SYNOPSIS
Development of Microprocessor-based Encoders for Secured Transmission

Introduction:

We are in the age of global communication systems. A communication system exchanges information between source and destination. It connects a data source to a data user through a channel. The most widely used method of communicating over distances is through electrical signal, either over cables or through free space using radio wave. Data when enter the communication system from the data source, are first processed by a source encoder and it is called the source code. This source code is processed by the channel encoder, which transforms a sequence of source code into another sequence called the channel code. Next the modulator converts each channel code into corresponding analog signal and transmitted through the channel. A demodulator demodulates the signal into received code at the receiving end. Then, the signal passes a channel decoder, which converts the coded signal into source code at the destination.

In general, there are three type of problems associated with such data transmission. Firstly, a large volume of data has to be handled. Secondly, much of the data is very sensitive to errors and thirdly, the most important; a lack of security exists when a volume of data is transferred from its source to the destination if no measure is taken for its security [1,2,3,4,5,6,7,8,9].

The process of adding security to a message is called encryption. In the process of encryption some measure is taken to encapsule the actual information to make it well secured. Hence the process by which an unprotected message is transformed into ciphertext of an unintelligible form is known as encryption. In order to restore the original message, a deciphering technique is used. This process is known as decryption or decipherment. The ciphering facility for conversion of plaintext into ciphertext and vice versa is known as a cryptographic system or cryptosystem [10].

A number of cryptographic encoder has been developed over the years from ancient times. Each has its advantages and disadvantages. Generation of crypto-models is such a technique where research for the models on cryptosystem is a continuous and growing need against the threat of privacy violations [7,9].

Cryptosystem can basically be of two types – symmetric cryptosystem and asymmetric cryptosystem. In symmetric cryptosystem, the same key is used to encrypt and decrypt a message. This key is called secret key and hence it is better known as secret key cipher system. Asymmetric cryptosystem uses one key (the public key) to encrypt a message and a different key (the private key) to decrypt the message [4,11,12,13,14,15].

From e-mail to cellular communication, from secured web access to digital cash, cryptography is an essential part of today's information systems. It helps to provide accountability, fairness, accuracy and confidentiality. It can prevent fraud in electronic commerce and assure the validity of financial transactions. It can prove one's identity and protect one's anonymity. It can keep away vandals from altering one's web page and prevent industrial competitors from reading one's confidential documents. As commerce and communications continue to move to computer networks as e-commerce, cryptography is becoming vital issue in communication. This electronic commerce schemes may fall fraud through forgery, misrepresentation, denial of service and cheating if we do not add security to these systems. In fact, computerization makes the risks even greater by allowing attacks that are impossible in non-automated systems. Only strong cryptography can protect against these attacks [5].

A cryptographic system may be obtained in two ways

i. Through software algorithms and its implementations using High Level Language
ii. Through design of microprocessor based encoder and implementation through Assembly Level Language.

The objective of the present study is to design microprocessor based encoder to implement the same through assembly level language [16,17,18,19,20,21].

In the present study, some microprocessor based encoding systems for encapsulations of data to enhance the security of the message have been developed. Its principle has been verified through implementation using microprocessor-based system.

All techniques considered the inputs as stream of bits. The stream is divided into blocks of size k which varies from 1 to n. For the present implementations, a block of bits, varying from 8 to 256 bits has been selected on which the developed schemes are applied. The block size may be increased to any value beyond 256 as the algorithms proposed are generalized in nature.

In the present study six different techniques have been implemented through the microprocessor based system. These are

1. Prime Position Encoding (PPE)
2. Triangular Encoding (TE)
3. Recursive Pair Parity Encoding (RPPE)
4. Rotational Encoding (RE)
5. Johnson Encoding (JE)
6. Bit swap Encoding (BSE)

The cascading of implemented encoders is also proposed in this study.

For each encoder the frequency distribution of characters has been studied, a corresponding graph has been drawn and a statistical method, called chi-square test, has been performed to test the non-homogeneity of the encoded file in comparison with the source file.

Section 1 of the present study deals with Prime Position Encoding (PPE). Section 2 considers the Triangular Encoding (TE) Technique. Recursive Pair Parity Encoding (RPPE) has been discussed in section 3. Rotational Encoding (RE) is discussed in section 4. Section 5 deals with Johnson Encoding (JE) and section 6 Bit swap Encoding (BSE). The cascading of the schemes is presented in section 7.

1. Prime Position Encoding (PPE)

1.1 Principle of Prime Position Encoding

The technique accepts the binary level information and transforms the same into a cipher-text based on the principle of PP Encoding. In this encoding, a block of data, X, is considered in the form of binary as

$X= X_n X_{n-1} ……..X_i …….X_3 X_2 X_1$ i varies from 1 to n

where X_1, X_2, ……X_n are the bits of the block, X_1 being the first position (least significant bit), X_2 being the second position and so on in the block X_n being the msb . The transformed block,

$Y =Y_n Y_{n-1} ……. Y_i …….. Y_3 Y_2 Y_1$ i varies from 1 to n as described in X.

The transformation is made through reorientation of bit positioning generated through PP encoding on the basis of prime position as stated below.

The transformation for source block to encoded block is made using following operations. Considering that a block contains n bit

i) The bit in i^{th} position of X will move to the j^{th} position of Y.

$X_i \rightarrow Y_j$ where the symbol → indicates ‘moves to’

for i =1 to (n-1)

j = (n-i) if j is non-prime

else j= previous prime of (n-i)

When j is the first prime number, then the last prime number will be considered as previous prime in n-bit block.

ii) $X_i = Y_i$ for i= n

So the content of i^{th} position in X will occupy the j^{th} position in Y. The string X having n bit would generate string Y with same number of bit.

1.2 Realization of PP Encoder using Microprocessor Based System

A generalized technique has been developed to realize the encoder through microprocessor-based system. The program has been developed in assembly level language for the length of the block up to 256 bits. The technique may be applied for higher bit length also.

To generate the generalized algorithm for PP Encoder a 16 bit block is considered. The algorithm required for 16 bit string is developed first. Then the concept is extended to higher bit, multiple of 8.

Considering that the source data, X is stored in two adjacent locations in the memory. The transformed data, Y would be stored in another two adjacent locations. This is also termed as Target data.

For the transformation, it can be considered that the bit in a particular position of the source data will occupy a particular position of the Target data as per the proposed encoding principle. The logic of the bit movement from the source to the target is defined as

X_i moves to Y_j for $i = n = j$

X_i moves to Y_j for i = 1 to (n-1)

where $j = n - i$ when j = non-prime

= previous prime of j

To realize efficiently up to 256 bit length string, the bit position of the source as well as that of target requires to be designated differently.

The principle is explained with the help of an example. Consider the source data as **76DD**h stored in two adjacent locations of the memory. The 0^{th} bit (**0**) of X is moved to the 14^{th} bit position of Y, the 1^{st} bit to the 13^{th}, and so on. After the transformation, the source data 76DDh is converted to **5FA7**h.

The 0^{th} bit of source data moves to the 14^{th} bit position of the target data i.e. to the 1^{st} row, 6^{th} column position of the target data. This target position (1^{st} row, 6^{th} column) is coded in 8 bits as 00001 110b or 0Eh. The 5 most significant bits show the row number and 3 least significant bits the column. The target information for each bit in source is coded and is used as table for transformation. This coding can accommodate 32 rows and 8 columns. The maximum length of the block will be 32 x 8 = 256 bits. So 256 bit string can be transformed to a target string of 256 bit.

This target information is stored in memory for transformation. The program will collect the information of the target from the stored memory and set the target to either 0 or 1. The process of transformation can be made faster, if the target string is made clear initially and the only 1s of the source data are considered for transformation in order to set the corresponding bits of target string. Then the bits of source block, which are 1, need to be sent to the corresponding the bit position of the target block consulting a table so generated.

So the bit position of 1s in the source block is required to be detected and the number of 1s present in the source block using a routine.

Another routine will collect the position from F900h address .i.e. 01 here. This position value will be added to the base address (FB00h) of the memory as given in the table 1.6. The final address will be FB01h and the content of FB01h is 0Dh which is the target position (1st row, 5th column) of 1st bit, where the bit has to be set to 1. In this way all the 8 bits of 1 in X will virtually move to the target positions in Y.

The principle of implementation described above has been extended to 256 bit block with the required modification. However the principle is generalized in nature and may be extended for higher block length.

The algorithms of different routines of generalized PP Encoding scheme are given in section 1.3 and that of main in section 1.4.

1.3 Algorithm of Different Routines in Assembly Level

The routines are developed for realizing the PP Encoder are given in section 1.3a to 1.3g. The routines are generalized in nature. With proper change in parameter in the routines, these may be used for any string, multiple of 8. The algorithms are written for 256 bit string. The registers described in algorithms below are A (Accumulator), B, C, D, E, H, L, SP (Stack Pointer) and PC (Program Counter). The BC, DE and HL are used as a pair of registers.

1.3a) Routine '**save**'

The routine saves the string stored from FA00h onwards to the save area which starts from F9B0h onwards.

1. The D register is used as counter, loaded with 20h.
2. The HL pair is used as pointer pointed to F9B0h, the destination
3. The BC pair is used as pointer pointed to F9B0h, the source.
4. The memory content pointed by BC pointer is moved to A.
5. The content of A is moved to destination.
6. The HL and BC pairs are incremented
7. The counter register, D is decremented.
8. Untill the counter is exhausted, go to (4)
9. Return

1.3b) Routine '**b**'

This routine clears the temporary result area starts from FA20h onwards for 20h bytes.

1. The HL pair is used as memory pointer pointed to FA20h
2. The register A is cleared.
3. The register C, used as counter is loaded with 20h.
4. The content of A is moved to memory.
5. The memory pointer, HL pair is incremented.
6. The counter is decremented.
7. Till the counter is exhausted, go to (4)
8. Return

1.3c) Routine '**a**'

This routine finds the position of 1s in the string and stored the position in memory location FE00h onwards. This routine also finds the number of 1s in the string and stored in FDFFh.

1. The register A is loaded with count value, 20h and saved in FE00h
2. The HL pair used as pointer is pointed to memory location FE00h.
3. The BC pair is pointed to memory location FA00h.
4. The register D is cleared.
5. The register E is loaded with 08h.
6. The content of the memory pointed by the BC pair is moved to the regis A.

7. *The content of A is rotated right through carry.*
8. Jump on no-carry to (11).
9. On carry, the content of D will move to the memory pointed by the HL pair.
10. The HL pair is incremented.
11. The D register is incremented.
12. The E register is decremented.
13. Until the register E is exhausted, go to (7).
14. Else the BC pair is incremented.
15. The count value is retrieved and decremented and pushed back to the stack.
16. Until the count value is exhausted, go to (5).
17. The content of L is moved to the memory location FDFFh.
18. Return

1.3d) Routine '**c**'

This routine checks the number of 1s stored in memory location FDFFh by the routine '**a**'. If it finds any positive number, then the memory pointer is set to FE00h and calls 'prg' routine for number of 1s stored in FDFFh.

1. The HL pair used as pointer is set to the memory location FDFFh.
2. The content of the memory location FDFFh is moved to C register.
3. If the memory content zero, then return.
4. The HL pointer is incremented to FE00h.
5. The routine '**prg**' is called.
6. The register C is decremented.
7. Till the content of register C is exhausted, go to (4).
8. Return.

1.3e) Routine '**prg**'

This routine is the important one. It takes the HL pair being the memory pointer as parameter. The content of the memory is the 1s' bit position of the string. It needs to be moved to the target position as per the table, starts from FB00h onwards. The routine performs the main function of PP Encoding.

1. The content of the memory is moved to the register L.
2. The register H is loaded with FBh.
3. The memory content is moved to register A and C register. This 8 bit data gives the row and column information of the position of the target. The 5 most significant bits give the row and the 3 least significant bits the column information.
4. The row information is derived from the data, stored in register E and in memory FAFFh.
5. The column information is derived from the data and stored in FAFFh.
6. The BC pair is set to FAFEh.
7. The HL pair is set to FA20h, the base address of the result area.
8. The content of the memory, pointed by BC pair is moved to register A.
9. If the content of A is zero, go to (11).
10. The HL pair is incremented and the A register is decremented till the content of A is zero.
11. The BC pair is pointed to the memory location FAFFh.
12. The content of FAFFh is moved to the register A.
13. The register D is loaded with 00000001b.

14. If the content of A is zero, go to (20).
15. Else, the content of A is moved to the register C.
16. The content of register D is moved to the register A.
17. The content of A is rotated left and the register C is decremented.
18. Until the content of C is exhausted, go to (17).
19. The content of A is moved to D register.
20. The content of A is ORed with that of the memory and the result is in A.
21. Return.

1.3f) Routine **'compr'**

This routine compares the data from location F9B0h onwards with that of FA20h onwards for 20h bytes.

1. The HL pair and BC pair are pointed to F9B0h and FA20h respectively.
2. The register D used as counter is loaded with 20h.
3. The content of memory pointed by BC pair is moved to A.
4. The content of A is compared with that of the memory, pointed by the HL.
5. If not zero, go to (9).
6. Else, the HL pair and BC pair are incremented.
7. The register D is decremented.
8. Until the content of D is exhausted go to (3).
9. Return.

1.3g) Routine '**supply**'

This routine is used to supply the generated string at FA20h onwards to FA00h onwards for 20h bytes.

1. The HL and BC pair are pointed to FA00h and FA20h respectively.
2. The register D used as counter is loaded with 20h.
3. The content of the memory pointed by BC pair is moved to A.
4. The content of A is moved to the memory pointed by the HL pair.
5. Both the BC and HL pairs are incremented.
6. The content of D register is decremented.
7. Till the content of D is exhausted, go to (3).
8. Return.

1.4) Algorithm of the Main Program for PP Encoder

The main program for PP Encoding calls the routines described above for the transformation. When the routines are called, the registers used for it are properly saved in the stack and at the time of leaving the routine the previous condition is restored by popping.

1. The stack pointer SP is initialized at the highest address of the usable RAM.
2. A register pair HL is used as pointer for iteration / cycle is initialized.
3. Another pointer used for storing the result (the transformed block) is saved in memory.
4. The routine '**save**' is called to save the string from FA00h onwards to F9B0h onwards.
5. The routine '**b**' is called the temporary result area.
6. The routine '**a**' is called to find the positions of 1s in the string and store from FE00h onwards.
7. The routine '**c**' is called to check for presence of 1s in the string and calls 'prg' for the transformation consulting the table stored FB00h onwards.

8. The content of the memory pointed by the pointer is incremented.
9. The routine '**compr**' is called to compare the generated string with the saved string.
10. If the transformed block is equal to the original block, the result displayed and go to (12).
11. Else, the generated result is supplied to the location from where the transformation will begin, by calling the routine '**supply**' and go to (5).
12. Stop and end.

The above algorithm is implemented in assembly level for 16 bit initially in order to verify the transformation. The same algorithm is extended to 256 bit block with the microprocessor based kit. It is worth pointing that there is no limitation in increasing the length of the block, if the microprocessor based system supports with large memory. For the bit-length higher than 256 bit, the destination of the target has to be coded for more than 8 bits.

To visualize the frequency distribution of characters of the source file and the encoded file, a frequency distribution graph has been plotted in fig 1. The blue lines show the characters in the source while the red lines show the characters in the encoded file.

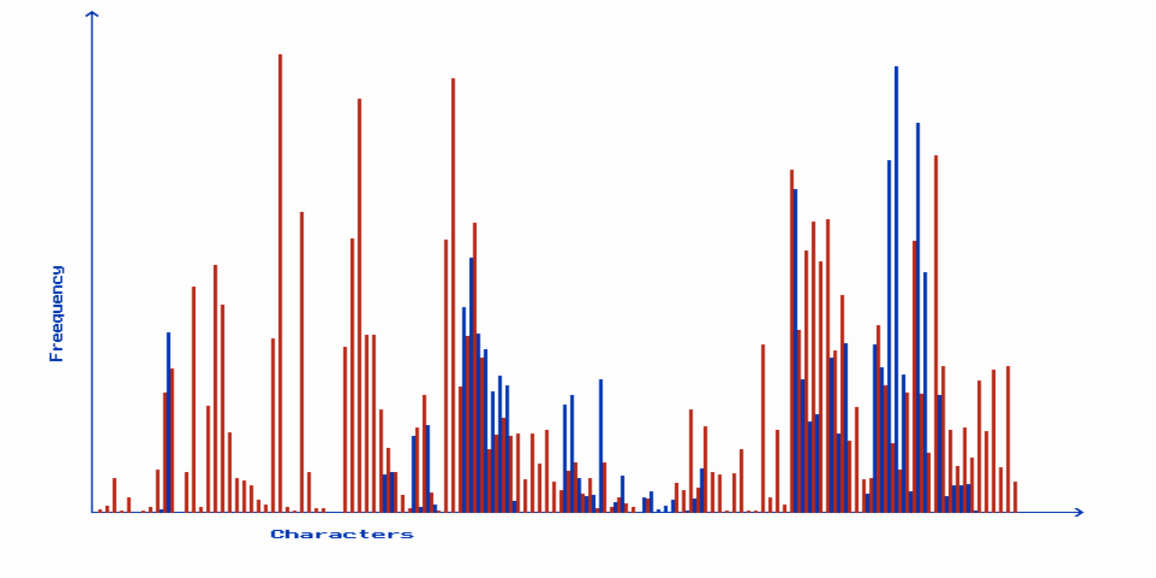


Fig 1: Frequency Distribution of characters in source message and encoded message under Prime Position Encoding

The Chi-square test has also been performed for source file and encoded file. Table 1 shows the values of Chi-square for different bit stream length. Hence the Chi-square is highly significant at 1 % level of significant.

Table 1: Chi-square value of PP Encoder for different bit length

Sl no	Source File	Source File size	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	990.86	362	771	258.71	195.18	173.21	
2	b.abc	1061	1106.68	370.32	903.4	265.37	265.37	246.75	
3	c.abc	907	984.74	301.21	769.58	228.42	228.42	215.09	

Sl no	Source File	Source File size	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
4	d.abc	2841	3090.79	536.96	2359.34	673.25	458.53	478.49	
5	e.abc	1765	1866.81	384.07	1532.2	480.74	332.36	336.88	
6	f.abc	2227	2580.27	501.98	1898.17	633.79	477.68	504.04	
7	g.abc	2157	2423.77	662.23	1817.52	563.04	410.77	437.07	
8	h.abc	7121	7686.52	1811.56	5809.87	1668.83	1169.3	986.9	
9	i.abc	8830	9559.08	1242.22	7392.4	2081.02	1189.76	1627.33	
10	j.abc	2182	2432.87	481.64	1779.85	558.12	425.06	452.05	
Average		2999.5	3272.24	665.419	2503.33	741.129	515.243	545.781	1373.86

2. Triangular Encoding (TE)

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 process looks like a triangle and hence it is termed as triangular encoder. The generated bits in the triangle give the encoded string. The encoding and decoding operation are discussed in section 2.1.

2.1 Principle of Triangular Encoding

i) Encoding Process

The stream of binary bits are grouped into a finite block length, n and fed to memory. A = a_1, a_2, ... a_n are the bits of the string, a_1 the most significant bit (msb) and a_n the least significant bit (lsb) of the string.

The encoding operation starts from most significant bit and continues up 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 $a_{n-1,n}$. Thus it generates n-1 bits in the process. The same operation is repeated on the intermediate stream of bits generated in each cycle till it generates single bit. The output of each stage of operation for 8 bit stream is given in figure 2.1.

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

Fig 2.1 : Generation of Encoded string

The input 8 bit stream (a_1 to a_8) is shown in the first row. After the first encoding operation, it generates 7 bit stream shown in the second row in the figure. The operation

continues till the output is a single bit data ($a_{12345678}$) as shown in the last row in figure 2.1. The operation may be described as follows

$a_{12} = a_1$ 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 most significant bits i.e. $a_1\ a_{12}\ a_{13}\ a_{1234}\ a_{15}\ a_{1256}\ a_{1357}\ a_{12345678}$. Let the generated string be **B = {**$b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7\ b_8$**}** . 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 **= {**$b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7\ b_8$**}**. The encoded string can be used as cipher text in encryption.

ii) Decoding Process

In order to decode the encoded string, the same encoding operations are to be applied on the encoded string B **= {**$b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7\ b_8$**}**. The intermediate string for each step of decoding and hence the source stream generated is given in figure 2.2.

Encoded String							
b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
b_{12}	b_{23}	b_{34}	b_{45}	b_{56}	b_{67}	b_{78}	
b_{13}	b_{24}	b_{35}	b_{46}	b_{57}	b_{68}		
b_{1234}	b_{2345}	b_{3456}	b_{4567}	b_{5678}			
b_{15}	b_{26}	b_{37}	b_{48}				
b_{1256}	b_{2367}	b_{3478}					
b_{1357}	b_{2468}						
$b_{12345678}$							
Decoded String	$b_1\ b_{12} b_{13}\ b_{1234} b_{15}\ b_{1256}\ b_{1357}\ b_{12345678} = a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8$						

Fig 2.2 : Decoding of Encoded string

The decoded string A=(a_1 ……a_8)= $b_1\ b_{12} b_{13}\ b_{1234} b_{15}\ b_{1256}\ b_{1357}\ b_{12345678}$. The proof of decoding is given as follows.

Consider the encoded string as B **= {**$b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7\ b_8$**}** and the decoded string generated as **{**$b_1\ b_{12} b_{13}\ b_{1234}\ b_{15}\ b_{1256}\ b_{1357}\ b_{12345678}$**}** and the decoded string is equal to the source string A **= {**$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8$**}**.

$b_1 = a_1$

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

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

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

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

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

$b_{1357} = b_1$ XOR b_3 XOR b_5 XOR $b_7 = a_1$ XOR a_1 XOR a_3 XOR a_1 XOR a_5 XOR a_1 XOR a_3 XOR a_5 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 this proves the decoding correctness.

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 form a triangle, if we see all intermediate bits as a whole and hence it is termed as **Triangular Encoder**.

2.2 Realization of Triangular Encoder through Microprocessor based System

The Triangular Encoding operation / transformation has been implemented with the help of an 8 bit microprocessor based system, up to 256 bit block length.

2.3 Algorithm of Different Routines for TE

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 algorithm for main program and all related routines are given section 2.3.1 to 2.3.5.

There are 4 routines, which are called by the main program stated below.

2.3.1 Routine 'rotat'

This routine can rotate n bytes in memory by one bit left. To illustrate the technique, consider a 4 byte long string. 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 rotation of the string by one bit to the left means msb will go to 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.

1. The registers used in the routine are pushed in stack and HL pair is pointed to the most significant byte of the string.
2. The D register is loaded with the masking byte 10000000b.
3. The register E is loaded with counter value representing the number of byte in the string.
4. The MS byte is moved to A register.
5. The content of D register is ANDed with A to check the msb of the string.
6. If the msb of the string is 0, go to (7).
   Else, carry is set, and go to (8).
7. The carry is set and complement the carry.
8. The memory HL is set to the address of the LS byte of the string.
9. The memory content is rotated left through carry.
10. The memory pointer is incremented.
11. The byte counter E is decremented.
12. Until the byte counter is zero, go to (10).
13. The registers are popped.
14. Return.

2.3.2 Routine 'clr'

This routine clears n bytes from a particular address of the memory. The algorithm of 'clr' routine is given below.

1. The registers used in the routine are pushed in stack and HL pair is pointed to the first location of the memory.
2. The C register is loaded with byte count value and A is cleared.
3. The content of A is stored to memory, pointed by the HL pair.
4. The memory pointer is incremented to the next location of the memory.
5. The byte counter, C register is decremented by one.
6. Until the byte counter value is 0, go to (3).
   Else the content of the registers are restored by popping.
7. Return.

2.3.3 Routine 'mvtrgt'

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

1. The registers used in the routine are pushed in the stack.
2. The HL pair is pointed to the last location of the generated string in memory.
3. The D register is loaded with 1000000b.
4. The content of the memory is moved to A.
5. The D register is ANDed with A and the result is moved to B.
6. The memory pointer is moved to the last location of the target string in memory.
7. The content of B register is ORed with memory content and the result is stored in memory.
8. The registers are popped.
9. Return.

2.3.4 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.

1. The registers used in the routine are pushed in the stack.
2. The HL register pair, used as memory pointer is set to the last location of the memory where the string is stored.
3. The register B, byte counter is loaded with count value.
4. The register D is loaded with the byte, 01000000b.
5. The register C is loaded with 07h, the number of operation for a byte.
6. The content of the memory is ANDed with D, the result is shifted left by one bit and is stored in E.
7. The content of D is rotated left by one bit.
8. The content of memory is moved to A and the content of D is ANDed with A.
9. The content of E is XORed with A.
10. The zero flag is checked, if it is set, go to (13).
    Else, the content of memory is moved to A, the D is ORed with A and the result is sent to memory.
11. The D register is rotated right by 2 bits.
12. The C is decremented, till it is zero go to (6). Else, go to (16).
13. The unmasked bit position in memory as determined by D is reset to 0.

14. The masking byte in D is restored and rotated right by 2 bits.
15. The C, operation counter is decremented, till it is zero go to (6).
16. The 8th 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.
17. The register B, byte counter is decremented by one.
18. Till the byte counter is 0, go to (4).
    Else, the registers are restored by pop operation.
19. Return.

2.3.5 Main Program

The algorithm of the main program which calls the other routines described above applies the triangular encoding operation.

1. The stack pointer is initialized.
2. Routine 'clr' is called.
3. The registerB is initialized with 10h for 2 byte data.
4. The 'mvtrgt' is called and then the routine 'rotat'.
5. The counter B is decremented.
6. The routine 'XOR' is called.
7. The 'mvtrgt' is called and then the routine 'rotat'.
8. The counter B is decremented.
9. Till the B is zero, go to (6).
10. End.

To visualize the frequency distribution of characters of the source file and the encoded file, a frequency distribution graph has been plotted in fig 2. The blue lines show the characters in the source while the red lines show the characters in the encoded file.

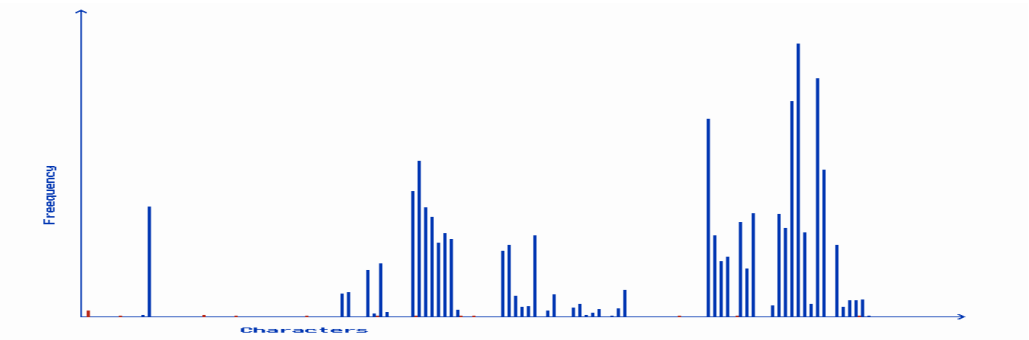


Fig 2: Frequency Distribution of characters in source message and encrypted message under Triangular Encoding

The Chi-square test has also been performed for source file and encoded file. Table 2 shows the values of Chi-square for different bit stream length. In this case also the Chi-square is highly significant at 1 % level of significant

Table 2: Chi-square value of Triangular Encoder for different bit length

Triangular Encoder									
Sl no	Source File	Source File size	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	

Triangular Encoder									
Sl no	Source File	Source File size	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
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	3234.642	3138.836	3080.545	3022.29	**3313.21**

3. Recursive Pair Parity Encoding (RPPE)

This is another method of Encoding. It is symmetric and block cipher encoding in connection with the encryption.

Considering a k-bit string is passed through the Recursive Pair Parity (RPP) encoder, which encodes a string of same length at its output. Let X be the string of k-bit. It is supplied as an input to the RPP Encoder. The encoder will generate a string X^1 of k-bit at the output. This is the first cycle of encoding. If the generated string is allowed to pass to the input of the encoder again, then the encoder will generate a string X^2. This is called the 2^{nd} cycle and so on. The process is repeated and checked each time at the output, whether the output is same as the string supplied initially (i.e. X) or not. It is assumed that the original string is generated after i cycles. Then the intermediately generated one of (i-1) strings can be used as encoded string. Consider that after m (m<i) cycles the generated string is used as encoded string. The original string X can be decoded by applying (i-m) cycles on the encoded string.

3.1 Principle of RPP Encoding

To describe the principle of Encoding, a binary string, X^0 with the bit length of n, is considered. The source string represented by subscript 0 i.e.

$$X^0 = x_1^0\ x_2^0\ x_3^0\ x_4^0 \ldots\ldots\ x_n^0$$

where x_1^0 is the most significant bit and x_n^0 the least significant bit of the string.
After the application of 1^{st} cycle of RPP Encoding operation, the string generated is

$$X^1 = x_1^1\ x_2^1\ x_3^1\ x_4^1 \ldots\ldots\ x_n^1$$

using the rules given below.

$$x_1^1 = x_1^0$$
$$x_2^1 = x_1^1 \text{ XOR } x_2^0 = x_1^0 \text{ XOR } x_2^0$$
$$x_3^1 = x_2^1 \text{ XOR } x_3^0 = x_1^0 \text{ XOR } x_2^0 \text{ XOR } x_3^0$$
$$\ldots$$
$$x_n^1 = x_2^1 \text{ XOR } x_3^0 = x_1^0 \text{ XOR } x_2^0 \text{ XOR } x_3^0 \text{ XOR } x_4^0 \text{ XOR } \ldots\ldots\ldots \text{ XOR } x_n^0$$

If the same operation is repeated k times, the string generated after k^{th} iteration as $X^k = (x_1^k\ x_2^k\ x_3^k \ldots\ldots\ldots\ x_n^k)$ can be obtained using rules given below.

$$x_1^k = x_1^{k-1} = x_1^0$$
$$x_2^k = x_1^k \text{ XOR } x_2^{k-1} = x_1^{k-1} \text{ XOR } x_2^{k-1}$$
$$x_3^k = x_2^k \text{ XOR } x_3^{k-1} = x_1^{k-1} \text{ XOR } x_2^{k-1} \text{ XOR } x_3^{k-1}$$
$$\ldots$$
$$x_n^k = x_{n-1}^k \text{ XOR } x_n^{k-1} = x_1^{k-1} \text{ XOR } x_2^{k-1} \text{ XOR } x_3^{k-1} \text{ XOR } x_4^{k-1} \ldots\ldots\ldots \text{ XOR } x_n^{k-1}$$

So any bit generated in the transformation is the result of XORing between the bit position in the previous transformation and parity generated up to the previous bit. Hence is termed as Pair Parity Encoder. The same transformation is applied recursively on the string, hence it is termed Recursive Pair Parity (RPP) Encoder.

3.2 Realization of Recursive Pair Parity Encoder

The encoder applies on a 256 (say) bit block by calling routines developed given in section 3.2.1 to 3.2.5. There are four routines (**ctr_clr, sav_data, xor_data, ctr_inr**) used by the main program of RPPE. The algorithms of the routines and the main program are given in 3.2.1 to 3.2.5 for a block of 256 bit.

3.2.1 Routine ctr_clr

This routine clears the area (F900h & F920h) of the memory used as counter for storing the number of transformation required to reappear the original string.

1. The Processor Status Word (PSW), BC, DE and HL register pairs are saved in the stack.
2. The HL pair is pointed to the location(F950h) of the memory.
3. The register D, used as counter is loaded with 20h for clearing 32 bytes.
4. The register A is cleared.
5. The content of A is moved to the Memory.
6. The memory pointer is incremented.
7. The counter D is decremented.
8. Until the content of counter is zero, go to (5).
   Else, the register pairs and PSW are popped.
9. Return.

3.2.2 Routine sav_data

This routine saves the data from F900h onwards to the memory area, pointed by memory pointer (HL pair).

1. The Processor Status Word (PSW), BC, DE and HL register pairs are
2. saved in the stack.
3. The register D, used as byte counter is loaded with 20h **for 256 bit string**.
4. The BC pair, used as memory pointer, is pointed to the location(F900h) of the memory.
5. The content of the memory is taken to the register A.
6. The content of A is moved to that of memory.
7. The memory pointer is incremented.
8. The byte counter is decremented.
9. Until it is zero, go to (4).
   Else, the register pairs and PSW are popped.
10. Return.

3.2.3 Routine xor_data

This is the important routine which XORs the bits of the string according to the principle of RPP operation. The content of B will be taken as parameter.

1. The registers is pushed in stack.
2. The HL pair is pointed to the last location of the string.
3. The masking register D is loaded with 10000000b
4. The counting register C is loaded with 07h.
5. The most significant bit of the memory is taken to A, is rotated right by one bit and stored in E.
6. The content of D is rotated right by one bit.
7. The next to ms bit is taken to A and is XORed with E.

8. If the result is zero, go to (12).
   Else, next to ms bit is set.
9. Are all the operations, as supplied through B, over ? if yes, go to (23).
10. The C is decremented.
11. Till it is zero, go to (5).
    Else, go to (16).
12. The next to ms bit is set to zero.
13. Are all the operations, as supplied through B, over ? if yes, go to (23).
    Else, the mask byte is restored.
14. The C is decremented.
15. Till it is zero, go to (5).
16. The D is loaded with 00000001b.
17. The ls bit is collected, shifted right by one bit and stored to E.
18. The ms bit of previous location is collected in A
19. It is XORed with E.
20. If it is zero, go to (21).
    Else, the bit concern is set and go to (22).
21. The bit concern is reset.
22. If all the operations, as supplied through B, are not over, go to (3).
23. The pushed registers are popped up.
24. Return.

3.2.4 Routine ctr_inr

This routine increments the memory counter, formed by the memory locations (F900h & F901h), by one.

1. The registers is pushed in stack.
2. The HL pair is pointed to F94Fh
3. The HL pair is incremented.
4. The content of the memory is taken to A and checked whether it is full or not.
5. If it is full, go to (3).
   Else, the content of the memory is incremented.
6. The pushed registers are popped up.
7. Return.

3.2.5 Main Program of RPP

The algorithm of main program calling the routines developed for RPPE is given below.

1. The stack pointer is initialized.
2. Call ctr_clr.
3. Counter C is initialized with 20h for 256 bit string.
4. HL pair pointer is initialized to initial address FA00h for storing.
5. Call sav_data.
6. Counter B is initialized with FFh for 255 operations.
7. Call xor_rpp
8. Call ctr_inr.
9. Call sav_data.
10. The counter C is decremented.
11. Until the content of C is exhausted, go to (6).
12. Stop.

To visualize the frequency distribution of characters of the source file and the encoded file, a frequency distribution graph has been plotted in fig 3. The blue lines show the characters in the source while the red lines show the characters in the encoded file.

The Chi-square test has also been performed for source file and encoded file. Table 3 shows the values of Chi-square for different bit stream length. In this case also the Chi-square is highly significant at 1 % level of significant.

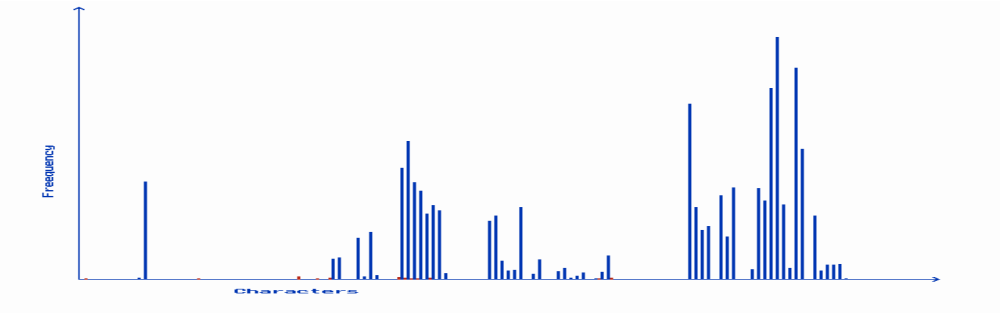


Fig 3: Frequency Distribution of characters in source message and encoded message under RPP Encoding

Table 3: Chi-square value of RPP Encoder for different bit length

RPP Encoder									
Sl no	Source File	Source File size	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	1156.13	1124.11	1109.2	1062.39	1049.01	1035.82	
2	b.abc	1061	1347.59	1320.54	1268.47	1260.58	1254.71	1247.69	
3	c.abc	907	1176.68	1140.3	1063.15	1047.21	1031.85	1030.67	
4	d.abc	2841	3731.23	3639.42	3575.04	3547.86	3502.24	3416.07	
5	e.abc	1765	2318.8	2292.7	2291.52	2217.45	2252.88	2204.06	
6	f.abc	2227	3033.61	3019.67	2976.92	2968.62	2960.28	2847.83	
7	g.abc	2157	2798.8	2748.02	2663.32	2588.36	2594.71	2560.16	
8	h.abc	7121	6655.57	9201.07	8894.04	8721.55	8602.96	8516.29	
9	i.abc	8830	11664.55	11310.32	11037.66	10803.58	10680.82	10501.99	
10	j.abc	2182	2921.47	2890.65	2838.21	2813.67	2799.74	2739.33	
Average		2999.5	3680.44	3868.68	3771.753	3703.127	3672.92	3609.991	3717.82

4. Rotational Encoding

In this encoding the bit stream considered of length, n is rotated placing in a ring. The bit length may very from 8 bit to 256 bit. The rotated string may be considered as encoded string. The decoding operation will depend upon the no of block length and the the number of rotation given at the time of encoding.

4.1 Principle of Rotational Encoding

Considering a very simplest case with a byte, A(a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8) string is allowed to rotate anticlockwise by one bit.

After the first rotation by one bit, lsb has taken the msb position and all other bits are shifted right by one bit. So 8 such rotations are required to get back the original string for an 8 bit string and one of the seven intermediate strings can be used as encoded string.

If the string generated after 2^{nd} rotation is used as encoded string, then (8 –2) = 6 more rotations are to applied on the encoded string to get back the original string.

The principle can be extended to n byte string. The number of rotation required to get back the original string for n byte string (m) = n x 8, where n is the number of bytes in the string.

The total number of intermediately generated string, (k) = (n x 8 –1)

For n = 1, k = 7.

Considering that after i-th rotation, the generated string is used as encoded string. Then the number of rotations (l) to be applied on the encoded string at the time of decoding = n x 8 – i.

For n = 1 and i = 2 , then l = 6.

When a large number of bytes are taken into consideration in the string, the rotational encoding will not be very effective. On 8^{th} rotation, the LS byte will go to the MS byte position and all other bytes will be moved to the right. The characters in the string will appear again in the shifted condition and LS byte character will come to the MS byte position. On 16^{th} rotation the same thing will happen. So after 8 and its multiple rotation the part of the message will reappear with cut and paste condition. This is the disadvantage with the rotational encoding.

On rotational encoding a modification is suggested here with a view to eliminate the disadvantage with the rotational encoding.

Before applying the rotational encoding, a particular bit (say, lsb) of each byte of the string under consideration is complemented. This additional feature is very effective and will eliminate the disadvantage of reappearing the bytes after 8 and its multiple rotations.

This will also be very effective for any number of bytes. The encoding with large number of bytes with a particular bit inverted will be more effective. The complexity will be high with large number of bits in the string.

4.2 Encoding Procedure

The Modified Rotational Encoding (MRE) needs two steps to encode the string..

1. Inversion of a bit in each byte of the string.
2. Rotational encoding: The number of rotation will be applied to the string under consideration, which is less than the number of rotations required to get back the original string.

4.3 Decoding Procedure

Two following steps in succession are required to decode the encoded string.

1. Rotational Decoding: The rest number of rotation will be applied to the encoded string under consideration to complete the number of rotations required to get back the original string.
2. Inversion of the bit in each byte of the string.

4.4 Realization of Modified Rotational Encoder

The microprocessor, as specified in the first chapter, has been used for realizing the Modified Rotational Encoder. To realize the encoder, three following routines are required. The main routine is calling the three routines (**lsbinv, rot, store**). The algorithms of routines are given from section 4.4.1 to 4.4.4.

4.4.1 Routine lsbinv

This routine has used HL pair as memory pointer and C register as counter, representing the number of bytes for which the ls bit will be inverted.

1. The BC pair, HL pair, DE pair is pushed in the stack.
2. The HL pair, used as memory pointer is pointed to f900h
3. The register C is loaded with the count value.
4. Register B is initialized with the masking byte (00000001b).
5. The memory content, pointed by HL, is tested for its lsb whether 0 or 1.
6. If it were zero, go to (7), for setting it 1.
   Else, reset it to 0 and go to (8).
7. The ls bit is set to 1
8. The memory pointer is incremented
9. The counter C is decremented.
10. Until the counter value is zero, go to (5).
11. The registers are popped back.
12. Return.

By changing the counter value, the bit length of the string can be changed. Here, only the lsb has considered, to be inverted. However any bit in the bytes of the string can be inverted by changing the masking byte in register B.

4.4.2 Routine rot

This routine rotates the string anticlockwise by one bit, containing n bytes. It is assumed that the string is stored from f900h onwards, LSB in f900h. The ls bit of the string stored in f900h is checked for 0 or 1 and set the carry accordingly. Then the memory pointer is set to the last location, containing the MSB, and rotate the byte right by one bit through carry. The process is continued until the first location, containing the LSB, is reached. The steps are

1. Register C is initialized as counter
2. HL pair, used as memory pointer, is set to F900h
3. The memory content is moved to A.
4. 01h is ANDed with A.
5. If the zero flag is set, register B is loaded with 00h, otherwise with 01h.
6. The memory pointer is set to the last location.
7. The ls bit in B is shifted to carry bit.
8. The memory content is rotated through carry.
9. The pointer is decremented.
10. The byte counter, C is decremented.
11. Until the counter is exhausted, go to (8).
12. Return.

By changing the count value in C, the bit length in String can be changed.

4.4.3 Routine store

This routine is used for storing the string as well as the intermediate string generated from F900h onwards during encoding or decoding. Here the HL pair is used the pointer of the memory from where the bytes will stored. The initialization of the HL pair is made through the main program and will be used as parameter to the routine ‘store’.

1. The BC and DE pair is saved in the stack.
2. The D is initialized with byte counter.
3. The BC pair is initialized with f900h.
4. The content of memory pointed by BC pair is moved to A.
5. The content of A is moved to the memory pointed by HL pair.
6. The HL and BC pair are incremented.

7. The D, byte counter is decremented.
8. Till it is zero, go to (4).
9. BC and DE pairs are incremented.
10. Returned.

By changing the counter value in D, the byte length can be changed.

4.4.4 Main Program

The algorithm of main program is as follows.

1. The stack pointer is initialized.
2. The HL pair is initialized with fa00h.
3. The routine 'store' is called.
4. The routine 'lsbinv' is called.
5. The routine 'store' is called.
6. The counter C is loaded with the value, the number of rotations to be given
a. during the encoding.
7. The routine 'rot' is called.
8. The counter is decremented.
9. Till it is zero, go to (7).
10. The routine 'lsbinv' is called.
11. The routine 'store' is called.
12. End.

4.5 Encoding of MRE

Here, the routine 'lsbinv' is called once and routine 'rot' is called j, the number of rotations to be given during the encoding, times. The output generated at the memory where the string was supplied.

1. The stack pointer is initialized at ffa0h.
2. The routine 'lsbinv' is called.
3. The counter C is loaded with the value, the number of rotations to be given during the encoding.
4. The routine 'rot' is called.
5. The counter is decremented.
6. Till it is zero, go to (4).
7. End.

4.6 Decoding of MRE

Here, the routine 'rot' is called (8 n – j), the number of rotations to be given during the decoding, times and routine 'lsbinv' is called once.

1) The stack pointer is initialized at ffa0h.
2) The counter C is loaded with the value, the number of rotations to be given during the decoding
3) The routine 'rot' is called.
4) The counter is decremented.
5) Till it is zero, go to (3).
6) The routine 'lsbinv' is called.
7) End.

To visualize the frequency distribution of characters of the source file and the encoded file, a frequency distribution graph has been plotted in fig 4. The blue lines show the characters in the source while the red lines show the characters in the encoded file.

The Chi-square test has also been performed for source file and encoded file. Table 4 shows the values of Chi-square for different bit stream length. In this case also the Chi-square is highly significant at 1 % level of significant.

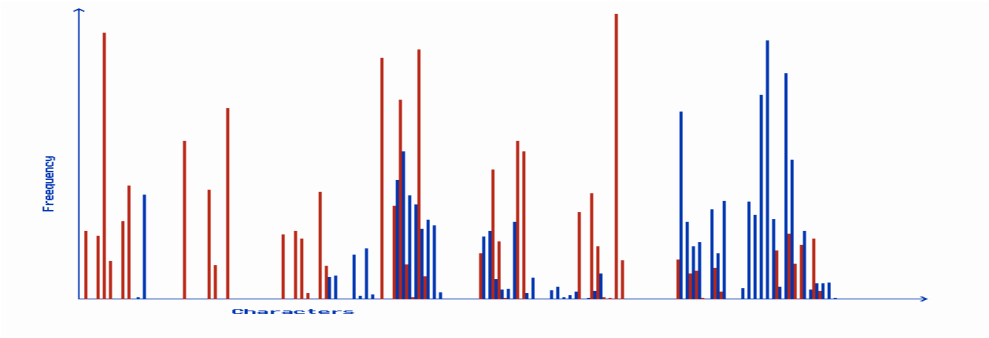


Fig 4: Frequency Distribution of characters in source message and encrypted message under Rotational Encoding

Table 4 : Chi-square value of Rotational Encoder for different bit length

Rotational Encoder									
Sl no	Source File	Source File size	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	1109.35	1316.06	1136.07	1284.01	1098.57	1086.26	
2	b.abc	1061	1471.86	1615.00	1441.31	1605.69	1435.07	1308.72	
3	c.abc	907	1258.83	1370.76	1237.29	1361.99	1248.86	1091.92	
4	d.abc	2841	3515.57	4142.43	3897.72	4147.48	3601.20	3407.21	
5	e.abc	1765	2294.55	2602.47	2455.67	2638.70	2317.90	2194.66	
6	f.abc	2227	2887.01	3339.16	3140.01	3376.64	3005.89	2769.38	
7	g.abc	2157	2760.56	3110.47	3000.47	3097.51	2842.84	2596.32	
8	h.abc	7121	8930.41	10587.39	9835.08	10544.42	9071.24	8496.01	
9	i.abc	8830	10944.10	13039.84	12041.59	13025.41	11049.97	10651.96	
10	j.abc	2182	2846.57	3288.61	3048.81	3316.18	2910.09	2670.99	
Average		2999.5	3801.88	4441.22	4123.40	4439.80	3858.16	3627.34	4048.63

5. Modified Johnson Encoding (MJE)

The Johnson encoding is based on Johnson counter. This counter considers a block of n bits and is subjected to rotation through left or right. While rotating the block to left (say), the msb is inverted only and takes the position of lsb and all other bits are shifted to the left by one bit. If we call this as one rotation, the number of rotation required to get back the original block is 2n. A modified Johnson encoding is suggested.

In this modified Johnson encoding, a particular bit of each byte within the block of 256 bit is inverted and then the Johnson encoding is applied. This is a generalized technique. The block length can be increased to any length.

5.1 Modified Johnson Encoding for N byte string

The string consisting of N bytes is considered for Modified Johnson Encoding. The following steps are followed for the encoding.

1. Inversion of lsb of each byte of the string
2. Johnson Encoding on the string

Both these steps are followed during Encoding as well as Decoding. Step (1) followed step (2) is used at the time of Encoding, while step (2) followed by step (1) at the time of Decoding.

The step (1) is a very simple technique, as discussed in the earlier section. Also it is known that the total number of Johnson Encoding for n byte string required to get back the string is twice the number of bits in the string i.e. (2 x 8 x N). Any one of (2 x 8 x n – 1) can be used as encoded string.

So at the time of Encoding, let us assume that K number of Johnson encoding operations are applied, where k is less than (2 x 8 x N – 1).

At the time of Decoding, (2 x 8 x N – 1 - K) number of operations will be applied.

5.2 Realization of Modified Johnson Encoder using Microprocessor Based System

To realize the Modified Johnson Encoder using Microprocessor Based System, 3 following routines (lsbinv, rotj, store) are developed. The routines 'lsbinv' and 'store' are the same as the routines used in Rotational Encoding. So the discussion of the algorithms of these routines is not given for clarity. The routine 'rotj' differs slightly from the routine 'rot' in rotational encoder. In Rotational Encoding, the lsb takes the position of msb and other bits in the string moves one bit to the right, while in Johnson Encoding, the lsb is inverted first and takes the position of msb and other bits in the string moves one bit to the right. The algorithm of routine rotj is given in 5.2.1 and that of main program is given in 5.2.2.

5.2.1 Routine rotj

This routine rotates the string anticlockwise by one bit, containing n bytes. It is assumed that the string is stored from f900h onwards, LSB in f900h. The ls bit of the string stored in f900h is checked for 0 or 1 and set the carry to its inverted bit accordingly. Then, the memory pointer is set to the last location, containing the MSB, and rotated the byte right by one bit through carry. The process is continued till the first location, containing the LSB, is reached. The steps are

1. Register C is initialized as counter
2. HL pair, used as memory pointer, is set to F900h
3. The memory content is moved to A.
4. 01h is ANDed with A.
5. If the zero flag is set, register B is loaded with 01h, otherwise with 00h.
6. The memory pointer is set to the last location.
7. The ls bit in B is shifted to carry bit.
8. The memory content is rotated through carry.
9. The pointer is decremented.
10. The byte counter, C is decremented.
11. Until the counter is exhausted, go to (8).
12. Return.

By changing the count value in C, the bit length in String can be changed.

5.2.2 Main Program of MJE

The algorithm of main program is given below.

1. The stack pointer is initialized at ffa0h.
2. The HL pair is initialized with fa00h.
3. The routine 'store' is called.
4. The routine 'lsbinv' is called.
5. The routine 'store' is called.

6. The counter C is loaded with the value, the number of rotations to be given during the encoding.
7. The routine ‘rotj’ is called.
8. The counter is decremented.
9. Until it is zero, go to (7).
10. The routine ‘lsbinv’ is called.
11. The routine ‘store’ is called.
12. End.

To visualize the frequency distribution of characters of the source file and the encoded file, a frequency distribution graph has been plotted in fig 5. The blue lines show the characters in the source while the red lines show the characters in the encoded file.

The Chi-square test has also been performed for source file and encoded file. Table 5 shows the values of Chi-square for different bit stream length. Hence the Chi-square is highly significant at 1 % level of significant

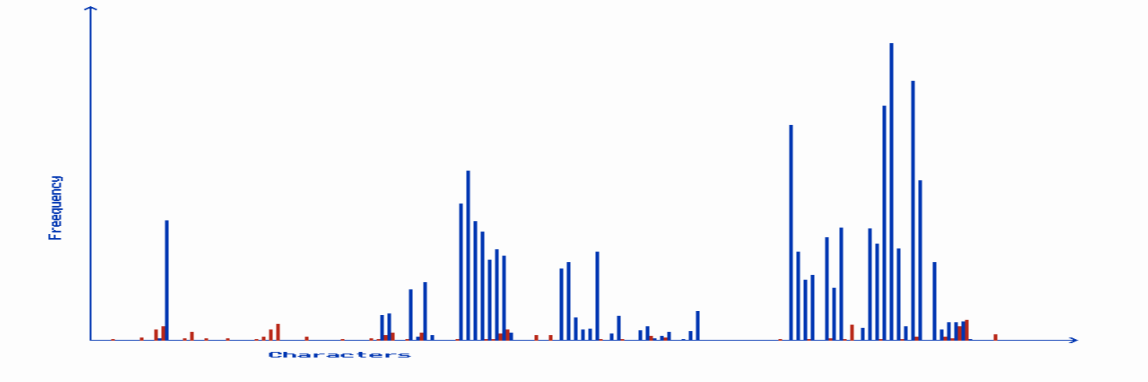


Fig 5: Frequency Distribution of characters in source message and encrypted message under Johnson Encoding

Table 5: Chi-square value of Johnson Encoder for different bit length

Johnson Encoder									
Sl no	Source File	Source File size	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 (overall)
1	a.abc	904	1120.88	1182.92	1214.89	1242.94	1268.80	1273.19	
2	b.abc	1061	1428.69	1535.73	1583.92	1590.85	1597.17	1607.06	
3	c.abc	907	1226.82	1312.53	1347.78	1355.73	1369.26	1367.45	
4	d.abc	2841	3748.21	3860.95	4011.11	4096.90	4109.80	4123.25	
5	e.abc	1765	2324.76	2458.00	2527.23	2588.38	2600.17	2617.36	
6	f.abc	2227	2876.52	3095.78	3228.18	3310.49	3349.98	3354.83	
7	g.abc	2157	2816.41	2920.78	2998.06	3068.83	3076.86	3113.73	
8	h.abc	7121	9307.56	9868.90	10170.16	10365.23	10478.05	10531.60	
9	i.abc	8830	11687.59	1143.22	12600.84	12894.00	12986.56	12993.66	
10	j.abc	2182	2850.30	3051.06	3176.02	3247.08	3300.38	3309.84	
Average		2999.5	3938.77	3042.99	4285.82	4376.04	4413.70	4429.20	4081.09

6. Bit Swap Encoding (BSE)

A stream of input string (binary) is divided into blocks which may vary from 1 to n. on each block a bit swapping transformation will be applied. The transformed message will be an unintelligible form. It is a kind of block cipher method and symmetric in nature. The technique has been tested and verified up to 256 bit string using a microprocessor based system.

6.1 Principle of Bit Swap Encoding

The total message can be considered as blocks of bits with different size like 2/ 4/ 8/ 16/ 32/ 64/ 128/ 256. The bit swapping can be applied to each block separately. The principle of bit swapping is discussed in following for different block size.

a) Swapping on 2 bit:

Considering a block (X) with two binary bits a and b

$X = b\ a$

$X_1 = a\ b$ after 1st swapping

$X_2 = b\ a$ after 2nd swapping

X_1 is the first swapping and X_2 is the second swapping. It takes two swapping to get back the original block.

b) Swapping on 4 bit:

Considering a block (X) with four binary bits, we have

$X = d\ c\ b\ a$

$X_1 = c\ d\ a\ b$ after 2 bit swapping on X

$X_2 = a\ b\ c\ d$ after 4 bit swapping on X_1

$X_3 = b\ a\ d\ c$ after 2 bit swapping on X_2

$X\ 4 = d\ c\ b\ a$ after 4 bit swapping on X_3

It is mentioned that while dealing with 4 bit block, first 2 bit swapping and then 4 bit swapping are applied on X generating X_1 and X_2. The same process is applied on X_2 generating X_3 and X_4. The block (X_4) generated is the original block.

c) Swapping on 8 bit:

Considering a block (X) with eight binary bits, we have

$X = h\ g\ f\ e\ d\ c\ b\ a$

$X\ 1 = g\ h\ e\ f\ c\ d\ a\ b$ after 2 bit swapping on X

$X\ 2 = e\ f\ g\ h\ a\ b\ c\ d$ after 4 bit swapping on X_1

$X\ 3 = a\ b\ c\ d\ e\ f\ g\ h$ after 8 bit swapping on X_2

$X\ 4 = b\ a\ d\ c\ f\ e\ h\ g$ after 2 bit swapping on X_3

$X\ 5 = d\ c\ b\ a\ h\ g\ f\ e$ after 4 bit swapping on X_4

$X\ 6 = h\ g\ f\ e\ d\ c\ b\ a$ after 8 bit swapping on X_5

2 bit, 4 bit and 8 bit swapping are applied on the block and the same process is followed to get back the original block.

Swapping on 16, 32, 64, 128 and 256 follows the same principle as described above. The number of swapping required to get back the original string is as follows.

The number of swapping required is 2 for 2 (2^1) bit swapping, 4 for 4 (2^2) bit swapping, 6 for 8 (2^3) bit swapping, 8 for 16 (2^4) bit swapping and By induction it can be written as

The number of swapping (m) required to get back the original string for n bit string will be

$m = 2 \log (n) / \log(2)$ where n is an integer.

For 256 bit block swapping, 2, 4, 8, 16, 32, 64, 128 and 256 bit swapping are required and again these swapping are applied to get back the original block. Therefore 16 number of swapping is required to generate the original block for 256 bit block string.

Out of these 16 swapping, any of 15 intermediately generated block can be used as cipher text for security for 256 bit block string. Considering that after 5^{th} swapping, the generated block is used for cipher text, then (16-5) number of rest swapping are to be applied to get back the original block.

Hence the technique can successfully be applied for encryption purpose for security. Here maximum block length is considered as 256, but any bit higher than 256 can be considered without loss of generality. Higher the bit length, better is the security, since attack by the eavesdropper for higher bit length block needs exponential increase in computation.

6.2 Realization of BSE

In order to verify the technique a microprocessor based system has been considered. 256 bit i.e. 32 byte data is stored in the memory on which the bit swapping technique will be applied. 2, 4, 8, 16, 32, 64, 128 and 256 bit swapping routines and a rotation of 256 bit string have been developed. The algorithms are presented here. 2, 4 and 8 bit swapping routines need a special care, while 16, 32, 64, 128 and 256 bit swapping are comparatively easier, since these are basically byte movement only.

6.2.1 Routine for 2 bit swapping

This routine will swap 2 least significant bit (lsb) of a byte stored in a memory (say, F900h).

1. Registers A,B,C are cleared.
2. HL register pair is pointed to the memory F900h.
3. The content of the memory is moved to D register and A register.
4. The lsb of A is collected and stored in B.
5. Next to lsb of A is collected and stored in C.
6. The bit in B is shifted by one bit to the left.
7. The bit in C is inserted to the lsb position of B.
8. The swapped two bit is stored in C.
9. The two lsb of D register is cleared.
10. The content of C is ORed with D and the result is moved to F900h.
11. Return.

The algorithms for 4 and 8 bit swapping are same as that of 2 bit with proper change in steps 4, 5, 6, 8 and 9.

6.2.2 Routine for 64 bit swapping

The algorithms for 16, 32, 128 and 256 bit swapping are same as that of 64 bit swapping. Here the algorithm for 64 bit swapping is presented for clarity.

A string of 64 bit block i.e. 8 byte data is assumed to be stored from F900h onwards. The least significant part is stored in lower address and the most significant part in higher address of the memory.

1. The string stored from F900h onwards is saved to FA00h onwards
2. The most significant part of the saved string is moved to least significant part of F900h area.
3. The least significant part of the saved string is moved to most significant part of F900h area.
4. Return..

6.2.3 Routine of Rotation for 256 bit string

The routine can rotate the string of 256 bit stored from F900h onwards by one bit to the left. The steps of the algorithm are as

1. A register C is loaded with 20h and used as counter.
2. HL register pair, used a pointer, is pointed to the memory F900h.
3. The lsb of the byte stored in F900h is collected and sent to carry flag.

4. The pointer is moved to the last byte of the string.
5. The carry is sent to the most significant bit position and lsb to carry using right rotation.
6. The pointer is decremented by one memory location.
7. The counter is decremented.
8. Until the counter value is zero, go to step 5.
9. Return.

By changing proper parameters, the algorithm may be applicable to any other bit string, multiple of 8.

6.2.4 Routine for 2 bit swapping on 256 bit string

To apply 2 bit swapping on 256 bit string, stored from F900h onwards, the swapping is applied on 2 lsb of F900h, then the whole string of 256 bit is rotated right by 2 bits, again the bit swapping applied and so on, until all the bits are swapped in the string.

1. A register C is loaded with 80h and used as counter.
2. 2 bit swapping is called.
3. Rotation for 256 bit string is called.
4. The counter C is decremented.
5. Until the counter is zero, go to step 2.
6. Return.

The principle of algorithm for 4, 8, 16, 32, 64, 128 and 256 bit swapping on 256 bit string are same that of 2 bit swapping.

6.2.5 Main Program of BSE on 256 bit string

1. The stack pointer is initialized.
2. Routine for 2 bit swapping on 256 bit string is called.
3. Routine for 4 bit swapping on 256 bit string is called.
4. Routine for 8 bit swapping on 256 bit string is called.
5. Routine for 16 bit swapping on 256 bit string is called.
6. Routine for 32 bit swapping on 256 bit string is called.
7. Routine for 64 bit swapping on 256 bit string is called.
8. Routine for 128 bit swapping on 256 bit string is called.
9. Routine for 256 bit swapping on 256 bit string is called.
10. End.

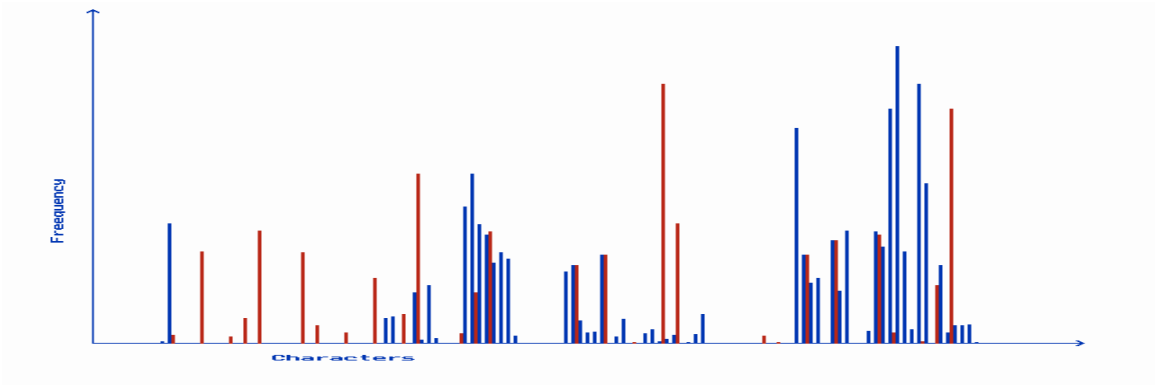


Figure 6: Frequency Distribution of characters in source message and encoded message for Bit Swap Encoding (prt.txt & prt.jen)

To visualize the frequency distribution of characters of the source file and the encoded file, a frequency distribution graph has been plotted in fig 6. The blue lines show the characters in the source while the red lines show the characters in the encoded file.

The Chi-square test has also been performed for source file and encoded file. Table 6 shows the values of Chi-square for 64-bit stream length. Hence the Chi-square is highly significant at 1 % level of significant.

Table 6: Value of Chi-square for 64 bit block lengths in Bit Swap Encoding

	Source File (size in bytes)	Encoded File	No of operation	Value of chi-square
1	A0.abc (918)	A64.abc (918)	8	1232.17
2	B0.abc (563)	B64.abc (563)	8	711.65
3	C0.abc(1987)	C64.abc(1987)	8	2862.32
4	D0.abc(1733)	D64.abc(1733)	8	2503.53
5	E0.abc(1659)	E64.abc(1659)	8	2150.60
6	F0.abc(2073)	F64.abc(2073)	8	2689.39
7	G0.abc(1549)	G64.abc(1549)	8	2040.13
8	H0.abc(1909)	H64.abc(1909)	8	2464.68
9	I0.abc(1830)	I64.abc(1830)	8	2358.44
10	J0.abc(1335)	J64.abc(1335)	8	1788.38
				2080.13

7. Cascaded Encoders

7.1 Introduction

The principle of encoders are stated and verified through a microprocessor-based system. These are Prime Position Encoding (PPE), Triangular Encoding (TE), Recursive Pair Parity Encoding (RPPE), Modified Rotational Encoding (MRE), Modified Johnson Encoding (MJE) and Bit Swap Encoding (BSE).

The four encoders (PPE, TE, RPPE and BSE) are symmetric in nature, while 4^{th} and 5^{th} (MRE and MJE) are asymmetric, with reference to the encryption of a message. An encoder will be designated as symmetric, when the encoding process is applied repeatedly on a string, generates the original string. The 4^{th} and 5^{th} have lost its symmetric character, the least significant bit of each byte in the string has been inverted before the transformation is applied. If this operation (lsb inversion) is excluded, then rotational and Johnson encoding as presented in the previous chapters are basically symmetric encoders. So the last two encoders have been transformed into the cascaded encoders due to adding of lsb inversion to overcome the shortcomings associated to.

7.2 Cascading of Encoders

It is proposed that any two encoders, out of these six encoders, can be cascaded, one encoders followed by other, to have a cascaded encoder. There are 20 encoders possible as shown in the table 7.

Here, the encoders (4) and (5) are considered without lsb inversion and hence, symmetric. The encoder (1,2) indicates the encoder (1) followed by encoder (2) i.e. the output of the encoder (1) is connected to input of the encoder (2). The string is supplied as input to the encoder (1) and available at the output of the encoder (2).

Table 7: Possible Cascaded Encoders

Number	Cascaded Encoder	Number	Cascaded Encoder
1	(1,2)	16	(2,1)
2	(1,3)	17	(3,1)
3	(1,4)	18	(4,1)
4	(1,5)	19	(5,1)
5	(1,6)	20	(6,1)
6	(2,3)	21	(3,2)
7	(2,4)	22	(4,2)
8	(2,5)	23	(5,2)
9	(2,6)	24	(6,2)
10	(3,4)	25	(4,3)
11	(3,5)	26	(5,3)
12	(3,6)	27	(6,3)
13	(4,5)	28	(5,4)
14	(4,6)	29	(6,4)
15	(5,6)	30	(6,5)

The encoded string is available from the output of second encoder. It is to mention that each of the encoders are symmetric, but the cascaded encoder as whole as a whole becomes asymmetric. The same encoding process under cascaded condition applying on a string, will not decode the original string. Hence, a separate encoding and decoding process are required.

a) **Encoding:**

It is assumed that encoder (1) and encoder (2) need N_1 and N_2 operations respectively to get back a string. And it is also assumed that N_3 operation is applied by encoder (1) and then it is sent to the encoder (2), where N_4 operation is applied and final output is available as the encoded string. N_1 and N_2 are known from the encoders selected and N_3 and N_4 are selected at the time of encoding by the user. N_3 and N_4 should be less than N_1 and N_2 respectively.

b) Decoding:

To decode the encoded string, it is first sent to the encoder (2) and is subjected to $(N_2 – N_4)$ operation and then sent to the encoder (1) for $(N_1 – N_3)$ operations.
The original string will be available from the output of the encoder (1).

8. Application of the Encoders

The encoders as in the previous chapters are designed in order to maintain the security of a message. A message can be considered as a stream of strings concatenated. The string length of 256 bit maximum is considered in the encoders. The message can be split up into the strings of length, considered by the user. To each of the string, any of the encoding process or the cascaded encoding process is applied to convert the same into encoded strings making the message into a cipher text.

The cipher text is sent to the receiving end through telephone line using internet. The message will be recovered by the decoder (may be the same encoder used as decoder or a different one) at the receiving end.

The principles of the encoders are verified by the microprocessor based system. For a user-friendly application in practice, the principles have to be realized in a high level language and to be interfaced with suitable software.

The prospective area where the encoders can be successfully applicable to are as follows.

i) Defence:

The message in defence is highly valuable and needs to be secured in the interest of the country.

ii) Banking / Insurance Sector:

The information in banking / Insurance sector has to be maintained reliable and secured in the interest of the user and the sector itself.

And any other places where the security of the message is the prime importance.

9. Comparative Study of Five Realized Encoders

The five basic encoders are presented in first five chapters. The encoders are discussed with a string of maximum 256 bit on which its operate. A comparative study of these five encoders is presented in table 8 for the number of operations required to get back the string.

The **PP Encoding** is a typical one, the operations required with increase in bit length increase rapidly. There is no direct relationship between the bit length and the operations.

The **Triangular Encoding** shows only two operations, one for encoding and other for decoding, to get back the string. But it has sufficient number of computations inside the operation, which is not reflected in the table.

Table 8: Comparative Study of Encoders

	Operations required to get back the string				
Bit length of the string	PP Encoding	Triangular Encoding	RPP Encoding	Modified Rotational Encoding	Modified Johnson Encoding
16	06	2	16	18	34
32	30	2	32	34	66
64	84	2	64	66	130
128	456	2	128	130	258
256	1820	2	256	258	514

The **RPP Encoding** gives a simple relationship between the operations and the bit length. For detailed study, the respective chapter is referred.

The **Modified Rotational Encoding** is the simplest one. It takes the number of bit length operations plus two lsb inversion to get back the string.

The **Modified Johnson Encoding** can be considered as the enhancement of the Modified Rotational Encoding. The number of operation required is twice the number of bit in the string plus two for lsb inversion.

10. Comparison with RSA

With a view to compare the frequency distribution of characters with encoded file, the same source file is encrypted with existing RSA technique and the frequency distribution of encoded file is compared with the source file. The pictorial representation is given in figure 8.

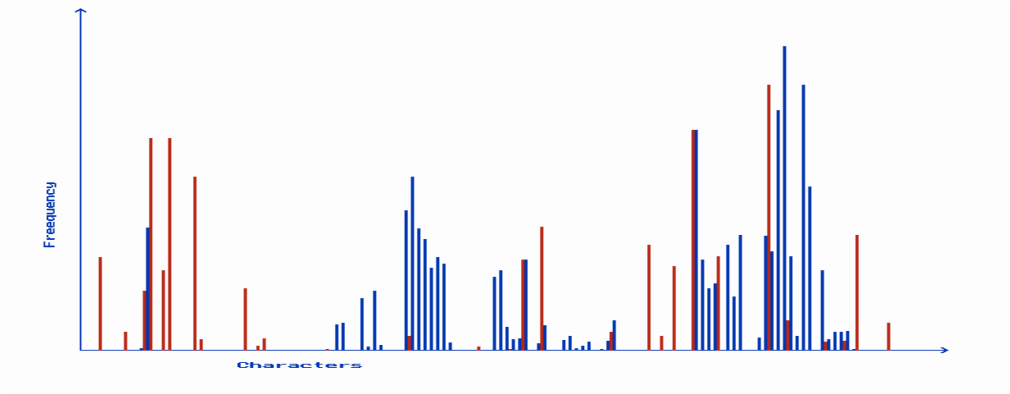


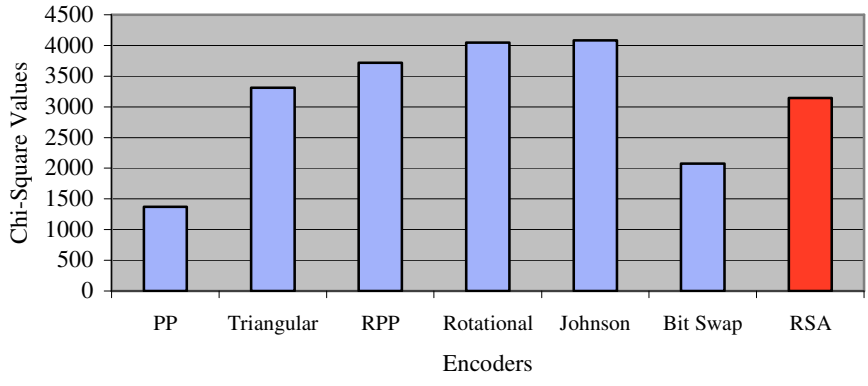
Figure 8: Frequency Distribution of characters in source message and encrypted message for RSA Encoding (prt.txt & prt.jen)

To compare the result with RSA, a ten text files of different file size have been taken. To each file the developed techniques applied to generate the encoded files for different block length. The average chi-square value is computed. RSA encoding technique is also applied to generate the encoded files and corresponding average chi-square value is computed. The values are given in table 9 and corresponding graph is given in figure 9.

Table 9: Comparison of Chi-Square Value among different Techniques

Sl No	Technique	Average Chi-Square Value
1	PP Encoder	1373.86
2	Triangular Encoder	3313.21
3	RPP Encoder	3717.82
4	Rotational Encoder	4048.63
5	Johnson Encoder	4081.09
6	Bit Swap Encoder	2073.22
7	RSA	3144.71

Fig 9: Chi-Square Values for Different Encoders



11. Key Generation

It is evident that the length of key increases the security and the maximum length may be equal to that of the message. Then the suitable method is to generate the random number, which may be used as key each time a message requires to be encrypted and transmitted to the receiver. The receiver with the same key, to be transmitted by the sender, recovers the message from cipher text. The generation of the purely random number is not a very easy task. A good number of literatures are available on the random number generation. These are basically a generation of pseudo-random number.

For a short message we can use anyone of the six encoders to encrypt it by selecting the block length and the number of operations. The algorithms are such that we can extend the block length even equal to the message, though the study of each block length has been restricted to 256 bit. This will give the maximum security. But the computation time will increase at a very high rate and will make impracticable.

Under the condition a proposition is made here to generate the key to be used for the encryption of a message. These three parameters (the selection of encoder, the length of block and the number of operation) are to be sent by the sender through a secured channel and the encrypted message will be sent through insecure channel, which can be accessed by the eavesdroppers. This is shown in the following figure.

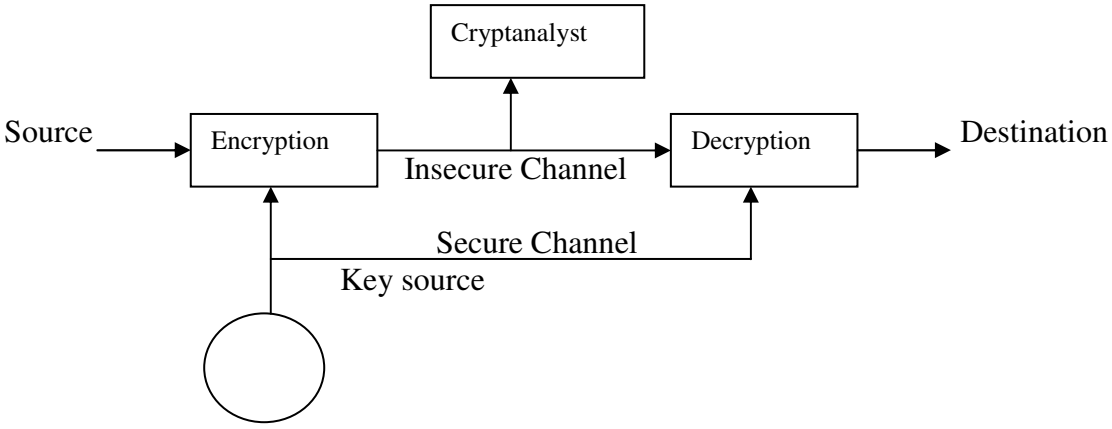


Fig : Model of Symmetric Cryptosystem

It is considering that X is the message and Y, the ciphertext generated by the encryption algorithm and key, K. At the receiving end the message is generated from the ciphertext and the securely transmitted key, K. The ciphertext is accessible to the cryptanalysts who can estimate the message and the key.

Here three parameters are variable: **variable technique, variable block length** and **variable operation** made at the time of encryption. The selection of block length and operation are the numbers only chosen at the sending end each time separately in pseudo-random fashion. But the selection of technique is described in the following.

Out of 6 encoders, two are substitution type and four are of transposition type. Each time the technique selected, two encoders in cascade are selected, one from substitution type and the other from transposition type. The message is subjected to the first encoder and then to the second one. There will be 16 possible techniques.

These three variable components of the key will add sufficient security to the message to be transmitted to the receiver. The receiver will receive the ciphertext through the insecure channel and the key, the components of the key is encrypted in 256 bit length

block using any of the substitution encoder, through the secure channel. At the receiving end the key components will be recovered from the key and the ciphertext will be decrypted generating the message. This kind of key, different in each session of transmission, is called the session key.

In brute-force attack, it will be a difficult task to the cryptanalysts to find the clue of attack in the variable parameter key of 256 bit length.

12. Conclusion

The technique presented here is a simple, takes little time to encode and decode. The encoded string will not generate any overhead bits. It can be easily implemented to any high level language for any practical application for encryption purpose in order to provide a security in message.

13. References

1. Mandal J. K. and Dutta S., "A Space-Efficient Universal Encoder for Secured Transmission", International Conference on Modelling and Simulation (MS' 2000-Egypt), Cairo, April 11-14, 2000, pp-193-201.
2. Mandal J. K. and Dutta S., "A Universal Encryption Technique", Proceedings of the National Conference of Networking of Machines, Microprocessors, IT and HRD-Need of the Nation in the Next Millennium, Kalyani-741 235, Dist. Nadia, West Bengal, India, November 25-26,1999, pp-B114-B120.
3. Mandal J.K. and Dutta S., "A Universal Bit-Level Encryption Technique", Proceedings of the 7th State Science and Technology Congress, Jadavpur University, West Bengal, India, February 28 - March 1, 2000, pp-INFO2.
4. D. K. Bhattacharryya, S. N. N. Pandit, S. Nandi, "A New Enciphering Scheme", Second International Workshop on Telematics, NERIST, India, May'97
5. Arnold G. Reinhold, "Diceware for Passphrase Generation and other Cryptographic Applications", downloaded from Internet
6. D. Welsh , "Codes and Cryptography", Oxford: Claredon Press, 1988
7. J. Seberry and J. Pieprzyk , "An Introduction to Computer Security", Australia: Prentice Hall of Australia, 1989.
8. R.E. Blahut, "Theory and Practice of Error Control Codes" , Addison Wesley Publishing Co., Sydney.
9. D. Boneh, "Twenty Years of Attacks on RSA Cryptosystem" in notices the American Mathematical Society (AMS), vol 46,no 2, 1998, pp 203-213.
10. B. Schneier, "Applied Cryptography", Second Edition, John Wiley & Sons Inc. 1996
11. C. Coupe, P.Nguyen and J. Stern, " The Effectiveness of Lattice Attacks against Low-Exponent RSA", Proceedings of Second International Workshop on Practice and Theory in Public Key Cryptography, PKC'99, vol1560, of lecture notes in Computer Science, Springer-Verlag, 1999, pp 204-218.
12. D. Coppersmith, "Finding a small Root of a Univariate Modular Equation", Proceedings of EUROCRYPT'96,VOL 1070, of lecture notes in Computer Science, Springer-Verlag, 1996, pp 155-165.
13. D. Coppersmith, "Small Solutions to Polynomial Equations and Low Exponent RSA Vulnerabilities", Journal of Cryptology, vol 10, 1997, pp 233-260.
14. M. Wiener, "Cryptanalysis of Short RSA Secret Exponents", IEEE Transactions on Information Theory, vol 36, 1990, pp 553-558.

15. V P Gulati, A Saxena and D Nalla, “On Determination of Efficient Key Pair”, Journal of the Institution of Engineers (India), vol 84, May 2003, pp 1-3.
16. J K Mandal, S Mal and S Dutta, “ Security in E-Business – A Strategic Issue”, Natioal Seminar on Emerging Issues and Strategic Options Before Business in the Liberalised Regime”, 7th March2001, pp 5-6.
17. J K Mandal, S Mal and S Dutta, “Aspects of Storage Efficient Security in GIS Data” Workshop on Remote Sensing and GIS for Sustainable Development and Management in the Himalayas and Adjoining Areas” at NBU, West Bengal by Indian Society of Remote Sensing, Kolkata Chapter, March 8-9, 2002, pp-24.
18. S Mal and J K Mandal, “A Cascaded Technique of Encryption Realised using Microprocessor Based System”, Association for the Advancement of Modelling & Simulation techniques in Enterprises, vol 7, no 2, 2002, pp 25-35.
19. S Mal and J K Mandal and S Dutta, “A Microprocessor Based Encoder for Secured Transmission” Conference on Intelligent Computing on VLSI, Kalyani Govt. Engg. College, 1-17 Feb, 2001, pp 164-169.
20. S Mal and J K Mandal and S Dutta, “A Microprocessor Based Cascaded Technique of Encryption”, Proc. Of XXXVI Annual Convention of CSI-2001, Kolkata, 20-24 November, 2001, pp.C269-275.
21. S Mal and J K Mandal and S Dutta, “A Cryptographic Model for Secured Transmission of Messages”, Proceedings of the National Conference on Applicable Mathematics, WMVC-2001, A.C. College of Commerce, Jalpaiguri, March 17-19,2001, pp-18-21.
22. A Pal, “Microprocessor: Principles and Applications”, TMH,1990.
23. L Leventhal, “Introduction to Microprocessors: Software, Hardware, Programming”, PHI, 1988.
24. R S Gaonkar,” Microprocessor Architecture, Programming, and Applications with the 8085”, Penram International Publishing (India), 4th Edition, 2000.
25. M. Slater, “Microprocessor Design – A Comprehensive Guide to Effective Hardware Design”, PHI, 1999.
26. S Mal and J K Mandal and S Dutta, “A 256 bit Recursive Pair Parity Encoder for Encryption”, Accepted for Publication by AMSE, France.