Chapter – III

Triangular Encoder (TE)

Chapter – III

Triangular Encoder

3 Introduction

In the chapter 2, the principle of PP encoder is described in detail. Its realization using microprocessor based system and comparison with RSA encoder are also made.

This is another encoder which deals string / block as input and produces the string / block as output as in the PP Encoder. This encoder follows the principle of block cipher technique. It is a symmetric encryption and can be applied for secured transmission through unprotected line and internet facility.

3.1 Principle of Triangular Encoder

This encoder will accept a string of bit length, n. It will generate the encoded string of bit length, n through encoding process. All the generating bits in the encoding as well as decoding process looks like a triangle and hence it is termed as triangular encoder. The generated bits in the triangle give the encoded string.

Considering the string or block length be n. (n-1) bit will be generated in the first phase, (n-2) bit in the second phase, (n-3) bit in the third phase, and so on, (n-(n-1)) bit i.e. 1 bit in the last phase. There will be (n-1) phase for n bit string in the encoding as well as decoding process. The first bit of the string and the first bits of each phase constitute the encoded string. The same process is followed to decode the encoded string. The encoding and decoding process is applicable to a string of any length. The principle is discussed with 8 bit string only. The result is taken for 8, 16, 32, 64, 128 and 256 bit string.

The encoding and decoding techniques are discussed in section 3.1.1 and 3.1.2 respectively.

3.1.1 Encoding Technique

To illustrate the encoding technique of the triangular encoder, a string consisting of 8 bits is considered, as shown in fig 3.1.

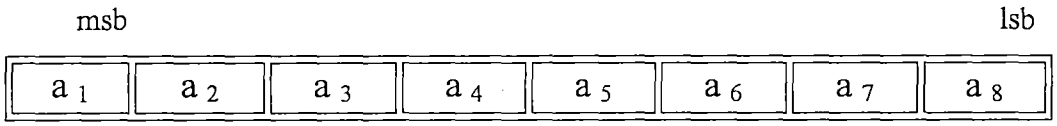

Fig 3.1 An 8 bit string for Triangular Encoding

a_1, a_2, a_3, a_4, a_5, a_6, a_7 and a_8 are the bits of the string, a_1 the most significant bit (msb) and a_8 the least significant bit (lsb) of the string.

The encoding operation starts from most significant bit and continues to least significant bit. The msb is XORed with the next to msb i.e. a_1 is XORed with a_2 and generates a_{12}; a_2 is XORed with a_3 and generates a_{23}; and so on, up to a_{78} . Thus it generates 7 bits (a_{12} a_{23} a_{34} a_{45} a_{56} a_{67} a_{78}) in the process as in figure 3.2. The same encoding operation will be applied on the 7 generated bits producing 6 bits (a_{13} a_{24} a_{35} a_{46} a_{57} a_{68}). The operation continues till it generates single bit at the output. The whole operation is illustrated in figure 3.2.

	Source String							
ENCODED String	**a** 1	**a** 2	**a** 3	**a** 4	**a** 5	**a** 6	**a** 7	**a** 8
	a 12	a 23	a 34	a 45	a 56	a 67	a 78	
	a 13	a 24	a 35	a 46	a 57	a 68		
	a 1234	a 2345	a 3456	a 4567	a 5678			
	a 15	a 26	a 37	a 48				
	a 1256	a 2367	a 3478					
	a 1357	a 2468						
	a 12345678							

Fig 3.2 : Generation of Encoded string through Triangular Encoding

The source string of 8 bit data (a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8) is shown in the first row as bold and the encoded string is shown in the second column as bold in figure 3.2.

$a_{12} = a_1$ XOR a_2 ; a_1 XOR $a_1 = 0$

a_{12} XOR $a_{23} = a_1$ XOR a_2 XOR a_2 XOR $a_3 = a_1$ XOR $a_3 = a_{13}$

The encoded 8 bit string is generated by collecting the bits in the second column (i.e. a_1 a_{12} a_{13} a_{1234} a_{15} a_{1256} a_{1357} $a_{12345678}$). Let the generated string be b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8. Then

$b_1 = a_1$; $b_2 = a_{12}$; $b_3 = a_{13}$; $b_4 = a_{1234}$; $b_5 = a_{15}$; $b_6 = a_{1256}$; $b_7 = a_{1357}$; $b_8 = a_{12345678}$.

So the encoded string will be b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 = a_1 a_{12} a_{13} a_{1234} a_{15} a_{1256} a_{1357} $a_{12345678}$. The encoded string can be used as cipher text in encryption.

3.1.2 Decoding Technique

In order to decode the encoded string (b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8) shown as bold in the second row of figure 3.3, the same operation, as described in section 3.1.1, is applied on it. The trace is given in figure 3.3.

	Encoded String							
D	**b_1**	**b_2**	**b_3**	**b_4**	**b_5**	**b_6**	**b_7**	**b_8**
E	**b_{12}**	b_{23}	b_{34}	b_{45}	b_{56}	b_{67}	b_{78}	
C	**b_{13}**	b_{24}	b_{35}	b_{46}	b_{57}	b_{68}		
O	**b_{1234}**	b_{2345}	b_{3456}	b_{4567}	b_{5678}			
D	**b_{15}**	b_{26}	b_{37}	b_{48}				
E	**b_{1256}**	b_{2367}	b_{3478}					
D	**b_{1357}**	b_{2468}						
string	**$b_{12345678}$**							

Fig 3.3: Decoding of Triangular Encoding

So the second column of figure 3.3 gives the decoded string (b_1 b_{12} b_{13} b_{1234} b_{15} b_{1256} b_{1357} $b_{12345678}$), shown as bold, derived during the encoding process.

3.1.3 Proof of Decoding

To prove the decoding correctness, a source string (a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8) has been considered as in section 3.1.1 and the encoded string is (b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8) derived in encoding operation. This encoded string is decoded in decoding operation as described in section 3.1.2, generates the decoded string (b_1 b_{12} b_{13} b_{1234} b_{15} b_{1256} b_{1357} $b_{12345678}$).

The following derivation shows that the decoded string equals the source string.

$b_1 = a_1$

$b_{12} = b_1 \text{ XOR } b_2 = a_1 \text{ XOR } a_{12} = a_1 \text{ XOR } a_1 \text{ XOR } a_2 = a_2$

$b_{13} = b_1 \text{ XOR } b_3 = a_1 \text{ XOR } a_{13} = a_1 \text{ XOR } a_1 \text{ XOR } a_3 = a_3$

$b_{1234} = b_1 \text{ XOR } b_2 \text{ XOR } b_3 \text{ XOR } b_4 = a_1 \text{ XOR } a_1 \text{ XOR } a_2 \text{ XOR } a_1 \text{ XOR } a_3 \text{ XOR } a_1 \text{ XOR } a_2 \text{ XOR } a_3 \text{ XOR } a_4 = a_4$

$b_{15} = b_1 \text{ XOR } b_5 = a_1 \text{ XOR } a_1 \text{ XOR } a_5 = a_5$

$b_{1256} = b_1 \text{ XOR } b_2 \text{ XOR } b_5 \text{ XOR } b_6 = a_1 \text{ XOR } a_1 \text{ XOR } a_2 \text{ XOR } a_1 \text{ XOR } a_5 \text{ XOR } a_1 \text{ XOR } a_2 \text{ XOR } a_5 \text{ XOR } a_6 = a_6$

$b_{1357} = b_1 \text{ XOR } b_3 \text{ XOR } b_5 \text{ XOR } b_7 = a_1 \text{ XOR } a_1 \text{ XOR } a_3 \text{ XOR } a_1 \text{ XOR } a_5 \text{ XOR } a_1 \text{ XOR } a_3 \text{ XOR } a_5 \text{ XOR } a_7 = a_7$

$b_{12345678} = b_1 \text{ XOR } b_2 \text{ XOR } b_3 \text{ XOR } b_4 \text{ XOR } b_5 \text{ XOR } b_6 \text{ XOR } b_7 \text{ XOR } b_8$
$= a_1 \text{ XOR } a_1 \text{ XOR } a_2 \text{ XOR } a_1 \text{ XOR } a_3 \text{ XOR } a_1 \text{ XOR } a_2 \text{ XOR } a_3 \text{ XOR } a_4 \text{ XOR } a_1 \text{ XOR } a_5 \text{ XOR } a_1 \text{ XOR } a_2 \text{ XOR } a_5 \text{ XOR } a_6 \text{ XOR } a_1 \text{ XOR } a_3 \text{ XOR } a_5 \text{ XOR } a_7 \text{ XOR } a_1 \text{ XOR } a_2 \text{ XOR } a_3 \text{ XOR } a_4 \text{ XOR } a_5 \text{ XOR } a_6 \text{ XOR } a_7 \text{ XOR } a_8 = a_8$

So (b_1 b_{12} b_{13} b_{1234} b_{15} b_{1256} b_{1357} $b_{12345678}$) = (a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8)

Therefore, it proves that the bits generated in the second column of fig 3.3 are the string in original.

The principle of encoding and decoding, shown for 8 bits only, are applicable to a string of n bits. Therefore it is generalized in nature.

The encoding and decoding operations as shown in fig 2.2 and fig 2.3 look like a triangle and hence it is called **Triangular Encoder**. Section 3.2 deals with an example of the process of encoding and decoding

3.2 Example to illustrate Triangular Encoding

A source string in binary has been assumed as an example arbitrarily to illustrate the encoding as well as decoding process. Since it is a symmetric algorithm, both encoding and decoding follow the same technique. Section 3.2.1 illustrates the encoding process whereas the section 3.2.2 demonstrates the decoding.

3.2.1 Encoding Process

An 8 bit string (10110001b = B1h) as source string is assumed for triangular encoding. The string is shown in second row of figure 3.4 as bold. The encoding operation is applied on the string successively till a single bit is generated.

The string, A = a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 = 1 0 1 1 0 0 0 1

	Source String							
E	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**
N	**1**	1	0	1	0	0	1	
C	**0**	1	1	1	0	1		
O	**1**	0	0	1	1			
D	**1**	0	1	0				
E	**1**	1	1					
D	**0**	0						
String	**0**							

Fig 3.4: Encoding of binary string (10110001b)

So a_1 = 1; a_2 = 0; a_3 = 1; a_4 = 1; a_5 = 0; a_6 = 0; a_7 = 0; a_8 = 1.

a_1 xor a_2 = 1 xor 0 = 1; a_2 xor a_3 = 0 xor 1 = 1; a_3 xor a_4 = 1 xor 1 = 0;

a_4 xor a_5 = 1 xor 0 = 1; a_5 xor a_6 = 0 xor 0 = 0; a_6 xor a_7 = 0 xor 0 = 0;

a_7 xor a_8 = 0 xor 1= 1.

Concatenating the bits (1101001) so generated completes the first phase. The third row of figure 3.4 shows 7 bits as discussed. So is the process for generating the fourth row with 6 bits and so on. The second column of figure 3.4 shown as bold is the encoded string.

The encoded string is 1011100b = DCh.

3.2.2 Decoding Process

Let me consider the encoded string, B (11011100) requires to be decoded.

$B = (b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7\ b_8) = (1\ 1\ 0\ 1\ 1\ 1\ 0\ 0)$

So $b_1 = 1$; $b_2 = 1$; $b_3 = 0$; $b_4 = 1$; $b_5 = 1$; $b_6 = 1$; $b_7 = 0$; $b_8 = 0$.

In the first phase the generation of seven bits from the eight encoded bits is shown in the following.

b_1 xor $b_2 = 1$ xor $1 = 0$; b_2 xor $b_3 = 1$ xor $0 = 1$; b_3 xor $b_4 = 0$ xor $1 = 1$;

b_4 xor $b_5 = 1$ xor $1 = 0$; b_5 xor $b_6 = 1$ xor $1 = 0$; b_6 xor $b_7 = 1$ xor $0 = 1$;

b_7 xor $b_8 = 0$ xor $0 = 0$.

Concatenating the seven bits generated from the encoded eight bits gives (0110010) shown in third row of figure 3.5. Following the same process, 6 bits in the fourth row; 5 bits in the fifth row; 4 bits in the sixth row; 3 bits in the seventh row; 2 bits in the eighth row and finally 1 bit in the ninth row of figure 3.5 are shown.

The encoded binary string (1011100) derived from the encoding operation requires to apply the decoding operation which will generate the source string, the goal. The decoding operation is applied using the same principle as encoding and the decoded string is given figure 3.5. The encoded binary string (1011100) is shown as bold in the second row of figure 3.5. The decoded binary string (10110001) is shown as bold in the second column of figure 3.5.

	Encoded String							
D	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**0**
E	**0**	1	1	0	0	1	0	
C	**1**	0	1	0	1	1		
O	**1**	1	1	1	0			
D	**0**	0	0	1				
E	**0**	0	1					
D	**0**	1						
String	**1**							

Fig 3.5: Decoding of an Encoded string (11011100b)

The decoded string can be obtained by concatenating the bits generated on the second column. It is a binary string (10110001) which is the same as the source string considered at the time of encoding.

The process of encoding and decoding is depicted in figure 3.6.

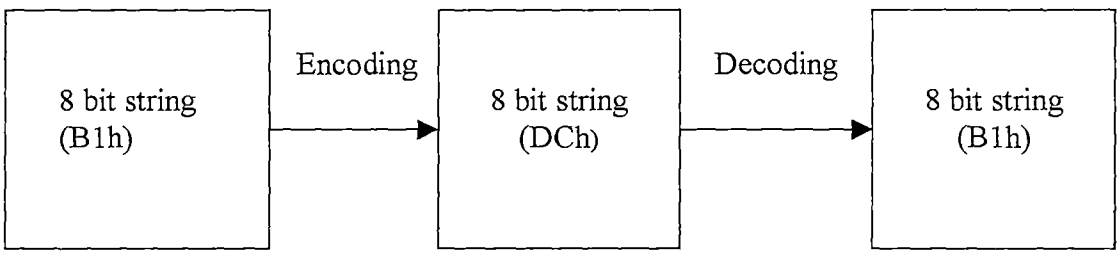

Fig 3.6: Encoding & Decoding operation on a string (10110001b)

The encoding and the decoding operation can be extended to a string of any bit without any loss of generality.

The routines developed for realizing the triangular encoder are given in the section 3.3.

3.3 Realization of Triangular Encoder through Microprocessor based System

The Triangular Encoding operation / transformation is implemented with the help of an 8 bit microprocessor based system, up to 256 bit stream. The specification of the system used for this purpose is already given in chapter II.

The main program along with other routines of the transformation for 16/ 32/ 64/ 128/ 256 bit string has been written in assembly level language. The programs are given is Appendix.

In this section the development of routines for realizing the triangular encoding is described. The routines are generalized in nature. With proper change in parameter in the routines, the same may be used for string of any length, multiple of 8. The routines are termed as 'rotat', 'clr', 'mvtrgt', 'xor' and 'main'. These are described in section 3.3.1 to 3.3.5. There are 4 routines, which are called by the main program.

3.3.1 Routine 'rotat'

This routine can rotate n bytes in memory by one bit left. For the description purpose a 4 byte long string is considered. The string is stored in the memory byte-wise. The Least Significant (LS) byte is stored in lower memory, its address is say, x and the Most Significant (MS) byte is stored in location, (x + 3). The lsb and msb of the string

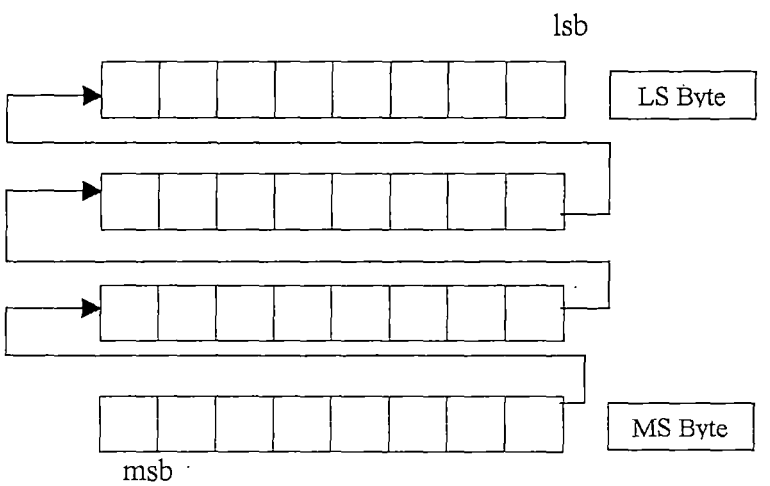

Fig 3.7: 4 byte string stored in memory

are shown in figure 3.7. The rotation of the string by one bit to the left means most significant bit (msb) will go to least significant bit (lsb) position and all other bits will move one bit to the left. To implement it, msb is tested first, whether it is 0 or 1, and set the carry flag accordingly. Then the LS byte is rotated left through carry. The msb will move to the lsb position, other bits of LS byte will move to the left by one bit and msb of LS byte will go to the carry position. The next byte of the string will be rotated through carry and continue till the last byte is reached.

The **algorithm** is given below.

i) The registers used in the routine are pushed in stack and HL pair is pointed to the most significant byte of the string.
ii) The D register is loaded with the masking byte 10000000b.
iii) The register E is loaded with counter value representing the number of byte in the string.
iv) The MS byte is moved to A register.
v) The content of D register is ANDed with A to check the msb of the string.
vi) If the msb of the string is 0, go to (viii).
vii) Else, carry is set, and go to (ix).
viii) The carry is set and complement the carry.
ix) The memory HL is set to the address of the LS byte of the string.
x) The memory content is rotated left through carry.
xi) The memory pointer is incremented.
xii) The byte counter E is decremented.
xiii) Till the byte counter is zero, go to (x).
xiv) The registers are popped.
xv) Return.

3.3.2 Routine 'clr'

This routine clears n bytes from a particular address of the memory.

The algorithm of 'clr' routine is given below.

i) The registers used in the routine are pushed in stack and HL pair is pointed to the first location of the memory.
ii) The C register is loaded with byte count value and A is cleared.
iii) The content of A is stored to memory, pointed by the HL pair.

iv) The memory pointer is incremented to the next location of the memory.

v) The byte counter, C register is decremented by one.

vi) Till the byte counter vale is 0, go to (iii).

vii) Else, the content of the registers are restored by popping.

viii) Return.

3.3.3 Routine 'mvtrgt'

This routine moves the most significant bit of the generated string to the place of the most significant bit in target string.

i) The registers used in the routine are pushed in the stack.

ii) The HL pair is pointed to the last location of the generated string in memory.

iii) The D register is loaded with 1000000b.

iv) The content of the memory is moved to A.

v) The D register is ANDed with A and the result is moved to B.

vi) The memory pointer is moved to the last location of the target string in memory.

vii) The content of B register is ORed with memory content and the result is stored in memory.

viii) The registers are popped.

ix) Return.

3.3.4 Routine 'XOR'

This is very important routine. This routine XORs the successive bits (i^{th} bit and $(i-1)^{th}$) of the string and stores the in i^{th} bit. The process will start from most significant bit and will continue to the least significant bit. The algorithm is given below.

i) The registers used in the routine are pushed in the stack.

ii) The HL register pair, used as memory pointer is set to the last location of the memory where the string is stored.

iii) The register B, byte counter is loaded with count value.

iv) The register D is loaded with the byte, 01000000b.

v) The register C is loaded with 07h, the number of operation for a byte.

vi) The content of the memory is ANDed with D, the result is shifted left by one bit and is stored in E.

vii) The content of D is rotated left by one bit.

viii) The content of memory is moved to A and the content of D is ANDed with A.

ix) The content of E is XORed with A.

x) The zero flag is checked, if it is set, go to (xiv).

xi) Else, the content of memory is moved to A, the D is ORed with A and the result is sent to memory.

xii) The D register is rotated right by 2 bits.

xiii) The C is decremented, till it is zero go to (vi). Else, go to (xvii).

xiv) The unmasked bit position in memory as determined by D is reset to 0.

xv) The masking byte in D is restored and rotated right by 2 bits.

xvi) The C, operation counter is decremented, till it is zero go to (vi).

xvii) The 8^{th} operation is made between lsb of the present location and msb of the previous location and the result is put in lsb of the present location.

xviii) The register B, byte counter is decremented by one.

xix) Till the byte counter is 0, go to (iv).

xx) Else, the registers are restored by pop operation.

xxi) Return.

3.3.5 Main Program

This is the main program which calls the other routines described above to apply the triangular encoding operation.

i) The stack pointer is initialized.

ii) Routine 'clr' is called.

iii) The B is initialized with 10h for 2 byte data.

iv) The 'mvtrgt' is called and then the routine 'rotat'.

v) The counter B is decremented.

vi) The routine 'XOR' is called.

vii) The 'mvtrgt' is called and then the routine 'rotat'.

viii) The counter B is decremented.

ix) Till the B is zero, go to (vi).

x) End.

3.4 Results

A 16 bit source string (0100000101000010b = 4142h) is assumed for Triangular Encoding operation as shown in second row of table 3.1. After the first phase of transformation, 15 bits are generated and are shown in third column of table as bold and the process continues. If we observe the table 3.1 carefully, we will see that upper triangular portion shown as bold is the relevant triangle, we need. However we are not interested for the lower triangular part. This can be eliminated in high level software. But in binary level the storing of bytes is considered in two consecutive locations. And without increasing the software overhead in assembly level, this additional part is allowed in the memory.

Table 3.1: Generation of Encoded string using Triangular

16 bit Input Data																Hex
0	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**4142**
1	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	0	**C3C6**
0	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	1	0	**444A**
1	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	1	1	0	**CCDE**
0	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	0	0	1	0	**5562**
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	0	0	1	1	0	**FFA6**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	1	0	1	0	1	0	**00EA**
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	0	1	1	1	1	1	0	**013E**
0	**0**	**0**	**0**	**0**	**0**	**1**	**1**	0	0	1	0	0	0	1	0	**0342**
0	**0**	**0**	**0**	**0**	**1**	**0**	1	0	1	1	0	0	1	1	0	**0566**
0	**0**	**0**	0	**1**	**1**	1	1	1	0	1	0	1	0	1	0	**0FAA**
0	**0**	**0**	**1**	**0**	0	0	0	1	1	1	1	1	1	1	0	**10FE**
0	**0**	**1**	**1**	0	0	0	1	0	0	0	0	0	0	1	0	**3102**
0	**1**	**0**	1	0	0	1	1	0	0	0	0	0	1	1	0	**5306**
1	**1**	1	1	0	0	1	1	0	0	0	0	1	0	1	0	**F30A**
0	0	0	1	0	1	0	1	0	0	0	1	1	1	1	0	**151E**

Input Data = 0100000101000010b = 4142h

Encoded data = 0101010000000010b = 5402h

The 16 bit input data (0100000101000010b = 4142h) is stored in two consecutive locations. The least significant 8 bit (42h) is stored in the memory (say, F900h address) and the most significant 8 bit data (41h) is stored in next memory location.

After applying the first operation on 4142h data, the generated data will be C3C6h shown in right-most column of table 3.1. To generate the encoded data, 15 such operations are applied successively. The data generated intermediately are shown in the 1st column of the table 3.1. The encoded data is the concatenated binary bits (0101010000000010b = 5402h), generated in the first column of table 3.1.

The experimental result of encoded string and decoded string for a set of arbitrarily assumed source string of 16, 32, and 64 bit block length are presented in table 3.2, 3.3, 3.4 respectively.

Table 3.2: 16 bit Source string, the Encoded string and Decoded string

Source string (hex)	Encoded String (hex)	Decoded string (hex)
4242	5402	4242
5402	4142	5402
56AA	4282	56AA
0000	0000	0000
FFFF	8000	FFFF
8080	FF00	8080
0404	0500	0404
00CC	00A0	00CC
FFB0	805D	FFB0
D1D0	BA01	D1D0
6776	6110	6776
0FF6	0886	0FF6
678B	6193	678B
A45D	C986	A45D
B691	DB34	B691

Table 3.3 : 32 bit Source string, the Encoded string and Decoded string

Source string (hex)	Encoded String (hex)	Decoded string (hex)
01000000	01010101	01000000
22222222	30000000	22222222
33333333	20000000	33333333
77777777	70000000	77777777
04030201	05070605	04030201
40302010	55776655	40302010
22001100	30302020	22001100
99887766	E0109000	99887766
87654321	F89AAFFF	87654321
32002300	21211010	32002300
AABBCCDD	C0106000	AABBCCDD
FFFFFFFF	80000000	FFFFFFFF
00000000	00000000	00000000

Table 3.4 : 64 bit Source string, the Encoded string and Decoded string

Source string (hex)	Encoded String (hex)	Decoded string (hex)
0101010101010101	0100000000000000	0101010101010101
0000000001000000	0000000001010101	0000000001000000
0202020202020202	0300000000000000	0202020202020202
0303030303030303	0200000000000000	0303030303030303
0404040404040404	0500000000000000	0404040404040404
0505050505050505	0400000000000000	0505050505050505
0606060606060606	0600000000000000	0606060606060606
0707070707070707	0700000000000000	0707070707070707
0808080808080808	0F00000000000000	0808080808080808
0909090909090909	0E00000000000000	0909090909090909
0A0A0A0A0A0A0A0A	0C00000000000000	0A0A0A0A0A0A0A0A

Source string (hex)	Encoded String (hex)	Decoded string (hex)
0B0B0B0B0B0B0B0B	0D00000000000000	0B0B0B0B0B0B0B0B
0C0C0C0C0C0C0C0C	0A00000000000000	0C0C0C0C0C0C0C0C
FFFFFFFFFFFFFFFF	8000000000000000	FFFFFFFFFFFFFFFF
0000000000000000	0000000000000000	0000000000000000

Table 3.5: 128 bit Source string and the Encoded string

Input string(hex)	Encoded String(hex)
00000000000000000000000000000000	00000000000000000000000000000000
01010101010101010101010101010101	01000000000000000000000000000000
02020202020202020202020202020202	03000000000000000000000000000000
03030303030303030303030303030303	02000000000000000000000000000000
04040404040404040404040404040404	05000000000000000000000000000000
05050505050505050505050505050505	06000000000000000000000000000000
06060606060606060606060606060606	06000000000000000000000000000000
07070707070707070707070707070707	07000000000000000000000000000000
08080808080808080808080808080808	0F000000000000000000000000000000
09090909090909090909090909090909	0E000000000000000000000000000000
0A0A0A0A0A0A0A0A0A0A0A0A0A0A0A0A	0C000000000000000000000000000000
0B0B0B0B0B0B0B0B0B0B0B0B0B0B0B0B	0D000000000000000000000000000000
0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C	0A000000000000000000000000000000
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	80000000000000000000000000000000

The data for 128 and 256 bit source string and the corresponding encoded data are presented in the table 3.5 and table 3.6 respectively as result. The decoded string corresponding to the encoded string is not presented for clarity for these two cases. The 256 bit data is shown in table 3.6 is split in two rows, the least significant part first, then the most significant part.

Table 3.6 : 256 bit Input string and the Encoded string

Input string(hex)	Encoded String(hex)
00000000000000000000000000000000	00000000000000000000000000000000
00000000000000000000000000000000	00000000000000000000000000000000
01010101010101010101010101010101	00000000000000000000000000000000
01010101010101010101010101010101	01000000000000000000000000000000
02020202020202020202020202020202	00000000000000000000000000000000
02020202020202020202020202020202	03000000000000000000000000000000
03030303030303030303030303030303	00000000000000000000000000000000
03030303030303030303030303030303	02000000000000000000000000000000
04040404040404040404040404040404	00000000000000000000000000000000
04040404040404040404040404040404	05000000000000000000000000000000
05050505050505050505050505050505	00000000000000000000000000000000
05050505050505050505050505050505	04000000000000000000000000000000
06060606060606060606060606060606	00000000000000000000000000000000
06060606060606060606060606060606	06000000000000000000000000000000
07070707070707070707070707070707	00000000000000000000000000000000
07070707070707070707070707070707	07000000000000000000000000000000
08080808080808080808080808080808	00000000000000000000000000000000
08080808080808080808080808080808	0F000000000000000000000000000000
09090909090909090909090909090909	00000000000000000000000000000000
09090909090909090909090909090909	0E000000000000000000000000000000
0A0A0A0A0A0A0A0A0A0A0A0A0A0A0A0A	00000000000000000000000000000000
0A0A0A0A0A0A0A0A0A0A0A0A0A0A0A0A	0C000000000000000000000000000000
0B0B0B0B0B0B0B0B0B0B0B0B0B0B0B0B	00000000000000000000000000000000
0B0B0B0B0B0B0B0B0B0B0B0B0B0B0B0B	0D000000000000000000000000000000
0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C	00000000000000000000000000000000
0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C0C	0A000000000000000000000000000000
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	00000000000000000000000000000000
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	80000000000000000000000000000000

3.5 Security Aspect in Triangular Encoding

The table 3.7 shows the source strings of two 8 bit and one 16 bit string concatenated of two 8 bit strings in the first column and the corresponding encoded strings for the source string in the second column. If we carefully observe the input string and the corresponding encoded string, we see that the 8 bit strings are producing the 8 bit strings and the two concatenated 8 bit strings are producing a 16 bit encoded string which is not equal to the concatenated encoded strings produced from its corresponding 8 bit input strings. 54h and 56h are the encoded strings of 41h and 42h respectively. The concatenated input string 4142h produces the encoded string 5402h, which are not the concatenated encoded strings of 54h and 56h.

Table 3.7: Input string of 8 and 16 bit

Source String (hex)	Encoded String (hex)
41	54
42	56
4142	5402

The concatenation of input strings produces a separate encoded string, other than the concatenated encoded strings produced from the input strings. This property may be very useful as far as the security of the string is concerned. This will generate the randomness in the encoded string of large length. The brute-force attack for 256 bit is not very easy with the simple machines. The program developed is applicable up to 256 bit string length. This string length may be increased further to enhance the security of the encoder.

3.6 Memory Efficient Program for Triangular Encoding in Assembly Level

The program of triangular encoding is developed so judiciously that no extra memory is required during the time of encoding as scratch pad. The intermediately generated data is saved in the same place of memory where the input string is stored initially. Only the size of the memory, equals to its string is required for the

encoding. For example, 16 bit input string requires 16 bit another memory space for its encoded data.

As a result the program developed is a memory efficient one for its encoding operation.

3.7 Number of operations required for Triangular Encoding

An 8 bit string is considered as data, on which the triangular encoding operation will be applied. The (8 – 1) XOR operations are required in the first phase or cycle, (8 – 2) XOR operations in the second phase and so on till it is 1.

So (8 – 1) cycles and (7 + 6 + 5 + 4 + 3 + 2 + 1) XOR operations are required for 8 bit string in the encoding operation. For n bit string, there will be (n - 1) cycles and [n (n – 1) / 2] XOR operations required to generate n bit encoded string. This is very much useful in high level realization.

3.8 Comparison of Triangular Encoding with PP Encoding and RSA Encoding

A comparative study of Triangular Encoding with PP Encoding and RSA Encoding has been made in order to grade the developed encoding. For this purpose two following methods are adopted as in the previous chapter.

1. Frequency Distribution of Encoded file with PP Encoding
2. Homogeneity Test (Chi-square) by a statistical method

The Frequency Distribution and Homogeneity Test presentations are given in section 3.8.1 and 3.8.2 respectively.

3.8.1 Frequency Distribution of Encoded file

To find out the frequency distribution of characters in encoded file with respect to the source file under the technique, a text file (prt.txt, size: 11351 Bytes) is taken. The triangular Encoding technique has been applied and then the distribution of the characters of the text file and that of encoded file is plotted in figure 3.7. The distribution of the characters of the text file and that of encoded file for PP Encoding is plotted in figure 3.8. The distribution of the characters of the text file and that of encoded file with RSA is plotted in figure 3.9. The blue lines in the bar graph show

the distribution of characters for the text file, while the red line for encoded file for figures 3.7 and 3.9.

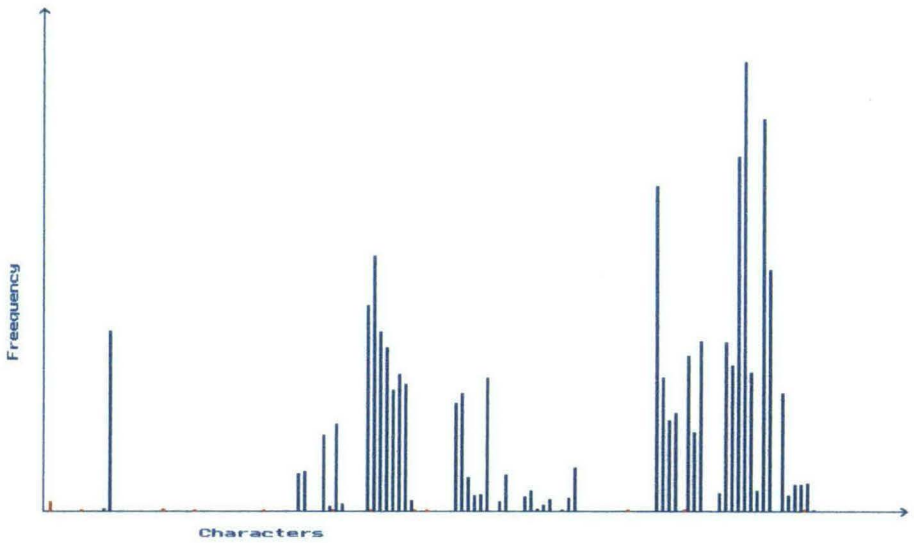

Figure 3.7: Frequency Distribution of characters in source message and encrypted message under Triangular Encoding

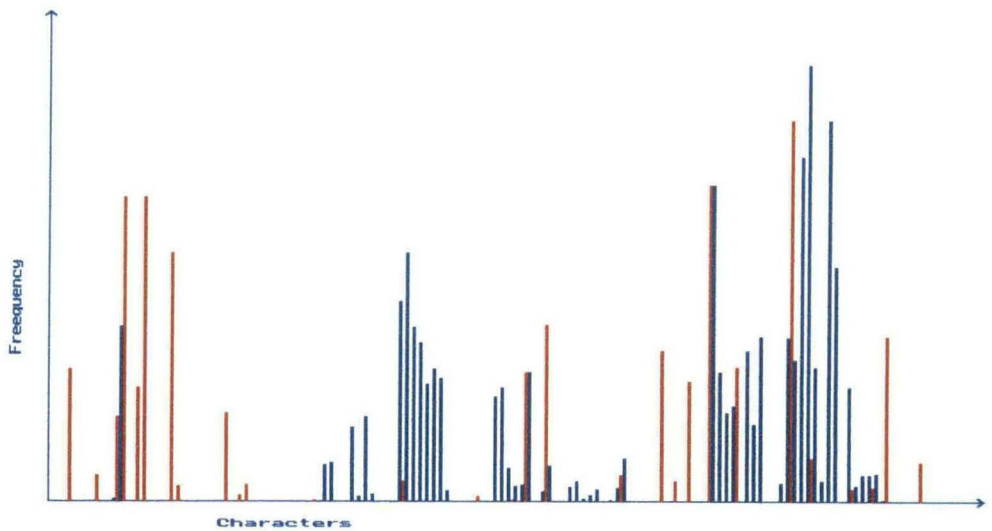

Figure 3.9: Frequency Distribution of characters in source message and encrypted message under RSA Encoding

3.8.2 Homogeneity Test (Chi-square)

For homogeneity test of the encrypted file generated through Triangular Encoder, the Chi-square method, a statistical procedure has been applied. A text file (g.abc) of size, 2157 byte is used for encryption. The corresponding encrypted files are generated for different block length. For each case the Chi-square value is computed. The maximum value for this text file is calculated as 2861.56 and the average value of all the values is 2389.27. The table 3.8 shows the Chi-Square values and the figure 3.10 shows the bar graph for different block lengths.

Table 3.8: Chi-Square value for different Block Length of Triangular Encoder

Sl.No.	Block Length	Text File	Encrypted File	File Size	Chi-Index
1	8	g.abc	Tr8.abc	2,157	2861.56
2	16	g.abc	Tr16.abc	2,157	2520.92
3	32	g.abc	Tr32.abc	2,157	2355.87
4	64	g.abc	Tr64.abc	2,157	2304.06
5	128	g.abc	Tr128.abc	2,157	2188.15
6	256	g.abc	Tr256.abc	2,157	2105.05
				Average Chi-Index	**2389.27**

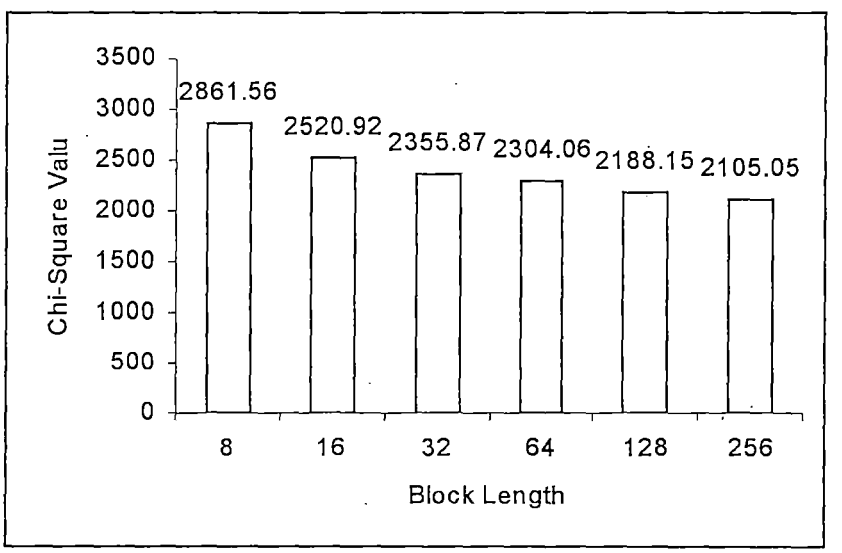

Figure 3.10: Chi-Square value for different Block length of Triangular Encoder

For homogeneity test of the encrypted file generated through PP Encoder, the same Chi-square method has been applied. The same text file (g.abc) of size, 2157 byte is used for encryption. The corresponding encrypted files are generated for different block length. For each case the Chi-square value is computed. The maximum value for this text file is calculated as 2423.77 and the average value of all the values is 1052.40. The table 3.9 shows the Chi-Square value and the figure 3.11 shows the diagram for different block lengths. Table 3.9m and figure 3.11 are presented here for comparative study of Triangular Encoder with the PP Encoder.

Table 3.9: Chi-Square value for different Block Length of Prime Position Encoder

Sl.No.	Block Length	Text File	Encrypted File	File Size	Chi-Square value
1	8	g.abc	Pr8.abc	2,157	2423.77
2	16	g.abc	Pr16.abc	2,157	662.23
3	32	g.abc	Pr32.abc	2,157	1817.52
4	64	g.abc	Pr64.abc	2,157	563.04
5	128	g.abc	Pr128.abc	2,157	410.77
6	256	g.abc	Pr256.abc	2,157	437.07
				Average Chi-Index	**1052.40**

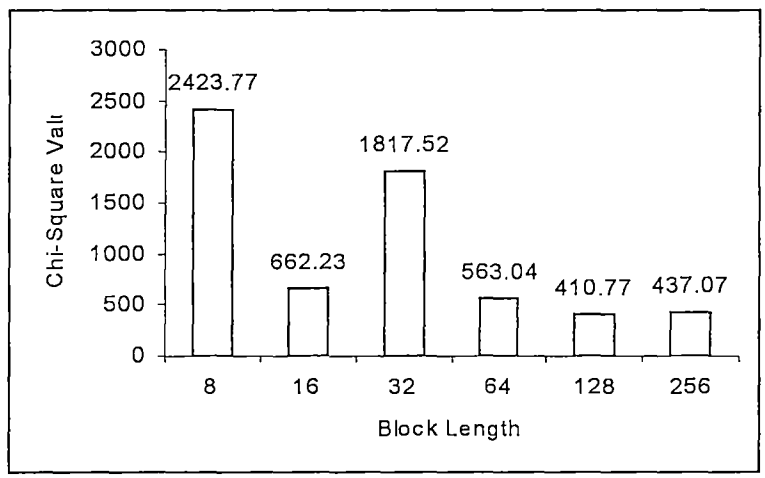

Figure 3.11: Graph of Chi-square value for different Block length of PP Encoder

The Chi-Square value for the same file under RSA encryption is 2359.03. The chi-Square value for 8 bit block length under PP Encoder is 2423.77 which is higher than that of RSA. The table 3.10 shows the Chi-square value for PP Encoding, Triangular Encoding and RSA Encoding. The Chi-square value of Triangular Encoder and that

of PP Encoder under 8 bit is comparable with that of RSA encoding. The pictorial diagram is shown in figure 3.12.

Table 3.10: Chi-square values for different Encoders

Sl. No	Encoders	Chi-square Value
1	PP Encoding	1052.40
2	Triangular Encoding	2389.27
3	RSA Encoding	2359.03

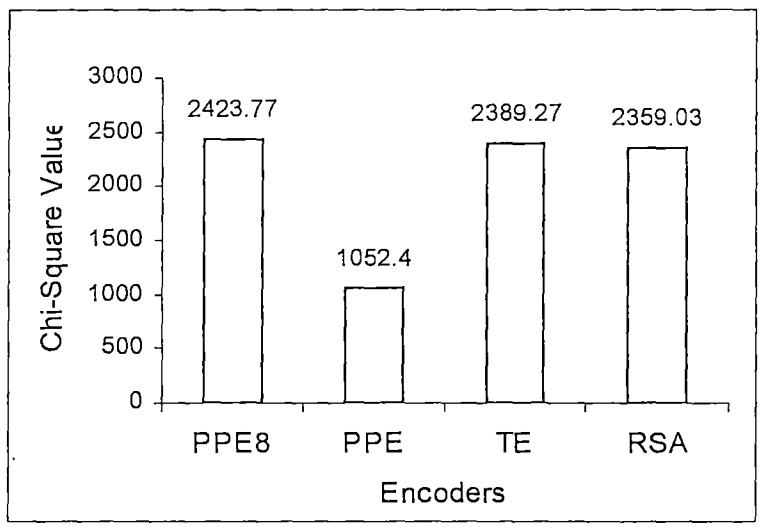

Figure 3.12: Comparison of Encoders

To analyze the encoder, ten different files of different size are considered and chi-square value for each is computed for different block length, from 8 to 256 bit as given in table 3.11. The average chi-square value is also computed in each block size and the pictorial diagram is shown in figure 3.13. The average chi-square value with same ten files under RSA is 3144.715 as shown in table 3.12. The average chi-square value of Triangular Encoder with 8, 16 and 32 bit is higher than that of RSA. Others are quite comparable with RSA. The overall chi-square value of Triangular Encoder is 3313.21, which is again higher than that of RSA.

Table 3.11: Chi-Square values for different files under Triangular Encoder

Triangular Encoder									
Sl no	Source File	Source File size (byte)	Chi-square For 8 bit Length	Chi-square For 16 bit length	Chi-square For 32 bit length	Chi-square For 64 bit length	Chi-square For 128bit length	Chi-square For 256bit length	Ave Chi-Square
1	a.abc	904	1121.94	1068.4	951.74	896.1	838.95	839.04	
2	b.abc	1061	1375.67	1205.6	1152.42	1152.93	1128.63	1116.81	
3	c.abc	907	1208.59	1029.99	966.96	975.8	948.21	937.24	
4	d.abc	2841	3165.98	3291.88	3065.41	2966.2	2870.36	2804.04	
5	e.abc	1765	2266.89	2030.04	1917.25	1910.28	1881.12	1850.23	
6	f.abc	2227	3027.19	2721	2494.09	2410.25	2389.46	2351.56	
7	g.abc	2157	2861.56	2510.92	2355.87	2304.06	2188.15	2105.05	
8	h.abc	7121	9373.46	8283.96	7546.94	7376.81	7268.61	7167.57	
9	i.abc	8830	11592.94	10293.64	9480.21	9108.76	9040.43	8847.35	
10	j.abc	2182	2947.55	2652.19	2415.53	2287.17	2251.53	2204.05	
Average		2999.5	3894.177	3508.762	3234.64	3138.83	3080.54	3022.29	3313.21

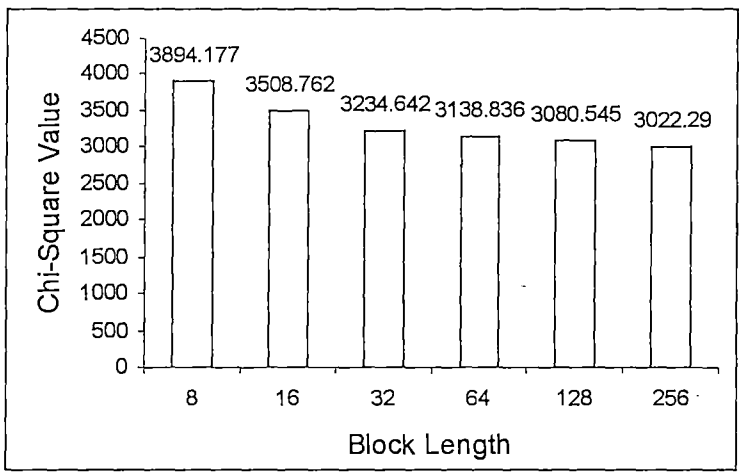

Figure 3.13: Comparison of Chi-square value for different Block length under Triangular Encoding

Table 3.12: Chi-Square value under RSA Encoder

Sl no	Source File	Source File size	Chi-Square value under RSA Encoder
1	a.abc	904	953.21
2	b.abc	1061	1135.65
3	c.abc	907	997.82
4	d.abc	2841	2988.19

Sl no	Source File	Source File size	Chi-Square value under RSA Encoder
5	e.abc	1765	1892.01
6	f.abc	2227	2406.50
7	g.abc	2157	2359.03
8	h.abc	7121	7311.02
9	i.abc	8830	9085.07
10	j.abc	2182	2318.65
Average Chi-Square Value			**3144.715**

Table 3.13: Comparison of Encoders on Chi-square value

Encoders	**Chi-Square Value**
PPE	1373.86
PPE8	3272.24
TE	3313.21
RSA	3144.715

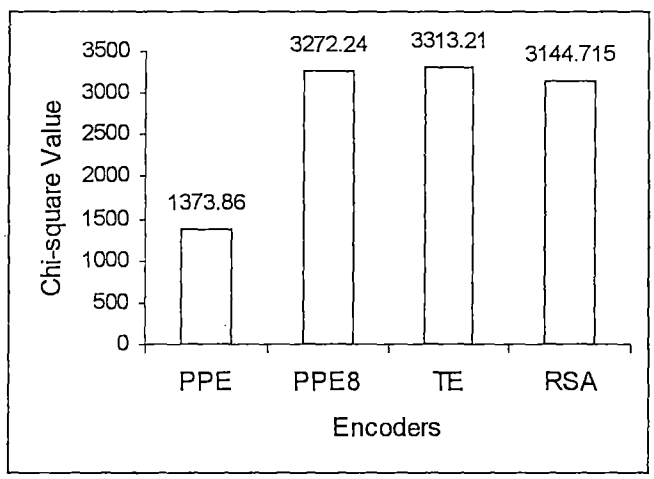

Figure 3.14: Graph for comparing Chi-Square value of Encoders

Table 3.13 shows chi-square values for different encoders. Figure 3.14 shows the pictorial diagram for the comparison of Chi-Square value of PP Encoder, Triangular Encoder (TE) with that of RSA Encoder. The Chi-square value against PPE is for overall average value of chi-square under Prime Position Encoder, that against PPE8 for average value of chi-square under Prime Position Encoder with 8 bit block length, that against TE for average value of Chi-square under Triangular Encoder and that against RSA for average value of RSA encoder.

The realized encoders are comparable with RSA with respect to Chi-square value.

3.9 Conclusion

Like PP Encoding the triangular encoding has no additional generation of bits. So there is no overhead in this encoder also. This triangular encoder can be used for encryption purpose if the technique is realized in high level language.

The brute-force attack may be applied for hacking the message. The block length of 128 bit and higher will be a troublesome to decode the message.