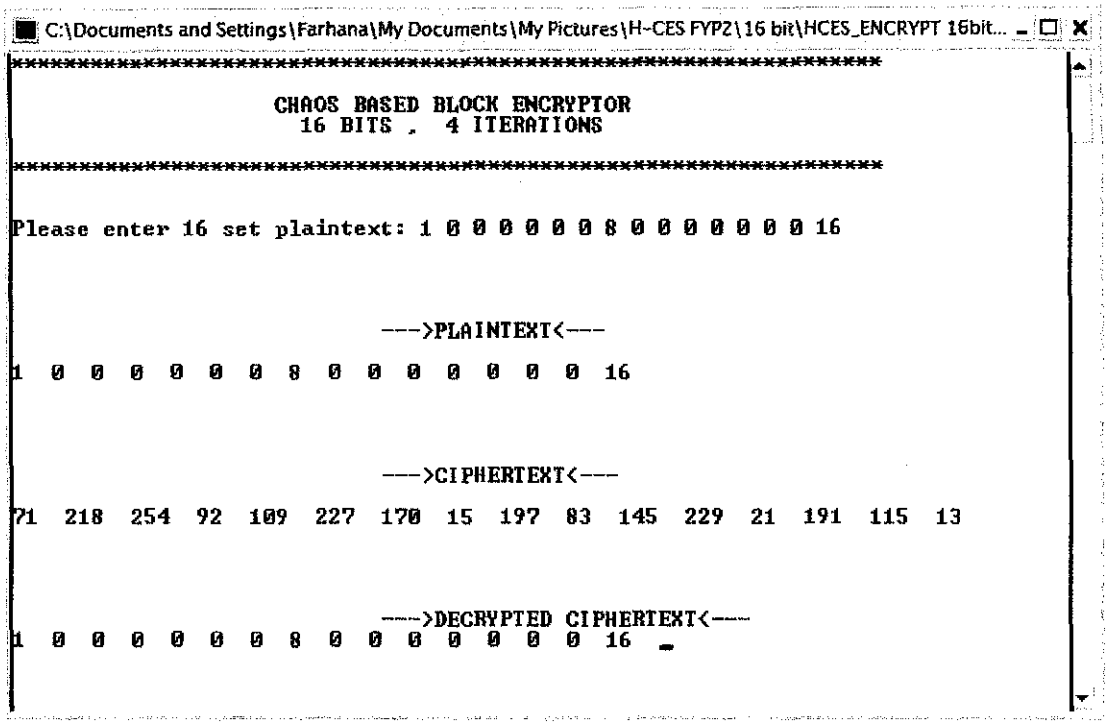


Figure 5.2: 16 bits, 4 iterations

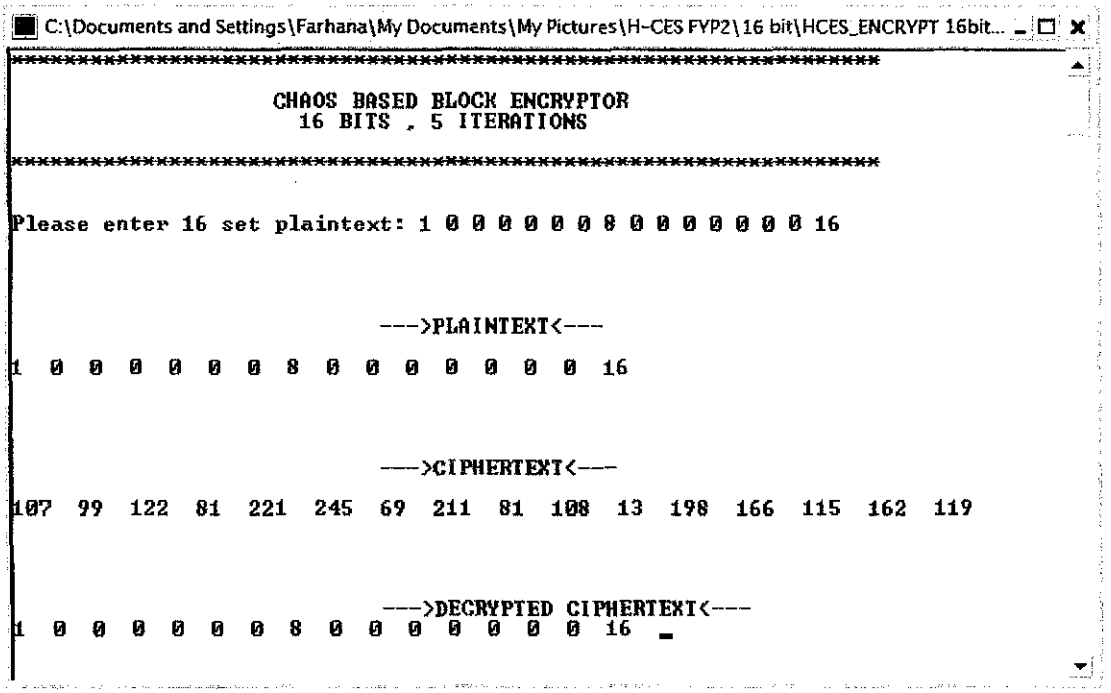


Figure 5.3: 16 bits, 5 iterations

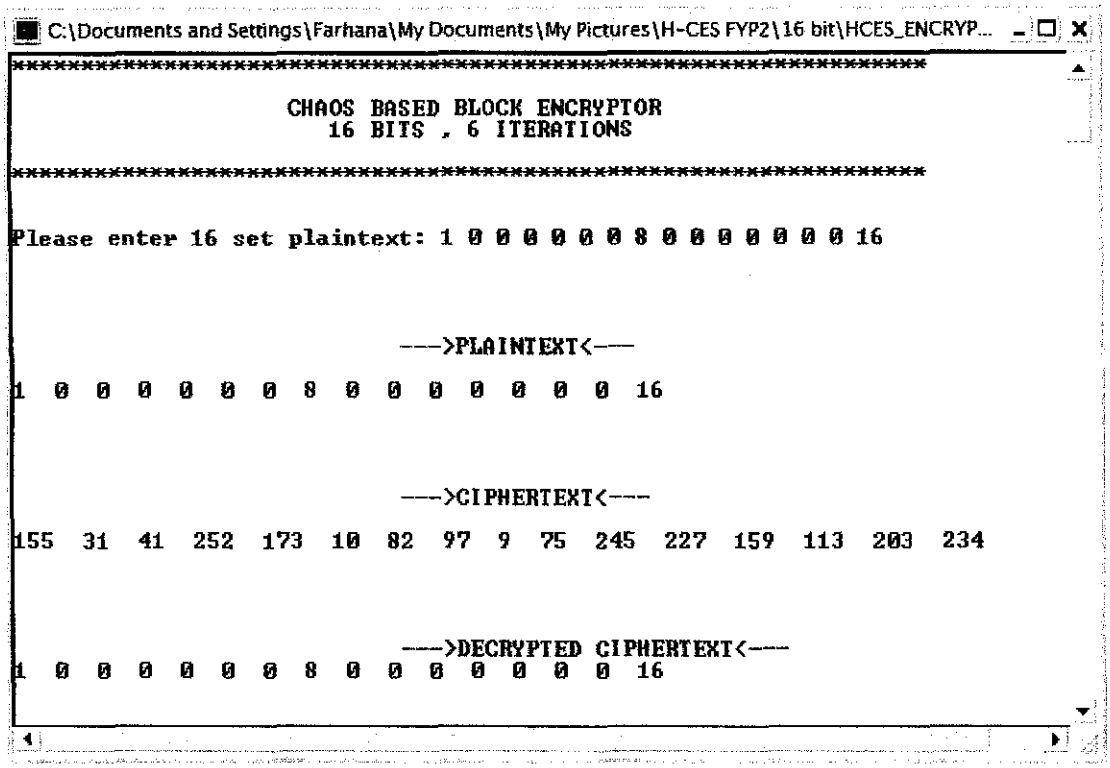


Figure 5.4: 16 bits, 6 iterations

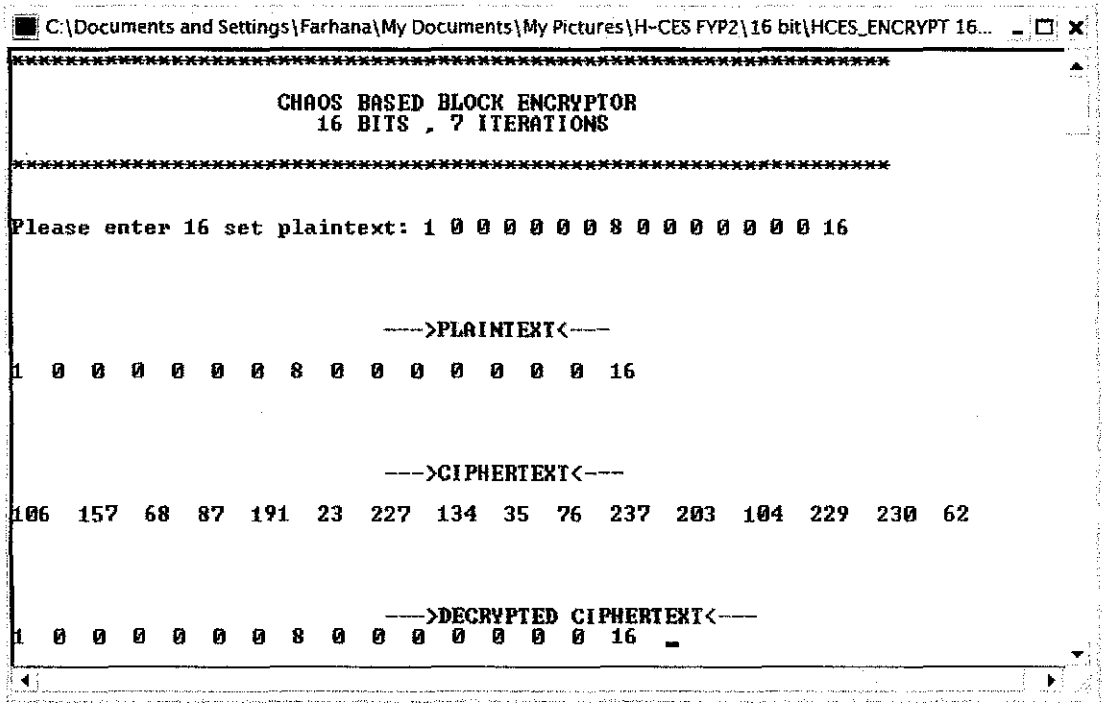


Figure 5.5: 16 bits, 7 iterations

B) 20 SET OF PLAINTEXT

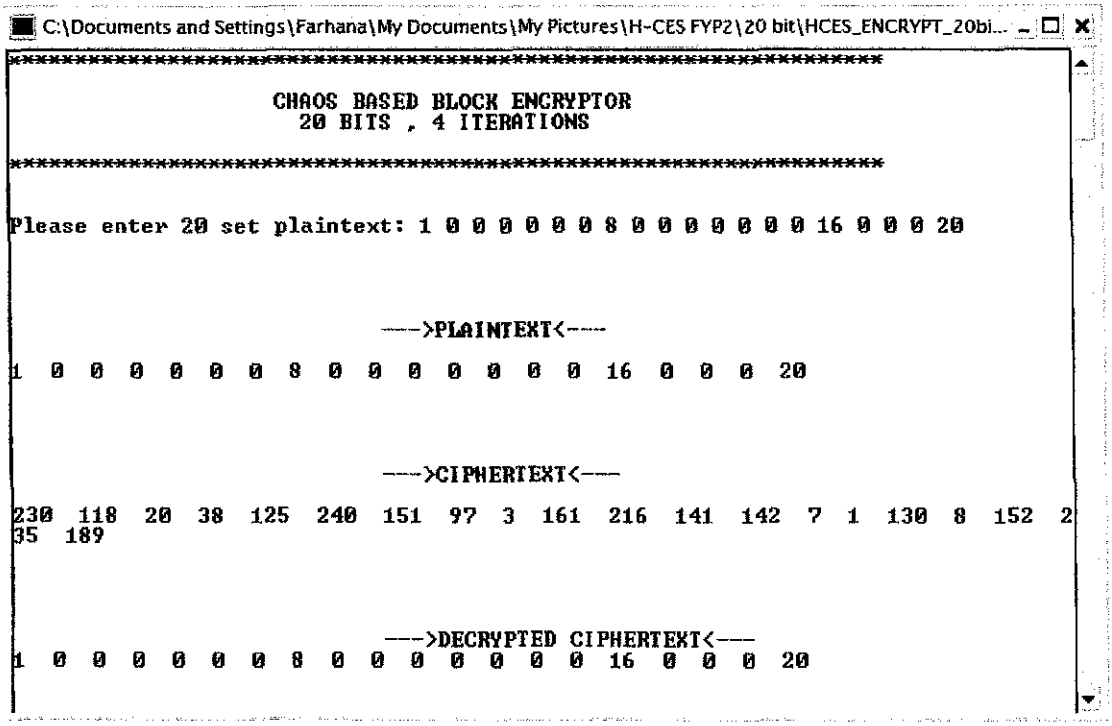


Figure 5.6: 20 bits, 4 iterations

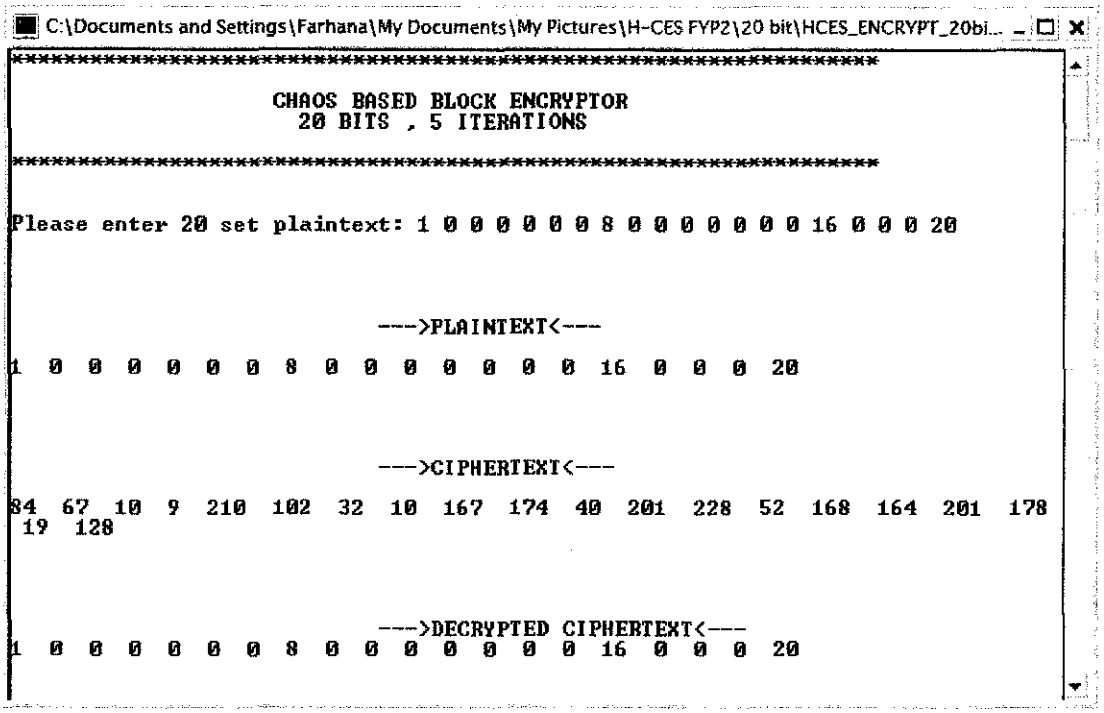


Figure 5.7: 20 bits, 5 iterations

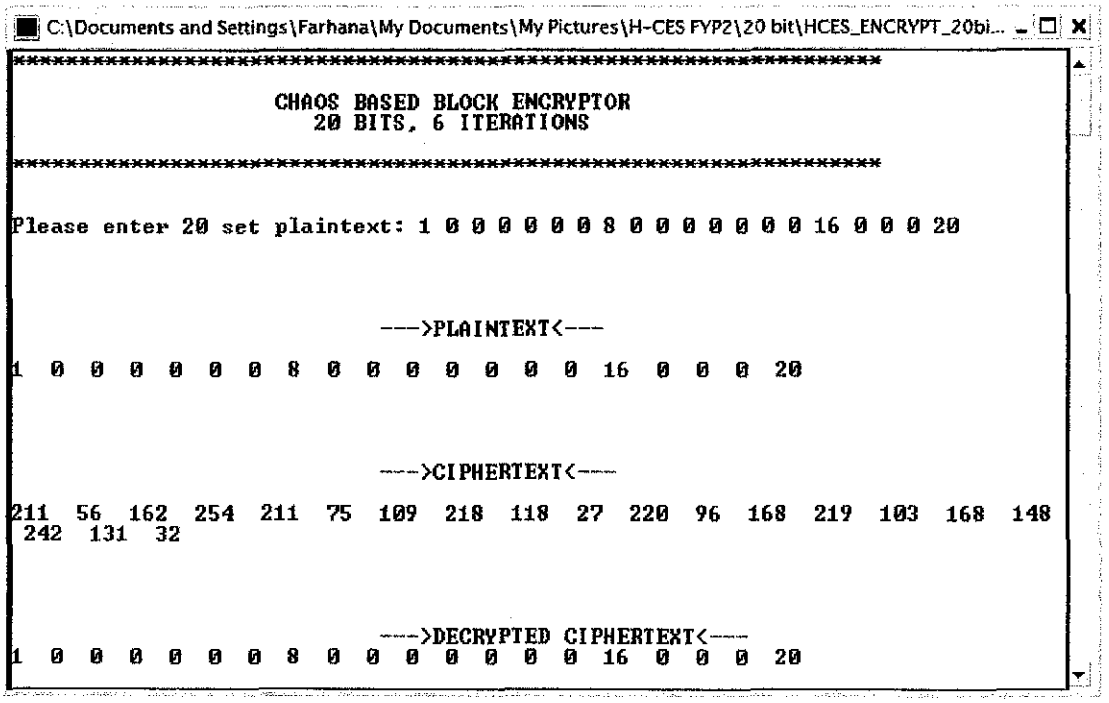


Figure 5.8: 20 bits, 6 iterations

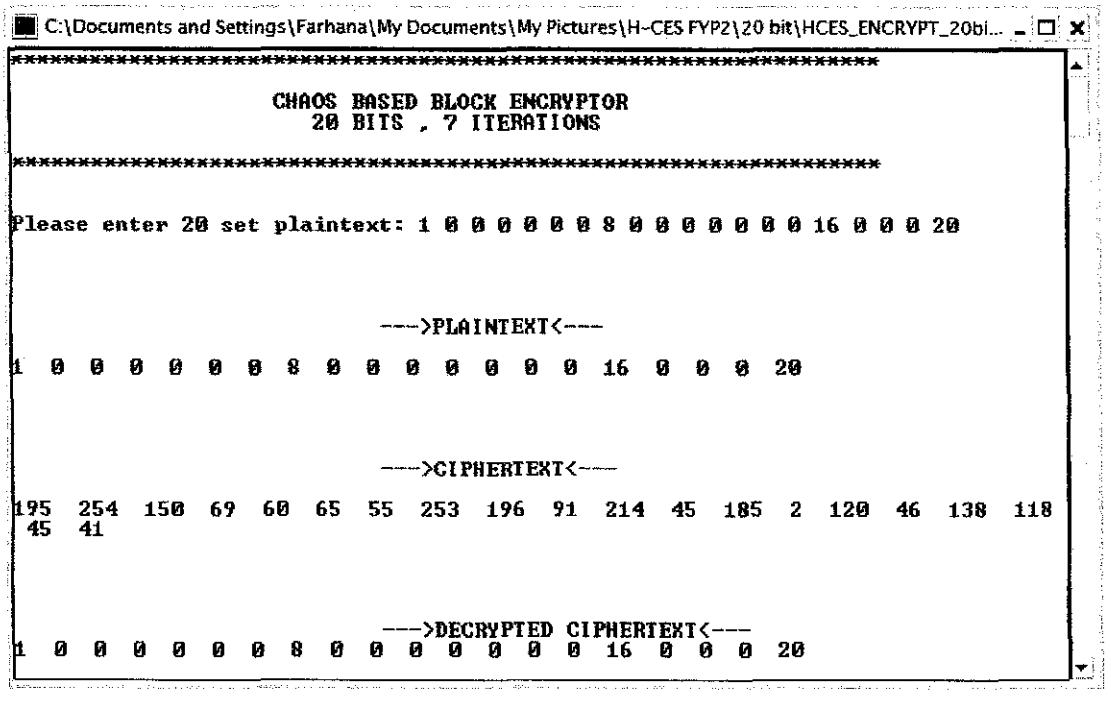


Figure 5.9: 20 bits, 7 iterations

C) 24 SET OF PLAINTEXT

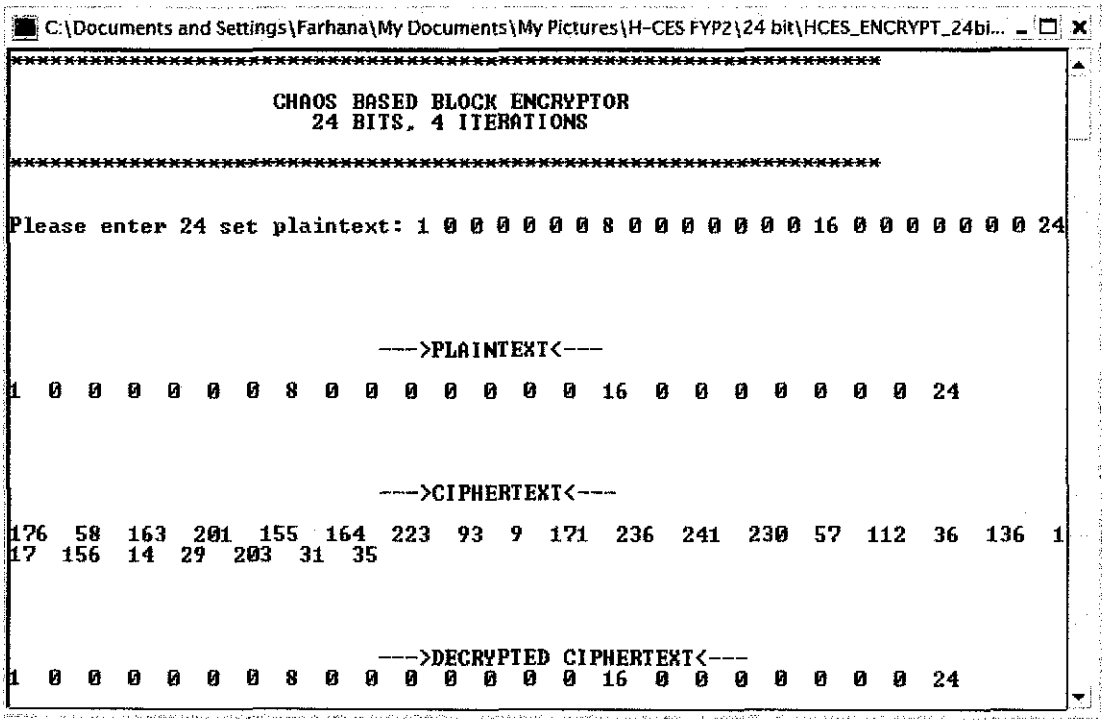


Figure 5.10: 24 bits, 4 iterations

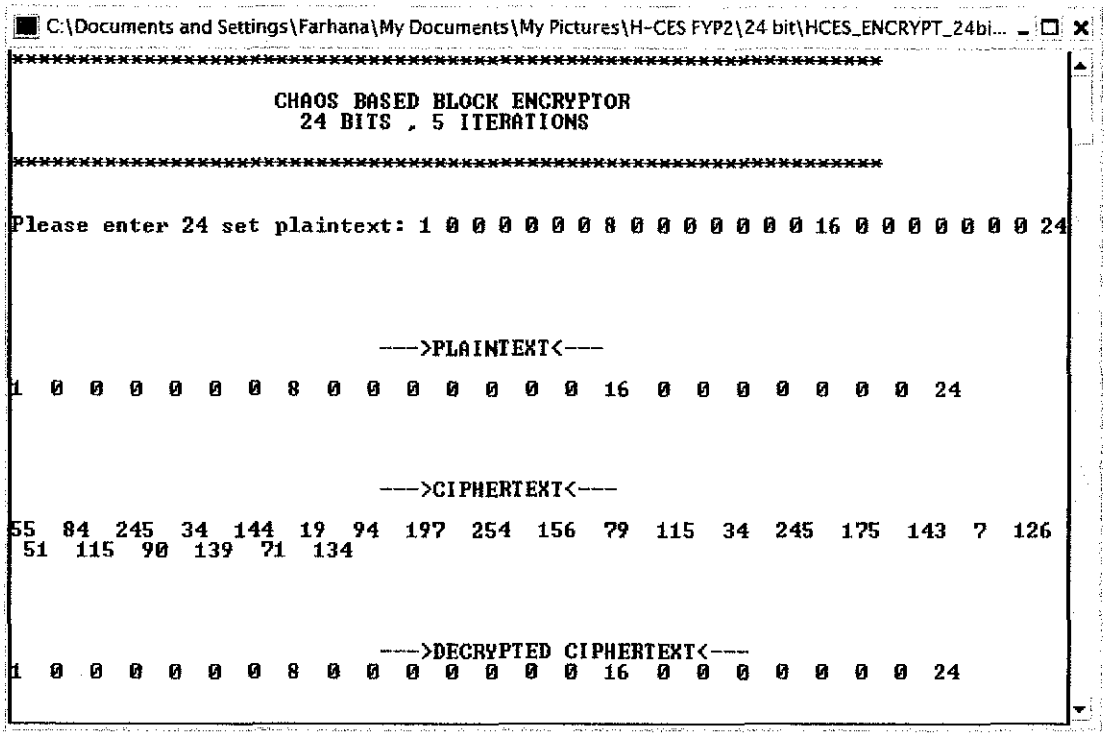


Figure 5.11: 24 bits, 5 iterations

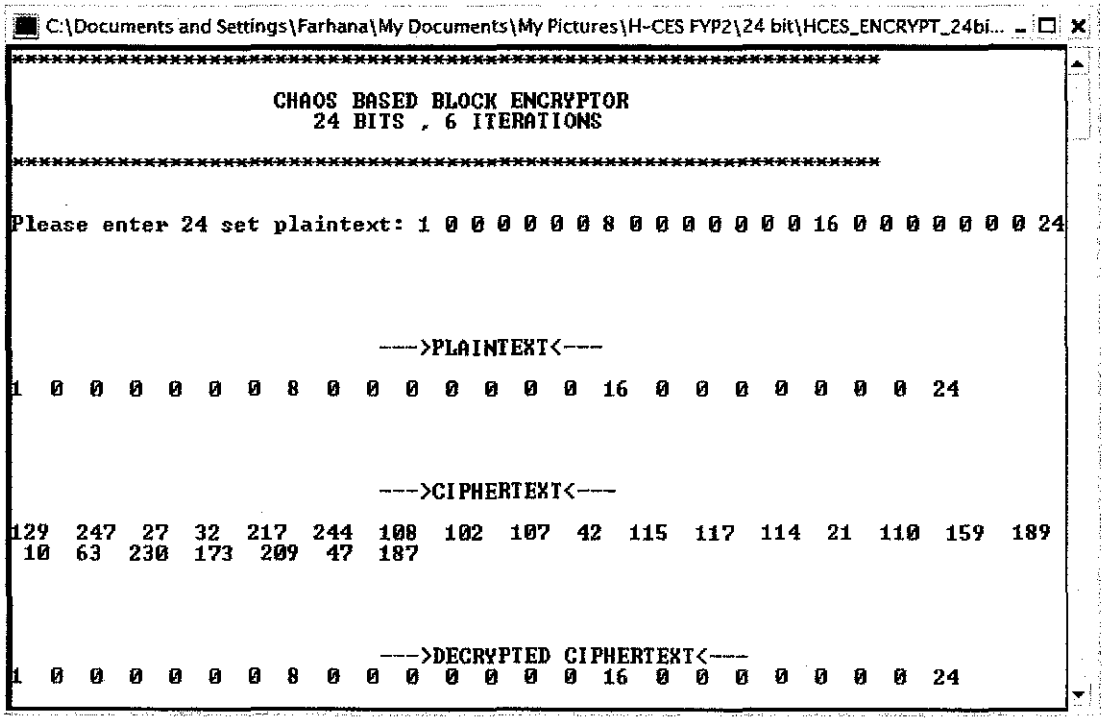


Figure 5.12: 24 bits, 6 iterations

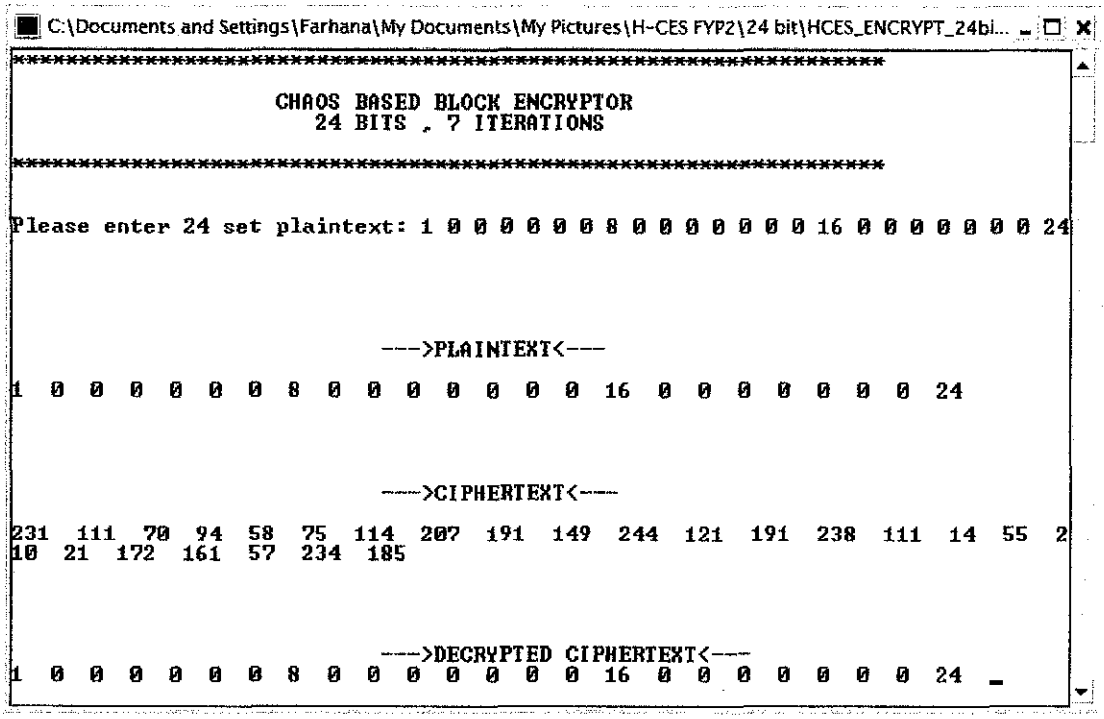


Figure 5.13: 24 bits, 7 iterations

5.2 Discussion

Based on the result of compilation, the input plaintext are been successfully pass the encryption and decryption process to get back the original plaintext data. The encryption produces ciphertext that is difference from one to another although the input plaintext is same. But there are some ciphertext values that been repeated and the frequency of occurrence is not more than 2 times.

Table 5.1: Result 16 bits for all iterations.

No	Input Plaintext Encrypt	Ciphertext Iteration 4	Iteration 5	Iteration 6	Iteration 7	Output Plaintext Decrypt
1	1	71	107	155	106	1
2	0	218	99	31	157	0
3	0	254	122	41	68	0
4	0	92	81	252	87	0
5	0	109	221	173	191	0
6	0	227	245	10	23	0
7	0	170	69	82	227	0
8	8	15	211	97	134	8
9	0	197	81	9	35	0
10	0	83	108	75	76	0
11	0	145	13	245	237	0
12	0	229	198	227	203	0
13	0	21	166	159	104	0
14	0	191	115	113	229	0
15	0	115	162	203	230	0
16	16	13	119	243	62	16

Table 5.2: Analysis result 16 bits for all iterations.

16 bits, 4 iterations																
	No	13	15	21	71	83	92	109	115	145	170	191	197	218	227	229	254
	Freq	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
16 bits, 5 iterations																	
	No	13	69	81	99	107	108	115	119	122	162	166	198	211	221	245	
	Freq	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1	
16 bits, 6 iterations																	
	No	9	10	31	41	75	82	97	113	155	159	173	203	227	243	245	252
	Freq	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
16 bits, 7 iterations																	
	No	23	35	62	68	76	87	104	106	134	157	191	203	227	229	230	237
	Freq	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 5.3: Result 20 bits for all iterations.

No	Input Plaintext Encrypt	Ciphertext				Output Plaintext Decrypt
		Iteration 4	Iteration 5	Iteration 6	Iteration 7	
1	1	230	84	211	195	1
2	0	118	67	56	254	0
3	0	20	10	162	150	0
4	0	38	9	254	69	0
5	0	125	210	211	60	0
6	0	240	102	75	65	0
7	0	151	32	109	55	0
8	8	97	10	218	253	8
9	0	3	167	118	196	0
10	0	161	174	27	91	0
11	0	216	40	220	214	0
12	0	141	201	96	45	0
13	0	142	228	168	185	0
14	0	7	52	219	2	0
15	0	1	168	103	120	0
16	16	130	164	168	46	16
17	0	8	201	148	138	0
18	0	152	178	242	118	0
19	0	235	19	131	45	0
20	20	189	128	32	41	20

Table 5.4: Analysis result 20 bits for all iterations.

20 bits, 4 iterations

No	1	3	7	8	20	38	97	118	125	130	141	142	151	152	161	189	216	230	235	240
Freq	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

20 bits, 5 iterations

No	9	10	19	32	40	52	67	84	102	128	164	167	168	174	178	201	210	228
Freq	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	2	1	1

20 bits, 6 iterations

No	27	32	56	75	96	103	109	118	131	148	162	168	211	218	219	220	242	254
Freq	1	1	1	1	1	1	1	1	1	1	1	2	2	1	1	1	1	1

20 bits, 7 iterations

No	2	41	45	46	55	60	65	69	91	118	120	138	150	185	195	196	214	253	254
Freq	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 5.5: Result 24 bits for all iterations.

No	Input Plaintext Encrypt	Ciphertext Iteration 4	Iteration 5	Iteration 6	Iteration 7	Output Plaintext Decrypt
1	1	176	55	129	231	1
2	0	58	84	247	111	0
3	0	163	245	27	70	0
4	0	201	34	32	94	0
5	0	155	144	217	58	0
6	0	164	19	244	75	0
7	0	223	94	108	114	0
8	8	93	197	102	207	8
9	0	9	254	107	191	0
10	0	171	156	42	149	0
11	0	236	79	115	244	0
12	0	241	115	117	121	0
13	0	230	34	114	191	0
14	0	57	245	21	238	0
15	0	112	175	110	111	0
16	16	36	143	159	14	16
17	0	136	7	189	55	0
18	0	117	126	10	210	0
19	0	156	51	63	21	0
20	0	14	115	230	172	0
21	0	29	90	173	161	0
22	0	203	139	209	57	0
23	0	31	71	47	234	0
24	24	35	134	187	185	24

Table 5.6: Analysis result 24 bits for all iterations.

24 bits, 4 iterations

No	9	14	29	31	35	36	57	58	93	112	117	136	155	156	163	164	171	176	201	203	223	230	236	241
Freq	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

24 bits, 5 iterations

No	7	19	34	51	55	71	79	84	90	94	115	126	134	139	143	144	156	175	197	245	254
Freq	1	1	2	1	1	1	1	1	1	1	2	1	1	1	1	1	1	1	1	2	1

24 bits, 6 iterations

No	10	21	27	32	42	47	63	102	107	108	110	114	115	117	129	159	173	187	189	209	217	230	244	247
Freq	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

24 bits, 7 iterations

No	14	21	55	57	58	70	75	94	111	114	121	149	161	172	185	191	207	210	231	234	238	244
Freq	1	1	1	1	1	1	1	1	2	1	1	1	1	1	1	2	1	1	1	1	1	1

Based on the round of iterations been tested, all of the plaintext data can be recovered back to its original plaintext without having any error. So the system is reliable for encryption and decryption process.

CHAPTER 6
CONCLUSION AND RECOMMENDATION

6.1 Conclusion

Based on the method use in proposed encryption technique, Hybrid Chaotic Encryption Scheme (H-CES) is easier to implement and offer less computational complexity compare to Advanced Encryption Standard (AES). At the same time, H-CES also can provide protection for the user data from any unwanted viewer.

As a conclusion, this project is very interesting and very useful in daily life especially for those who need to transmit sensitive data and need very high security level to protect it from unauthorized people.

6.2 Recommendation

For the recommendation purpose, the way of writing the C-language coding can be improve in order to make it more easier to understand and can offer much less computational complexity.

In the reality cases, the proposed encryption technique also can be implemented for image encryption to hide any image type such as JPEG and BMP types of images.

REFERENCES

[1] Muhammad Asim, "An Image Encryption Scheme based on Chaos and S-Box", Master Science Thesis, Department of Electrical and Electronic Engineering, Universiti Teknologi PETRONAS, 2007.

[2] Joan Daemen and Vincent Rijmen, "The Design of Rijndael AES – The Advanced Encryption Standard, Springer-Verlag, 2002.

[3] "Advanced Encryption Standard", Online URL:http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

[4] Behrouz A.Forouzan, Richard F.Gilberg, "Computer Science A Structured Programming Approach Using C", Second Edition,2001.

[5] Muhammad Asim and Varun Jeoti, "A Novel Efficient and Simple Method for Designing Chaotic S-Boxes, Technical Report, Department of Electrical and Electronic Engineering, Universiti Teknologi PETRONAS,2007.

[6] G.Jakimoski and L.Kocarev, " Chaos and cryptography: block encryption ciphers based on chaotic map" IEEE Trans. Circuits Syst. I, Vol.48,pp. 163-169, 2001.

[7] Philip P.Dang and Paul M.Chau, "Image Encryption for Secure Internet Multimedia Applications", Technical Report, Department of Electrical and Electronic Engineering, University of California, San Diago.

APPENDICES

APPENDIX A

H-CES CODING (16 BITS, 4 ITERATIONS)

```
/*************************************************************************
CHAOS BASED BLOCK ENCRYPTOR 16 BITS, 4 ITERATIONS.
FARHANA BT SARIN 5728
*************************************************************************/

#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/*********************************************************
        Here the declaration of the functions starts
 *********************************************************/
void plcm_init(int);
float plcm(float ic,float p);
unsigned short int gen_log_map(unsigned short int yi);
unsigned short int getSBoxValue(int num);
unsigned short int getSBoxInvert(int num);
void init(void);
void start_encrypt(void);
void start_decrypt(void);

 /* ************************************************************
    Here the declaration of the Global Variables starts
 ************************************************************/
/*************************************************************
        This is the 128bit key
*************************************************************/
static unsigned short int key[16]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

/*************************************************************
        The 16 initial conditions extracted from the 128 bit key
*************************************************************/
static float IC[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

/*************************************************************
        The extracted 8-bit values from 16-PLCMs
*************************************************************/
//the following variable stores 64 8-bit extracted values form the plcm, used for masking ( or key).
static unsigned short int
out[64]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

/*************************************************************
        The 16 control parameters for a bank of 16-PLCMs
*************************************************************/
static float p_control[16]={0.15,0.17,0.188,0.199,0.222,0.444,0.143,0.23,0.235,0.444,0.48,0.222,0.123,0.456,0.33,0.35};

/*************************************************************
        The 128-bit plaintexts and ciphertexts
*************************************************************/
unsigned short int plaintext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int ciphertext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int dec_ciphertext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; // this array is for to store the decrypted ciphertext

void main(void)
{
  int k=0;

// Here request user to enter data
  printf("*************************************************************************");
  printf("\n");
  printf("\n");
  printf("              CHAOS BASED BLOCK ENCRYPTOR");
  printf("\n");
  printf("                16 BITS , 4 ITERATIONS");
  printf("\n");
  printf("\n");
  printf("*************************************************************************");
```

```
  printf("\n");
  printf("\n");
  printf("\n");
  printf("Please enter 16 set plaintext: ");
  scanf("%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d",
  &plaintext[0] ,&plaintext[1] ,&plaintext[2] ,&plaintext[3] ,&plaintext[4], &plaintext[5] ,&plaintext[6] ,&plaintext[7] ,&plaintext[8],
 &plaintext[9] ,&plaintext[10],&plaintext[11],&plaintext[12],&plaintext[13],&plaintext[14], &plaintext[15]);

/*Clear the screen */
//clrscr();
 /* Initialization function*/
 init();
 /* After initialization start encryption*/
 start_encrypt();
  /* Start decryption*/
 start_decrypt();
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->PLAINTEXT<---\n\n");
 for(k=0;k<16;k++)
 {
          printf("%d  ",plaintext[k]);
 }
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->CIPHERTEXT<---\n\n");
 for(k=0;k<16;k++)
 {
          printf("%d  ",ciphertext[k]);
 }

 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->DECRYPTED CIPHERTEXT<---\n");

  for(k=0;k<16;k++)
 {
          printf("%d  ",dec_ciphertext[k]);
 }
getch();
}

void init(void)
{
/* Maping the 8-bit key value to a real number in the interval (0,1)*/
int i=0;
          for (i=0;i<16;i++)
          {
                    IC[i]=key[i]/256;
          }
}


void start_encrypt(void)
{
/* This is the counter for the number of rounds of HCES */
int counter=0;
int j=0;
int k=0;
int index;
int m;

unsigned short int temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int roundkey[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

          /* The following loop assigns plaintext to temp*/
          for (k=0;k<16;k++)
          {
                    temp[k]=plaintext[k];
```

```
	}
	m=4;	/* Extract 4*128 bit values*/
	plcm_init(m); /* Function call inorder to extract 4*128 bit  key stream*/
	index=0;

	for (counter=0;counter<4;counter++)
	{
		for (k=0;k<16;k++)
		{
			roundkey[k]=out[index];
			index=index+1;
		}
		for(j=0;j<16;j++)
		{
			if (j==0)
			{
				temp[j]=getSBoxValue(temp[j]);
			}
			else
			{
				temp[j]=getSBoxValue(temp[j])^gen_log_map(roundkey[j]^temp[j-1]);
			}
		}
			temp[0]=temp[0]^gen_log_map(temp[15]^roundkey[0]);
	}
	 /* The following loop assigns the final round output to the ciphertext array*/
	 for (k=0;k<16;k++)
	 {
		ciphertext[k]=temp[k];
	 }
}

void start_decrypt(void)
{
	/* This is the counter for the number of rounds of HCES */
	int counter=0;
	int j=0;
	int k=0;

	int index;
	unsigned short int temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	unsigned short int R_temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	unsigned short int roundkey[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

	/* The following loop assigns plaintext to temp*/
	for (k=0;k<16;k++)
	{
		temp[k]=ciphertext[k];
	}

	index=63;/* for decryption, index will start from 63, while for encryption it will start from 0*/

	for (counter=0;counter<4;counter++)
	{
		for (k=15;k>=0;k--)
		{
			roundkey[k]=out[index];
			index=index-1;
		}

		for(j=0;j<16;j++)
		{
			if (j==0)
			{
				R_temp[j]=(temp[j])^gen_log_map(roundkey[j]^temp[15]); // produces sx0
			}
			else
			{
				if (j==1)
				{
					R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^R_temp[0]));
					//produces x1
				}
				else
				{
					R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^temp[j-1]));
					//produces xj, except x0 & x1
```

```
                        }
                    }
                }
                    R_temp[0]=getSBoxInvert(R_temp[0]); //produces x0

                /*Re initialize the temp array, it serves as ciphertext for the next round of decryption*/
                for (k=0;k<16;k++)
                {
                    temp[k]=R_temp[k];
                }
        }

        /* The following loop assigns the final round output to the dec_ciphertext array*/
        for (k=0;k<16;k++)
        {
            dec_ciphertext[k]=temp[k];
        }

}

void plcm_init(int m)
{
    int i=0;
    unsigned short int temp=0;
        int k=0;
        int count=0;
        for (k=0; k<m; k++)
        {
                for (i=0; i<16; i++)
                {
                    IC[i]=plcm(IC[i],p_control[i]);
                    temp=IC[i]*100000;
                    temp=temp % 256;
                    out[count]=temp; /* This is 8-bit Extracted Value */
                    count=count+1;
                }
        }
}

float plcm(float ic, float p)
{
  int flag=0;
  float temp=0;
  while(flag==0)
    {
            if (0<=ic <=p)
        {
            temp=ic/p;
            flag=1;
        }

        else if( p<=ic<=0.5)
        {
            temp=(ic-p)/(0.5-p);
            flag=1;
        }

        else
        {
            ic=1-ic;
        }
    }
  return temp;
}

unsigned short int gen_log_map(unsigned short int yi)
{
/*The following function implements the Generalized Logistic Map */
unsigned short int result=0;
float t_result=0;
char c;

 t_result= floor ((yi * ( 256- yi) /64));
 if(t_result < 256)
 {
   result=t_result;
 }
```

```
 else
 {
   result=255;
 }
 return result;
}

unsigned short int getSBoxValue(int num)
{
   unsigned short int sbox[256] =  {
   //0     1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
   0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
   0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
   0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
   0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
   0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
   0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
   0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
   0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
   0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
   0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
   0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
   0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
   0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
   0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
   0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
   0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 }; //F
   return sbox[num];
}

unsigned short int getSBoxInvert(int num)
{
unsigned short int rsbox[256] =
{ 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb
, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb
, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e
, 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25
, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92
, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84
, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06
, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b
, 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73
, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e
, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b
, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4
, 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f
, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef
, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61
, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d };
return rsbox[num];
}
```

APPENDIX B

H-CES CODING (16 BITS, 5 ITERATIONS)

```
/*************************************************************************
CHAOS BASED BLOCK ENCRYPTOR 16 BITS, 5 ITERATIONS.
FARHANA BT SARIN 5728
*************************************************************************/
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/*********************************************************
        Here the declaration of the functions starts
 *********************************************************/
void plcm_init(int);
float plcm(float ic,float p);
unsigned short int gen_log_map(unsigned short int yi);
unsigned short int getSBoxValue(int num);
unsigned short int getSBoxInvert(int num);
void init(void);
void start_encrypt(void);
void start_decrypt(void);

/* ************************************************************
    Here the declaration of the Global Variables starts
 ************************************************************/
/************************************************************
        This is the 128bit key
************************************************************/
static unsigned short int key[16]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

/************************************************************
        The 16 initial conditions extracted from the 128 bit key
************************************************************/
static float IC[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

/************************************************************
        The extracted 8-bit values from 16-PLCMs
************************************************************/
//the following variable stores 80 8-bit extracted values form the plcm, used for masking ( or key).
static unsigned short int out[80];

/************************************************************
        The 16 control parameters for a bank of 16-PLCMs
************************************************************/
static float p_control[16]={0.15,0.17,0.188,0.199,0.222,0.444,0.143,0.23,0.235,0.444,0.48,0.222,0.123,0.456,0.33,0.35};

/************************************************************
        The 128-bit plaintexts and ciphertexts
************************************************************/
unsigned short int plaintext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int ciphertext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int dec_ciphertext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; // this array is for to store the decrypted ciphertext


void main(void)
{
  int k=0;
// Here request user to enter data
  printf("*************************************************************************");
  printf("\n");
  printf("\n");
  printf("              CHAOS BASED BLOCK ENCRYPTOR");
  printf("\n");
  printf("                16 BITS , 5 ITERATIONS");
  printf("\n");
  printf("\n");
  printf("*************************************************************************");
  printf("\n");
  printf("\n");
```

```
 printf("\n");
 printf("Please enter 16 set plaintext: ");
 scanf("%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d",
 &plaintext[0] ,&plaintext[1] ,&plaintext[2] ,&plaintext[3] ,&plaintext[4], &plaintext[5] ,&plaintext[6] ,&plaintext[7] ,&plaintext[8],
 &plaintext[9], &plaintext[10],&plaintext[11],&plaintext[12],&plaintext[13],&plaintext[14], &plaintext[15]);

//*Clear the screen */
//clrscr();
/* Initialization function*/
 init();
 /* After initialization start encryption*/
 start_encrypt();
/* Start decryption*/
 start_decrypt();

 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->PLAINTEXT<---\n\n");
 for(k=0;k<16;k++)
 {
          printf("%d  ",plaintext[k]);
 }
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->CIPHERTEXT<---\n\n");
 for(k=0;k<16;k++)
 {
          printf("%d  ",ciphertext[k]);
 }
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->DECRYPTED CIPHERTEXT<---\n");
  for(k=0;k<16;k++)
 {
          printf("%d  ",dec_ciphertext[k]);
 }
getch();
}

void init(void)
{
/* Maping the 8-bit key value to a real number in the interval (0,1)*/
int i=0;
          for (i=0;i<16;i++)
          {
                    IC[i]=key[i]/256;
          }
}


void start_encrypt(void)
{
/* This is the counter for the number of rounds of HCES */
int counter=0;
int j=0;
int k=0;
int index;
int m;
unsigned short int temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int roundkey[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

          /* The following loop assigns plaintext to temp*/
          for (k=0;k<16;k++)
          {
                    temp[k]=plaintext[k];
          }

          m=5;      /* Extract 5*128 bit values*/
          plcm_init(m); /* Function call inorder to extract 5*128 bit  key stream*/
```

```
	index=0;

	for (counter=0;counter<5;counter++)
	{
		for (k=0;k<16;k++)
		{
			roundkey[k]=out[index];
			index=index+1;
		}
		for(j=0;j<16;j++)
		{
			if (j==0)
			{
				temp[j]=getSBoxValue(temp[j]);
			}
			else
			{
				temp[j]=getSBoxValue(temp[j])^gen_log_map(roundkey[j]^temp[j-1]);
			}
		}
			temp[0]=temp[0]^gen_log_map(temp[15]^roundkey[0]);
	}
	/* The following loop assigns the final round output to the ciphertext array*/
	for (k=0;k<16;k++)
	{
		ciphertext[k]=temp[k];
	}
}

void start_decrypt(void)
{
	/* This is the counter for the number of rounds of HCES */
	int counter=0;
	int j=0;
	int k=0;
	int index;
	unsigned short int temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	unsigned short int R_temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	unsigned short int roundkey[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

	/* The following loop assigns plaintext to temp*/
	for (k=0;k<16;k++)
	{
		temp[k]=ciphertext[k];
	}

	index=79;/* for decryption, index will start from 79, while for encryption it will start from 0*/
	for (counter=0;counter<5;counter++)
	{
		for (k=15;k>=0;k--)
		{
			roundkey[k]=out[index];
			index=index-1;
		}
		for(j=0;j<16;j++)
		{
			if (j==0)
			{
				R_temp[j]=(temp[j])^gen_log_map(roundkey[j]^temp[15]); // produces sx0
			}
			else
			{
				if (j==1)
				{
					R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^R_temp[0]));
					// produces x1
				}
				else
				{
					R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^temp[j-1]));
					//produces xj, except x0 & x1
				}
			}
		}
			R_temp[0]=getSBoxInvert(R_temp[0]); //produces x0
```

```
/*Re initialize the temp array, it serves as ciphertext for the next round of decryption*/
                        for (k=0;k<16;k++)
                        {
                                temp[k]=R_temp[k];
                        }
                }

                /* The following loop assigns the final round output to the dec_ciphertext array*/
                for (k=0;k<16;k++)
                {
                        dec_ciphertext[k]=temp[k];
                }
}

void plcm_init(int m)
{
    int i=0;
    unsigned short int temp=0;
    int k=0;
    int count=0;
          for (k=0; k<m; k++)
          {
                for (i=0; i<16; i++)
                        {
                                IC[i]=plcm(IC[i],p_control[i]);
                                temp=IC[i]*100000;
                                temp=temp % 256;
                                out[count]=temp;      /* This is 8-bit Extracted Value */
                                count=count+1;
                        }
          }
}

float plcm(float ic, float p)
{
int flag=0;
float temp=0;
while(flag==0)
{
        if (0<=ic <=p)
        {
           temp=ic/p;
           flag=1;
        }

        else if( p<=ic<=0.5)
        {
                temp=(ic-p)/(0.5-p);
                flag=1;
        }

        else
        {
                ic=1-ic;
        }
   }
  return temp;
}

unsigned short int gen_log_map(unsigned short int yi)
{
/*The following function implements the Generalized Logistic Map */
unsigned short int result=0;
float t_result=0;
char c;

 t_result= floor ((yi * ( 256- yi) /64));
        if (t_result < 256)
                {
                  result=t_result;
                }
        else
                {
                  result=255;
                }
         return result;
}
```

APPENDIX C

H-CES CODING (16 BITS, 6 ITERATIONS)

```
/*************************************************************************
CHAOS BASED BLOCK ENCRYPTOR 16 BITS, 6 ITERATIONS.
FARHANA BT SARIN 5728
*************************************************************************/
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
/* ******************************************************
        Here the declaration of the functions starts
 ******************************************************/
void plcm_init(int);
float plcm(float ic,float p);
unsigned short int gen_log_map(unsigned short int yi);
unsigned short int getSBoxValue(int num);
unsigned short int getSBoxInvert(int num);
void init(void);
void start_encrypt(void);
void start_decrypt(void);

 /*************************************************************
    Here the declaration of the Global Variables starts
 *************************************************************/
/*************************************************************
        This is the 128bit key
*************************************************************/
static unsigned short int key[16]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

/*************************************************************
        The 16 initial conditions extracted from the 128 bit key
*************************************************************/
static float IC[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

/*************************************************************
        The extracted 8-bit values from 16-PLCMs
*************************************************************/
//the following variable stores 96 8-bit extracted values form the plcm, used for masking ( or key).
static unsigned short int out[96];

/*************************************************************
        The 16 control parameters for a bank of 16-PLCMs
*************************************************************/
static float p_control[16]={0.15,0.17,0.188,0.199,0.222,0.444,0.143,0.23,0.235,0.444,0.48,0.222,0.123,0.456,0.33,0.35};

/*************************************************************
        The 128-bit plaintexts and ciphertexts
*************************************************************/
unsigned short int plaintext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int ciphertext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int dec_ciphertext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; // this array is for to store the decrypted ciphertext

void main(void)
{
  int k=0;
// Here request user to enter data

  printf("*************************************************************************");
  printf("\n");
  printf("\n");
  printf("              CHAOS BASED BLOCK ENCRYPTOR");
  printf("\n");
  printf("                16 BITS , 6 ITERATIONS");
  printf("\n");
  printf("\n");
  printf("*************************************************************************");
  printf("\n");
  printf("\n");
  printf("\n");
```

```
printf("Please enter 16 set plaintext: ");
 scanf("%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d",
   &plaintext[0] ,&plaintext[1] ,&plaintext[2] ,&plaintext[3] ,&plaintext[4],
   &plaintext[5] ,&plaintext[6] ,&plaintext[7] ,&plaintext[8] ,&plaintext[9],
   &plaintext[10],&plaintext[11],&plaintext[12],&plaintext[13],&plaintext[14],
   &plaintext[15]);

   /*Clear the screen */
//clrscr();
 /* Initialization function*/
 init();
 /* After initialization start encryption*/
 start_encrypt();
  /* Start decryption*/
 start_decrypt();

 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->PLAINTEXT<---\n\n");
 for(k=0;k<16;k++)
 {
            printf("%d  ",plaintext[k]);
 }
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->CIPHERTEXT<---\n\n");
 for(k=0;k<16;k++)
 {
            printf("%d  ",ciphertext[k]);
 }

 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->DECRYPTED CIPHERTEXT<---\n");
  for(k=0;k<16;k++)
 {
            printf("%d  ",dec_ciphertext[k]);
 }
getch();
}

void init(void)
{
/* Maping the 8-bit key value to a real number in the interval (0,1)*/
int i=0;
            for (i=0;i<16;i++)
            {
                        IC[i]=key[i]/256;
            }
}

void start_encrypt(void)
{
/* This is the counter for the number of rounds of HCES */
int counter=0;
int j=0;
int k=0;
int index;
int m;
unsigned short int temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int roundkey[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
            /* The following loop assigns plaintext to temp*/
            for (k=0;k<16;k++)
            {
                        temp[k]=plaintext[k];
            }
            m=6;        /* Extract 6*128 bit values*/
            plcm_init(m); /* Function call inorder to extract 6*128 bit  key stream*/
            index=0;
```

```
	for (counter=0;counter<6;counter++)
	{
		for (k=0;k<16;k++)
		{
			roundkey[k]=out[index];
			index=index+1;
		}
		for(j=0;j<16;j++)
		{
			if (j==0)
			{
				temp[j]=getSBoxValue(temp[j]);
			}
			else
			{
				temp[j]=getSBoxValue(temp[j])^gen_log_map(roundkey[j]^temp[j-1]);
			}
		}
			temp[0]=temp[0]^gen_log_map(temp[15]^roundkey[0]);
	}
	 /* The following loop assigns the final round output to the ciphertext array*/
	 for (k=0;k<16;k++)
	 {
		ciphertext[k]=temp[k];
	 }

}

void start_decrypt(void)
{
	/* This is the counter for the number of rounds of HCES */
	int counter=0;
	int j=0;
	int k=0;
	int index;
	unsigned short int temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	unsigned short int R_temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	unsigned short int roundkey[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

		/* The following loop assigns plaintext to temp*/
	for (k=0;k<16;k++)
	{
		temp[k]=ciphertext[k];
	}

	index=95;/* for decryption, index will start from 95, while for encryption it will start from 0*/
	for (counter=0;counter<6;counter++)
	{
		for (k=15;k>=0;k--)
		{
			roundkey[k]=out[index];
			index=index-1;
		}
		for(j=0;j<16;j++)
		{
			if (j==0)
			{
				R_temp[j]=(temp[j])^gen_log_map(roundkey[j]^temp[15]); // produces sx0
			}
			else
			{
				if (j==1)
				{
					R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^R_temp[0]));
					// produces x1
				}
				else
				{
					R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^temp[j-1]));
					//produces xj, except x0 & x1
				}
			}
		}
			R_temp[0]=getSBoxInvert(R_temp[0]); //produces x0

		/*Re initialize the temp array, it serves as ciphertext for the next round of decryption*/
		for (k=0;k<16;k++)
```

```
                {
                        temp[k]=R_temp[k];
                }
        }

        /* The following loop assigns the final round output to the dec_ciphertext array*/
        for (k=0;k<16;k++)
        {
                dec_ciphertext[k]=temp[k];
        }
}

void plcm_init(int m)
{

    int i=0;
    unsigned short int temp=0;
        int k=0;
        int count=0;
        for (k=0; k<m; k++)
        {
                        for (i=0; i<16; i++)
                        {
                                IC[i]=plcm(IC[i],p_control[i]);
                                temp=IC[i]*100000;
                                temp=temp % 256;
                                out[count]=temp; /* This is 8-bit Extracted Value */
                                count=count+1;
                        }
        }
}

float plcm(float ic, float p)
{
int flag=0;
float temp=0;
while(flag==0)
{
        if (0<=ic <=p)
        {
           temp=ic/p;
           flag=1;
        }
        else if( p<=ic<=0.5)
        {

                temp=(ic-p)/(0.5-p);
                flag=1;
        }
        else
        {
                ic=1-ic;
        }
}
return temp;
}

unsigned short int gen_log_map(unsigned short int yi)
{
/*The following function implements the Generalized Logistic Map */
unsigned short int result=0;
float t_result=0;
char c;

 t_result= floor ((yi * ( 256- yi) /64));

 if(t_result < 256)
 {
   result=t_result;
 }
 else
 {
   result=255;
 }
 return result;

}
```

```
unsigned short int getSBoxValue(int num)
{
  unsigned short int sbox[256] =  {
  //0     1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 }; //F
  return sbox[num];
}


unsigned short int getSBoxInvert(int num)
{
unsigned short int rsbox[256] =
{ 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb
, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb
, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e
, 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25
, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92
, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84
, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06
, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b
, 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73
, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e
, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b
, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4
, 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f
, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef
, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61
, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d };
return rsbox[num];
}
```

APPENDIX D

H-CES CODING (16 BITS, 7 ITERATIONS)

```
/*************************************************************************
CHAOS BASED BLOCK ENCRYPTOR 16 BITS, 7 ITERATIONS.
FARHANA BT SARIN 5728
*************************************************************************/
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
/*******************************************************
        Here the declaration of the functions starts
 *******************************************************/
void plcm_init(int);
float plcm(float ic,float p);
unsigned short int gen_log_map(unsigned short int yi);
unsigned short int getSBoxValue(int num);
unsigned short int getSBoxInvert(int num);
void init(void);
void start_encrypt(void);
void start_decrypt(void);

/*************************************************************
    Here the declaration of the Global Variables starts
 *************************************************************/
/*************************************************************
        This is the 128bit key
*************************************************************/
static unsigned short int key[16]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
/*************************************************************
        The 16 initial conditions extracted from the 128 bit key
*************************************************************/
static float IC[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
/*************************************************************
        The extracted 8-bit values from 16-PLCMs
*************************************************************/
//the following variable stores 112 8-bit extracted values form the plcm, used for masking ( or key).
static unsigned short int out[112];
/*************************************************************
        The 16 control parameters for a bank of 16-PLCMs
*************************************************************/
static float p_control[16]={0.15,0.17,0.188,0.199,0.222,0.444,0.143,0.23,0.235,0.444,0.48,0.222,0.123,0.456,0.33,0.35};
/*************************************************************
        The 128-bit plaintexts and ciphertexts
*************************************************************/
unsigned short int plaintext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int ciphertext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int dec_ciphertext[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; // this array is for to store the decrypted ciphertext

void main(void)
{
  int k=0;
// Here request user to enter data
  printf("*************************************************************************");
  printf("\n");
  printf("\n");
  printf("                CHAOS BASED BLOCK ENCRYPTOR");
  printf("\n");
  printf("                  16 BITS , 7 ITERATIONS");
  printf("\n");
  printf("\n");
  printf("*************************************************************************");
  printf("\n");
  printf("\n");
  printf("\n");
  printf("Please enter 16 set plaintext: ");
  scanf("%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d",
  &plaintext[0] ,&plaintext[1] ,&plaintext[2] ,&plaintext[3] ,&plaintext[4],
  &plaintext[5] ,&plaintext[6] ,&plaintext[7] ,&plaintext[8] ,&plaintext[9],
  &plaintext[10],&plaintext[11],&plaintext[12],&plaintext[13],&plaintext[14], &plaintext[15]);
```

```
  /*Clear the screen */
//clrscr();
 /* Initialization function*/
 init();
 /* After initialization start encryption*/
 start_encrypt();
  /* Start decryption*/
 start_decrypt();
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->PLAINTEXT<---\n\n");
 for(k=0;k<16;k++)
 {
          printf("%d  ",plaintext[k]);
 }
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->CIPHERTEXT<---\n\n");
 for(k=0;k<16;k++)
 {
          printf("%d  ",ciphertext[k]);
 }

 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->DECRYPTED CIPHERTEXT<---\n");

  for(k=0;k<16;k++)
 {
          printf("%d  ",dec_ciphertext[k]);
 }
getch();
}

void init(void)
{
/* Maping the 8-bit key value to a real number in the interval (0,1)*/
int i=0;
          for (i=0;i<16;i++)
          {
                    IC[i]=key[i]/256;
          }
}


void start_encrypt(void)
{
/* This is the counter for the number of rounds of HCES */
int counter=0;
int j=0;
int k=0;
int index;
int m;


unsigned short int temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int roundkey[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

          /* The following loop assigns plaintext to temp*/
          for (k=0;k<16;k++)
          {
                    temp[k]=plaintext[k];
          }

          m=7;      /* Extract 7*128 bit values*/
          plcm_init(m); /* Function call inorder to extract 7*128 bit  key stream*/
          index=0;

          for (counter=0;counter<7;counter++)
```

```
{
        for (k=0;k<16;k++)
        {
                roundkey[k]=out[index];
                index=index+1;
        }
        for(j=0;j<16;j++)
        {
                if (j==0)
                {
                        temp[j]=getSBoxValue(temp[j]);
                }
                else
                {
                        temp[j]=getSBoxValue(temp[j])^gen_log_map(roundkey[j]^temp[j-1]);
                }
        }
                temp[0]=temp[0]^gen_log_map(temp[15]^roundkey[0]);
}
 /* The following loop assigns the final round output to the ciphertext array*/
 for (k=0;k<16;k++)
 {
        ciphertext[k]=temp[k];
 }

}

void start_decrypt(void)
{
        /* This is the counter for the number of rounds of HCES */
        int counter=0;
        int j=0;
        int k=0;

        int index;
        unsigned short int temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
        unsigned short int R_temp[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
        unsigned short int roundkey[16]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

                /* The following loop assigns plaintext to temp*/
        for (k=0;k<16;k++)
        {
                temp[k]=ciphertext[k];
        }

        index=111;/* for decryption, index will start from 111, while for encryption it will start from 0*/
        for (counter=0;counter<7;counter++)
        {
                for (k=15;k>=0;k--)
                {
                        roundkey[k]=out[index];
                        index=index-1;
                }

                for(j=0;j<16;j++)
                {
                        if (j==0)
                        {
                                R_temp[j]=(temp[j])^gen_log_map(roundkey[j]^temp[15]); // produces sx0

                        }
                        else
                        {
                                if (j==1)
                                {
                                        R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^R_temp[0]));
                                        // produces x1
                                }
                                else
                                {
                                        R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^temp[j-1]));
                                        //produces xj, except x0 & x1
                                }
                        }
                }
                        R_temp[0]=getSBoxInvert(R_temp[0]); //produces x0
```

```
                /*Re initialize the temp array, it serves as ciphertext for the next round of decryption*/
                for (k=0;k<16;k++)
                {
                        temp[k]=R_temp[k];
                }
        }

        /* The following loop assigns the final round output to the dec_ciphertext array*/
        for (k=0;k<16;k++)
        {
                dec_ciphertext[k]=temp[k];
        }

}
void plcm_init(int m)
{

    int i=0;
    unsigned short int temp=0;
        int k=0;
        int count=0;
        for (k=0; k<m; k++)
        {

                        for (i=0; i<16; i++)
                        {
                                IC[i]=plcm(IC[i],p_control[i]);
                                temp=IC[i]*100000;
                                temp=temp % 256;
                                out[count]=temp; /* This is 8-bit Extracted Value */
                                count=count+1;
                        }
        }
}

float plcm(float ic, float p)
{
int flag=0;
float temp=0;
while(flag==0)
{
        if (0<=ic <=p)
        {
           temp=ic/p;
           flag=1;
        }
        else if( p<=ic<=0.5)
        {

                temp=(ic-p)/(0.5-p);
                flag=1;

        }
        else
        {
                ic=1-ic;
        }
}
return temp;

}

unsigned short int gen_log_map(unsigned short int yi)
{
/*The following function implements the Generalized Logistic Map */
unsigned short int result=0;
float t_result=0;
char c;

 t_result= floor ((yi * ( 256- yi) /64));

 if(t_result < 256)
 {
   result=t_result;
 }
 else
```

```
 {
   result=255;
 }
 return result;

}
unsigned short int getSBoxValue(int num)
{
   unsigned short int sbox[256] =  {
   //0     1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
   0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
   0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
   0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
   0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
   0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
   0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
   0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
   0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
   0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
   0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
   0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
   0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
   0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
   0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
   0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
   0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 }; //F
   return sbox[num];
}


unsigned short int getSBoxInvert(int num)
{
unsigned short int rsbox[256] =
{ 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb
, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb
, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e
, 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25
, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92
, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84
, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06
, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b
, 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73
, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e
, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b
, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4
, 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f
, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef
, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61
, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d };

return rsbox[num];
}
```

APPENDIX E

H-CES CODING (20 BITS, 4 ITERATIONS)

```
/*************************************************************************
CHAOS BASED BLOCK ENCRYPTOR 20 BITS, 4 ITERATIONS.
FARHANA BT SARIN 5728
*************************************************************************/
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
/*********************************************************
        Here the declaration of the functions starts
 *********************************************************/
void plcm_init(int);
float plcm(float ic,float p);
unsigned short int gen_log_map(unsigned short int yi);
unsigned short int getSBoxValue(int num);
unsigned short int getSBoxInvert(int num);
void init(void);
void start_encrypt(void);
void start_decrypt(void);

 /*************************************************************
    Here the declaration of the Global Variables starts
 *************************************************************/
/*************************************************************
        This is the 160bit key
*************************************************************/
static unsigned short int key[20]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19};

/*************************************************************
        The 16 initial conditions extracted from the 128 bit key
*************************************************************/
static float IC[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

/*************************************************************
        The extracted 8-bit values from 16-PLCMs
*************************************************************/
//the following variable stores 80 8-bit extracted values form the plcm, used for masking ( or key).
static unsigned short int out[80];

/*************************************************************
        The 16 control parameters for a bank of 16-PLCMs
*************************************************************/
static float p_control[20]={
0.15,0.17,0.188,0.199,0.222,0.444,0.143,0.23,0.235,0.444,0.48,0.222,0.123,0.456,0.33,0.35,0.15,0.17,0.188,0.199};

/*************************************************************
        The 128-bit plaintexts and ciphertexts
*************************************************************/
unsigned short int plaintext[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int ciphertext[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
unsigned short int dec_ciphertext[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}; // this array is for to store the decrypted ciphertext

void main(void)
{
  int k=0;
// Here request user to enter data
  printf("*************************************************************************");
  printf("\n");
  printf("\n");
  printf("              CHAOS BASED BLOCK ENCRYPTOR");
  printf("\n");
  printf("                20 BITS , 4 ITERATIONS");
  printf("\n");
  printf("\n");
  printf("*************************************************************************");
  printf("\n");
  printf("\n");
  printf("\n");
```

```
  printf("Please enter 20 set plaintext: ");
  scanf("%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d",
  &plaintext[0] ,&plaintext[1] ,&plaintext[2] ,&plaintext[3] ,&plaintext[4],
  &plaintext[5] ,&plaintext[6] ,&plaintext[7] ,&plaintext[8] ,&plaintext[9],
  &plaintext[10],&plaintext[11],&plaintext[12],&plaintext[13],&plaintext[14],
  &plaintext[15],&plaintext[16],&plaintext[17],&plaintext[18],&plaintext[19]);

  /*Clear the screen */
//clrscr();
 /* Initialization function*/
 init();
 /* After initialization start encryption*/
 start_encrypt();
  /* Start decryption*/
 start_decrypt();
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    -->PLAINTEXT<---\n\n");
 for(k=0;k<20;k++)
 {
        printf("%d  ",plaintext[k]);
 }
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    -->CIPHERTEXT<---\n\n");
 for(k=0;k<20;k++)
 {
        printf("%d  ",ciphertext[k]);
 }
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    -->DECRYPTED CIPHERTEXT<---\n");
  for(k=0;k<20;k++)
 {
        printf("%d  ",dec_ciphertext[k]);
 }
 getch();
 }

 void init(void)
 {
 /* Maping the 8-bit key value to a real number in the interval (0,1)*/
 int i=0;
        for (i=0;i<20;i++)
        {
                IC[i]=key[i]/256;
        }
 }

 void start_encrypt(void)
 {
 /* This is the counter for the number of rounds of HCES */
 int counter=0;
 int j=0;
 int k=0;
 int index;
 int m;
 unsigned short int temp[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
 unsigned short int roundkey[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
        /* The following loop assigns plaintext to temp*/
        for (k=0;k<20;k++)
        {
                temp[k]=plaintext[k];
        }
        m=4;    /* Extract 4*160 bit values*/
        plcm_init(m); /* Function call inorder to extract 4*160 bit  key stream*/
        index=0;
        for (counter=0;counter<4;counter++)
        {
```

```
		for (k=0;k<20;k++)
		{
			roundkey[k]=out[index];
			index=index+1;
		}
		for(j=0;j<20;j++)
		{
			if (j==0)
			{
				temp[j]=getSBoxValue(temp[j]);
			}
			else
			{
				temp[j]=getSBoxValue(temp[j])^gen_log_map(roundkey[j]^temp[j-1]);
			}
		}
			temp[0]=temp[0]^gen_log_map(temp[19]^roundkey[0]);
	}
	/* The following loop assigns the final round output to the ciphertext array*/
	for (k=0;k<20;k++)
	{
		ciphertext[k]=temp[k];
	}
}

void start_decrypt(void)
{
	/* This is the counter for the number of rounds of HCES */
	int counter=0;
	int j=0;
	int k=0;
	int index;
	unsigned short int temp[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	unsigned short int R_temp[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	unsigned short int roundkey[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
	/* The following loop assigns plaintext to temp*/
	for (k=0;k<20;k++)
	{
		temp[k]=ciphertext[k];
	}

	index=79;/* for decryption, index will start from 79, while for encryption it will start from 0*/
	for (counter=0;counter<4;counter++)
	{
		for (k=19;k>=0;k--)
		{
			roundkey[k]=out[index];
			index=index-1;
		}
		for(j=0;j<20;j++)
		{
			if (j==0)
			{
				R_temp[j]=(temp[j])^gen_log_map(roundkey[j]^temp[19]); // produces sx0
			}
			else
			{
				if (j==1)
				{
					R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^R_temp[0]));
					// produces x1
				}
				else
				{
					R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^temp[j-1]));
					//produces xj, except x0 & x1
				}
			}
		}
			R_temp[0]=getSBoxInvert(R_temp[0]); //produces x0

		/*Re initialize the temp array, it serves as ciphertext for the next round of decryption*/
		for (k=0;k<20;k++)
		{
			temp[k]=R_temp[k];
		}
```

```
        }

        /* The following loop assigns the final round output to the dec_ciphertext array*/
        for (k=0;k<20;k++)
        {
                dec_ciphertext[k]=temp[k];
        }
}

void plcm_init(int m)
{

    int i=0;
    unsigned short int temp=0;
    int k=0;
    int count=0;
    for (k=0; k<m; k++)
            {
                    for (i=0; i<20; i++)
                            {
                                    IC[i]=plcm(IC[i],p_control[i]);
                                    temp=IC[i]*100000;
                                    temp=temp % 256;
                                    out[count]=temp; /* This is 8-bit Extracted Value */
                                    count=count+1;
                            }
            }
}

float plcm(float ic, float p)
{
int flag=0;
float temp=0;
while(flag==0)
  {
        if (0<=ic <=p)
        {
           temp=ic/p;
           flag=1;
        }
        else if( p<=ic<=0.5)
        {
                temp=(ic-p)/(0.5-p);
                flag=1;
        }
        else
        {
                ic=1-ic;
        }
   }
  return temp;

}

unsigned short int gen_log_map(unsigned short int yi)
{
/*The following function implements the Generalized Logistic Map */
unsigned short int result=0;
float t_result=0;
char c;
 t_result= floor ((yi * ( 256- yi) /64));
 if (t_result < 256)
  {
    result=t_result;
  }
  else
  {
    result=255;
  }
  return result;
}
```

```
unsigned short int getSBoxValue(int num)
{
  unsigned short int sbox[256] =  {
  //0     1    2     3     4    5     6   7     8     9     A     B     C     D     E     F
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 }; //F
  return sbox[num];
}


unsigned short int getSBoxInvert(int num)
{
unsigned short int rsbox[256] =
{ 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb
, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb
, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e
, 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25
, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92
, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84
, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06
, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b
, 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73
, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e
, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b
, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4
, 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f
, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef
, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61
, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d };

return rsbox[num];
}
```

APPENDIX F

H-CES CODING (24 BITS, 4 ITERATIONS)

```
/*************************************************************************
CHAOS BASED BLOCK ENCRYPTOR 24 BITS, 4 ITERATIONS.
FARHANA BT SARIN 5728
*************************************************************************/
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
/* ******************************************************
        Here the declaration of the functions starts
 ******************************************************/
void plcm_init(int);
float plcm(float ic,float p);
unsigned short int gen_log_map(unsigned short int yi);
unsigned short int getSBoxValue(int num);
unsigned short int getSBoxInvert(int num);
void init(void);
void start_encrypt(void);
void start_decrypt(void);
 /*************************************************************
    Here the declaration of the Global Variables starts
 *************************************************************/
/*************************************************************
        This is the 192bit key
*************************************************************/
static unsigned short int key[24]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23};

/*************************************************************
        The 16 initial conditions extracted from the 128 bit key
*************************************************************/
static float IC[24];

/*************************************************************
        The extracted 8-bit values from 16-PLCMs
*************************************************************/
//the following variable stores 96 8-bit extracted values form the plcm, used for masking ( or key).
static unsigned short int out[96];

/*************************************************************
        The 16 control parameters for a bank of 16-PLCMs
*************************************************************/
static float p_control[24]={0.15,0.17,0.188,0.199,0.222,0.444,0.143,0.23,0.235,0.444,0.48,0.222,0.123,0.456,0.33,0.35,
                0.15,0.17,0.188,0.199,0.222,0.444,0.143,0.23};


/*************************************************************
        The 128-bit plaintexts and ciphertexts
*************************************************************/
unsigned short int plaintext[24];
unsigned short int ciphertext[24];
unsigned short int dec_ciphertext[24]; // this array is for to store the decrypted ciphertext

void main(void)
{
  int k=0;
// Here request user to enter data
  printf("*************************************************************************");
  printf("\n");
  printf("\n");
  printf("              CHAOS BASED BLOCK ENCRYPTOR");
  printf("\n");
  printf("                24 BITS, 4 ITERATIONS");
  printf("\n");
  printf("\n");
  printf("*************************************************************************");
  printf("\n");
  printf("\n");
  printf("\n");
```

```
  printf("Please enter 24 set plaintext: ");
  scanf("%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d",
  &plaintext[0] ,&plaintext[1] ,&plaintext[2] ,&plaintext[3] ,&plaintext[4],
  &plaintext[5] ,&plaintext[6] ,&plaintext[7] ,&plaintext[8] ,&plaintext[9],
  &plaintext[10],&plaintext[11],&plaintext[12],&plaintext[13],&plaintext[14],
  &plaintext[15],&plaintext[16],&plaintext[17],&plaintext[18],&plaintext[19],
  &plaintext[20],&plaintext[21],&plaintext[22],&plaintext[23]);

  /*Clear the screen */
//clrscr();
 /* Initialization function*/
 init();
 /* After initialization start encryption*/
 start_encrypt();
  /* Start decryption*/
 start_decrypt();
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->PLAINTEXT<---\n\n");
 for(k=0;k<24;k++)
 {
          printf("%d  ",plaintext[k]);
 }
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->CIPHERTEXT<---\n\n");
 for(k=0;k<24;k++)
 {
          printf("%d  ",ciphertext[k]);
 }
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("\n");
 printf("                    --->DECRYPTED CIPHERTEXT<---\n");
  for(k=0;k<24;k++)
 {
          printf("%d  ",dec_ciphertext[k]);
 }
getch();
}

void init(void)
{
/* Maping the 8-bit key value to a real number in the interval {0,1)*/
int i=0;
          for (i=0;i<24;i++)
          {
                    IC[i]=key[i]/256;
          }
}

void start_encrypt(void)
{
/* This is the counter for the number of rounds of HCES */
int counter=0;
int j=0;
int k=0;
int index;
int m;
unsigned short int temp[24];
unsigned short int roundkey[24];
          /* The following loop assigns plaintext to temp*/
          for (k=0;k<24;k++)
          {
                    temp[k]=plaintext[k];
          }

          m=4;      /* Extract 4*192 bit values*/
          plcm_init(m); /* Function call inorder to extract 4*192 bit  key stream*/
          index=0;
```

```
	for (counter=0;counter<4;counter++)
	{
		for (k=0;k<24;k++)
		{
			roundkey[k]=out[index];
			index=index+1;
		}
		for(j=0;j<24;j++)
		{
			if (j==0)
			 {
				temp[j]=getSBoxValue(temp[j]);
			 }
			 else
			 {
				temp[j]=getSBoxValue(temp[j])^gen_log_map(roundkey[j]^temp[j-1]);
			 }
		}
			temp[0]=temp[0]^gen_log_map(temp[23]^roundkey[0]);
	}
	 /* The following loop assigns the final round output to the ciphertext array*/
	 for (k=0;k<24;k++)
	 {
		 ciphertext[k]=temp[k];
	 }
}

void start_decrypt(void)
{
	/* This is the counter for the number of rounds of HCES */
	int counter=0;
	int j=0;
	int k=0;
	int index;
	unsigned short int temp[24];
	unsigned short int R_temp[24];
	unsigned short int roundkey[24];
	/* The following loop assigns plaintext to temp*/
	for (k=0;k<24;k++)
	{
		temp[k]=ciphertext[k];
	}
	index=95;/* for decryption, index will start from 95, while for encryption it will start from 0*/
	for (counter=0;counter<4;counter++)
	{
		for (k=23;k>=0;k--)
		{
			roundkey[k]=out[index];
			index=index-1;

		}
		for(j=0;j<24;j++)
		{
			if (j==0)
			 {
				R_temp[j]=(temp[j])^gen_log_map(roundkey[j]^temp[23]); // produces sx0
			 }
			else
			{
				if (j==1)
				{
					R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^R_temp[0]));
					// produces x1
				}
				else
				{
					R_temp[j]=getSBoxInvert(temp[j]^gen_log_map(roundkey[j]^temp[j-1]));
					 //produces xj, except x0 & x1
				}
			}
		}
			 R_temp[0]=getSBoxInvert(R_temp[0]); //produces x0

		/*Re initialize the temp array, it serves as ciphertext for the next round of decryption*/
		for (k=0;k<24;k++)
```

```
                {
                        temp[k]=R_temp[k];
                }
        }

        /* The following loop assigns the final round output to the dec_ciphertext array*/
        for (k=0;k<24;k++)
        {
                dec_ciphertext[k]=temp[k];
        }
}

void plcm_init(int m)
{
    int i=0;
    unsigned short int temp=0;
    int k=0;
    int count=0;
    for (k=0; k<m; k++)
        {
                for (i=0; i<24; i++)
                        {
                                IC[i]=plcm(IC[i],p_control[i]);
                                temp=IC[i]*100000;
                                temp=temp % 256;
                                out[count]=temp; /* This is 8-bit Extracted Value */
                                count=count+1;
                        }
        }
}
float plcm(float ic, float p)
{
int flag=0;
float temp=0;
while(flag==0)
{
        if (0<=ic <=p)
        {
           temp=ic/p;
           flag=1;
        }
        else if( p<=ic<=0.5)
        {
                temp=(ic-p)/(0.5-p);
                flag=1;
        }
        else
        {
                ic=1-ic;
        }
}
return temp;
}

unsigned short int gen_log_map(unsigned short int yi)
{
/*The following function implements the Generalized Logistic Map */
unsigned short int result=0;
float t_result=0;
char c;

 t_result= floor ((yi * ( 256- yi) /64));

 if(t_result < 256)
 {
   result=t_result;
 }
 else
 {
   result=255;
 }
 return result;
}
```

```
unsigned short int getSBoxValue(int num)
{
  unsigned short int sbox[256] =  {
  //0     1     2     3     4     5     6     7     8     9     A     B     C     D     E     F
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 }; //F
  return sbox[num];
}


unsigned short int getSBoxInvert(int num)
{
unsigned short int rsbox[256] =
{ 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb
, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb
, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e
, 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25
, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92
, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84
, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06
, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b
, 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73
, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e
, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b
, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4
, 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f
, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef
, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61
, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d };
return rsbox[num];
}
```