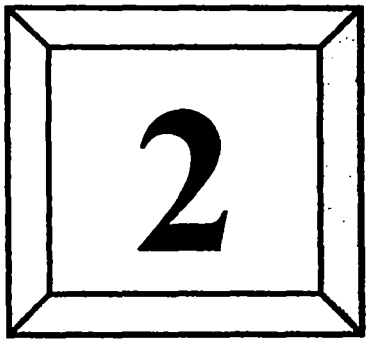

Encryption Through Recursive Positional Substitution Based on Prime-Nonprime (RPSP) of Cluster

	Contents	Pages

2.1 Introduction

The proposed technique in this chapter named, Encryption through Recursive Positional Substitution based on Prime-nonprime of cluster or RPSP, is a secret-key cryptosystem.

In this technique, after decomposing the source stream of bits into a finite number of blocks of finite length, the positions of the bits of each of the blocks is re-oriented using a generating function. For a particular length of block, the block itself is regenerated after a finite number of such iterations. Any of the intermediate blocks during this cycle is considered to be the encrypted block. To decrypt the encrypted block from the ciphertext, the same process is to be followed but the generating function may have to be applied different number of times.

To achieve the security of a satisfactory level, it is proposed that different blocks or blocks should be of different sizes. Accordingly, for different blocks, number of iterations during the encryption and the number of iterations during the decryption also not necessarily should be fixed. This information in a proposed fixed format, described later in this chapter, constitutes the secret key for the system, which is to be transmitted by the sender to the receiver, either with the message or in an isolated manner.

The technique does not cause any storage overhead. It provides a large key space, so that the chance of breaking the ciphertext is almost nullified by any technique of cryptanalysis. The implementation on practical scenario is well proven with positive outcome [3, 52, 55].

Section 2.2 of this chapter describes the scheme of this technique with simple examples. Since the entire scheme is the combination of the encryption and the decryption processes, this section also includes how one part of the scheme can be used for the encryption and how the remaining part can be used for the decryption. Section 2.3 shows a simple implementation of the technique, where a sample text message has been considered for the purpose of transmission using the encryption. Section 2.4 gives the results obtained after implementing the RPSP technique on a number of real-time files of different categories like .exe, .com, .dll, etc. Section 2.5 is an analytical presentation of the technique, where the RPSP technique has been analyzed from different perspectives. Section 2.6 draws a conclusion on the technique.

2.2 The Scheme

The technique considers the plaintext as a stream of finite number of bits N, and is divided into a finite number of blocks, each also containing a finite number of bits n, where 1 <= n <= N. Here orientation of the position of bits within a block is performed through a generating function. If we repeat the operation using the same generating function, the original block is regenerated after a finite number of iterations forming a cycle [38, 42, 46, 52].

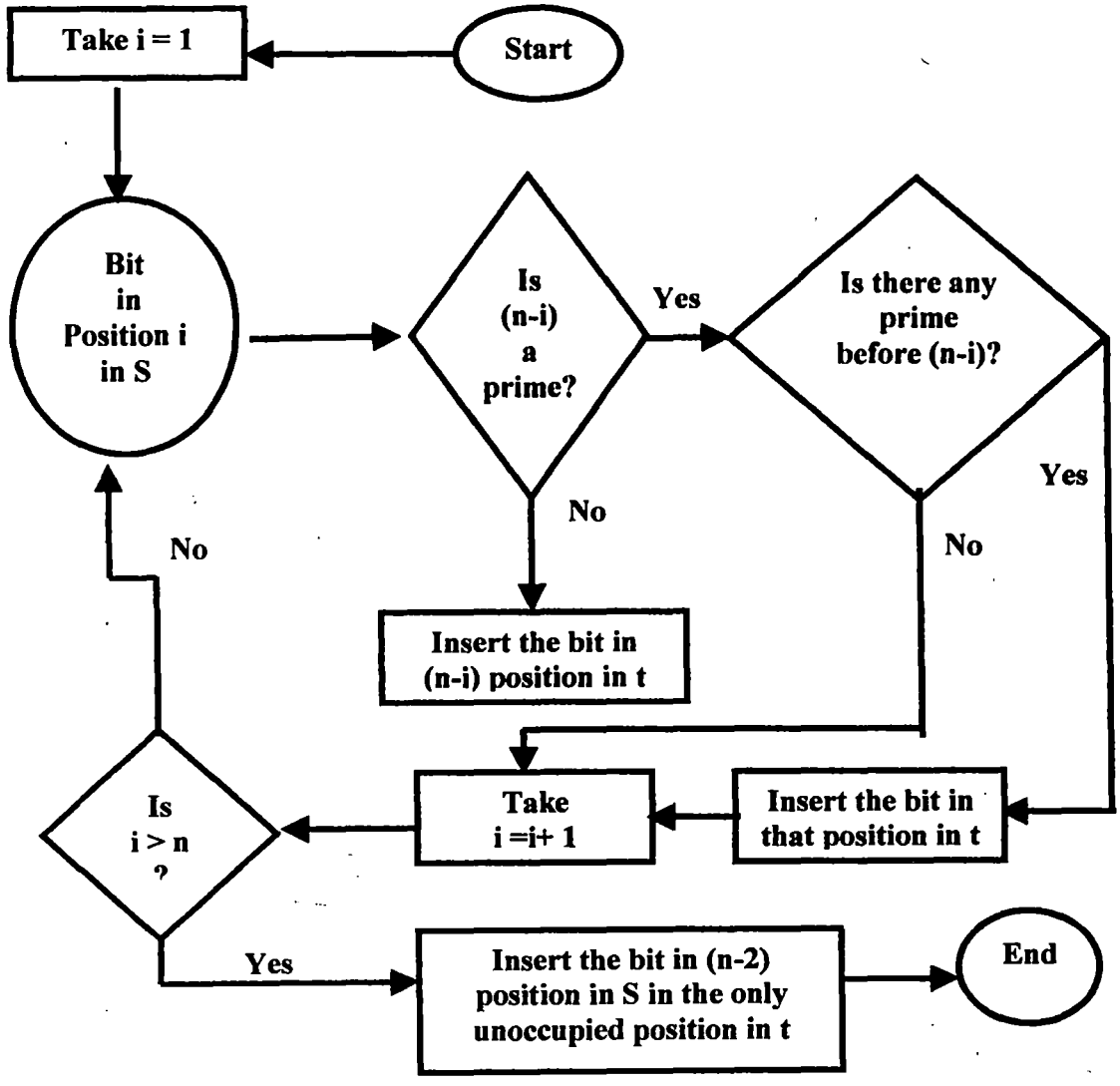

Figure 2.2.1
Overview of Entire Technique

The generating function g(s, t), aiming to orient the positions of different bits, is applied on a block s of size n to generate an intermediate block t of the same size. The intermediate block t is generated by the following rules:

1. A bit in the position i (1 <= i <= n-2) in the block s becomes the bit in the position (n-i) in the block t, if (n-i) is a non-prime integer.
2. A bit in the position i (1 <= i <= (n-2)) in the block s becomes the bit in the position j (1 <= j <= (n-i-1)) in the block t, where j is the precedent prime integer (if any) of (n-i), if (n-i) is a prime integer.
3. A bit in the position n in the block s remains in the same position in the block t
4. A bit in the position (n-2) in the block s is transferred in the block t to the position unoccupied by any bit after rules 1, 2 and 3 are applied.

The technique is illustrated using a flow diagram given in figure 2.2.1.

Now, for a block of finite size (n), a finite number of iterations (I) are required to regenerate the source block, where for an iteration, the source is the target block of the previous block and for the 1st iteration, the source is the source block of the entire technique. It can be shown that if for the p^{th} bit from MSB (1 <= p <= n) in the source block, i_p is the number of iterations required to be re-oriented to its source position, the total number of iterations (I) required to regenerate the source block is LCM of i_1, i_2, i_3, ..., i_p.

The process of encryption is the sub set of the entire set of work to form the cycle for a given block. Any intermediate block during the process of forming the cycle may be considered as the encrypted block. Hence if the number of iterations required to form the cycle is I for a block of size n, the number of intermediate blocks generated in the cycle is (i-1), because each iteration generates one block and the final iteration generates the target block, which, in turn, is the source block. Therefore if the p^{th} block formed in the cycle is considered to be the encrypted block, p may vary from 1 to (i-1).

Like in encryption, the process of decryption is also a sub set of the entire set of work required to form the cycle for the block. The only difference is that one intermediate block of the cycle (the encrypted block) is considered to be the source block

in the process of decryption. If the block generated after the p^{th} iteration in the cycle (1 <= p <=(I-1), I being the total number of iterations required to form the cycle) is considered to be the encrypted block, number of iterations required to decrypt the encrypted block is (I-p).

2.2.1 Example

Reorientation of bits in a finite length of block can be explained better with an example. Let us consider the block s = 01001011 of size 8 bits. Figure 2.2.1.1 depicts the different intermediate blocks and the target block generated during the encoding process. The Flow diagram to show how positions of the bits of s and the different intermediate blocks can be reoriented to complete the cycle is shown in figure 2.2.1.2. In this diagram, each arrow indicates positional orientation of a bit during iteration.

Source Block

0	1	0	0	1	0	1	1
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Block After 1st Iteration

1	1	0	0	0	1	0	1
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Block After 2nd Iteration

0	0	0	0	1	1	1	1
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Block After 3rd Iteration

1	1	0	0	0	0	1	1
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Block After 4th Iteration

1	0	0	0	1	1	0	1
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Block After 5th Iteration (Source Block)

0	1	0	0	1	0	1	1
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Figure 2.2.1.1
Different Intermediate and Target Blocks Generated in Forming the Cycle for Block 01001011

Here since number of iterations (I) required to form the cycle is 5, the p^{th} block may be considered as the encrypted block, where p ranges from 1 to 4.

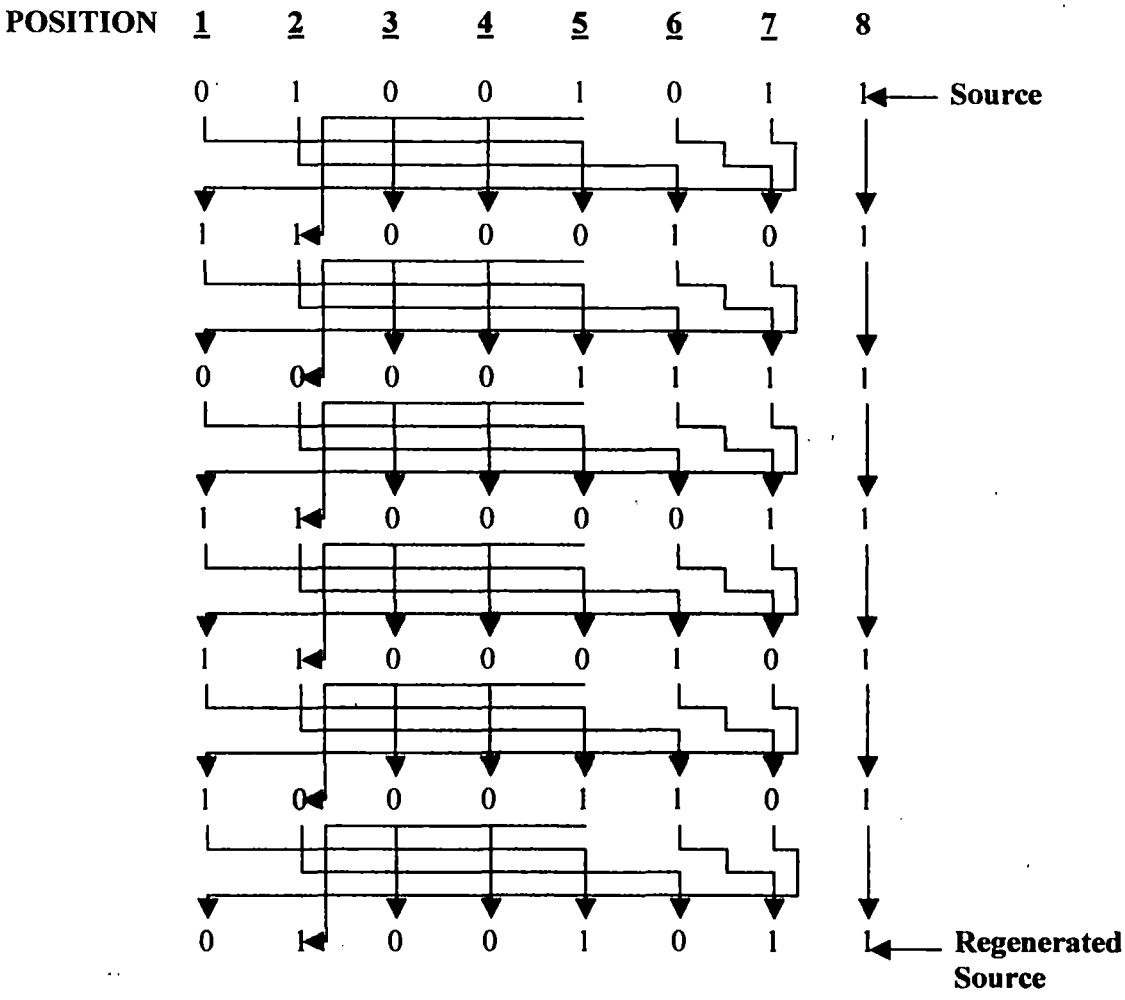

Figure 2.2.1.2
Formation of a Cycle for the Block 01001011

For this given example, the mapping (X → Y) between the position (X) of a bit in source block s and that (Y) of the same bit in target block t is shown in table 2.2.1.1 along with the logic followed. Here the block size n = 8. Table 2.2.1.2 shows the flow of bits As per the proposed algorithm and number of iterations needed to regenerate the 8-bit source block considering "01001001" as the input stream.

Table 2.2.1.1
Illustration of Mapping (X → Y) for Block of Size 8

X	Y	Logic Followed
1	5	Here 8 – 1 = 7, a prime; the precedent prime of 7 is 5
2	6	Here 8 – 2 = 6, a non-prime
3	3	Here 8 – 3 = 5, a prime, the precedent prime of 5 is 3
4	4	Here 8 – 4 = 4, a non-prime
5	2	Here 8 – 5 = 3, a prime, the precedent prime of 3 is 2
6	?	Here 8 – 6 = 2, a prime, there is no precedent prime, allocation suspended
7	1	Here 8 – 7 = 1, a non-prime
8	8	Here 8 being the position of the LSB, no change in position
6	7	One allocation was suspended earlier; since the position 7 is only the unoccupied position so far, that allocation is made there

Table 2.2.1.2
Illustration of Number of Iterations Required to Regenerate the Block of 8 Bits

Position in the Source Block p (0 <=p<= 7)	Reorientation of Position through Different Steps					Total No. of Iterations Required to Retain the Original Position i_p (0 <=p<= 7)
	Step 1	Step 2	Step 3	Step 4	Step 5	
1	1 → 5	5 → 2	2 → 6	6 → 7	7 → 1	5
2	2 → 6	6 → 7	7 → 1	1 → 5	5 → 2	5
3	3 → 3	--	--	--	--	1
4	4 → 4	--	--	--	--	1
5	5 → 2	2 → 6	6 → 7	7 → 1	1 → 5	5
6	6 → 7	7 → 1	1 → 5	5 → 2	2 → 6	5
7	7 → 1	1 → 5	5 → 2	2 → 6	6 → 7	5
8	8 → 8	--	--	--	--	1
Total No. of Iterations Required to Regenerate The Block	**LCM of 5, 5, 1, 1, 5, 5, 5, 1**					**5**

2.2.1.1 Example of Encryption

Let us take the block s = 01001011.considered in section 2.2.1. As shown in table 2.2.1.2, the number of iterations required to form the cycle is 5. Figure 2.2.1.1 shows different intermediate and target block generated through different iterations in the cycle.

2.2.1.2 Example of Decryption

Following the example given in section 2.2.1.2, if the block 00001111, generated after iteration 2, is considered to be the encrypted block, number of iterations required to decrypt the encrypted block is 5-2=3.

2.3 Implementation

In this section, we will consider a simple text message that is to be transmitted by encrypting using RPSP technique and after the transmission is over the encrypted message is to be decrypted using the same technique.

Consider the plaintext (P) as: Data Encryption. This stream is taken as the source stream.

Table 2.3.1 shows how each character in P can be converted into a byte.

Table 2.3.1
Character-to-Byte Conversion for the String "Data Encryption"

Character	Byte
D	01000100
a	01100001
t	01110100
a	01100001
<blank>	11111111
E	01000101
n	01101110
c	01100011
r	01110010
y	01111001
p	01110000
t	01110100
i	01101001
o	01101111
n	01101110

Combining all these bytes we obtain the following source stream of bits (S) consisting of 120 bits, where "/" is used as the separator between two consecutive bytes:

01000100/01100001/01110100/01100001/11111111/01000101/01101110/01100011/
01110010/01111001/01110000/01110100/01101001/01101111/01101110.

For the simplicity, let us take block size as 16, so that the number of source blocks is 7 and the final 8 bits are being kept as it is.

Now, using the analogy of table 2.2.1.1, another table may be formed to illustrate the mapping (X→Y) for blocks of size 16. Table 2.3.2 illustrates the same.

Table 2.3.2
Illustration of Mapping (X → Y) for Source Block of Size 16

X	Y	Logic Followed
1	15	Here 16 – 1 = 15, a non-prime
2	14	Here 16 – 2 = 14, a non-prime
3	11	Here 16 – 3 = 13, a prime, the precedent prime of 13 is 11
4	12	Here 16 – 4 = 12, a non-prime
5	7	Here 16 – 5 = 11, a prime, the precedent prime of 11 is 7
6	10	Here 16 – 6 = 10, a non-prime
7	9	Here 16 – 7 = 9, a non-prime
8	8	Here 16 – 8 = 8, a non-prime
9	5	Here 16 – 9 = 7, a prime, the precedent prime of 7 is 5
10	6	Here 16 – 10 = 6, a non-prime
11	3	Here 16 – 11 = 5, a prime, the precedent prime of 5 is 3
12	4	Here 16 – 12 = 4, a non-prime
13	2	Here 16 – 13 = 3, a prime, the precedent prime of 3 is 2
14	?	Here 16 – 14 = 2, a prime, but since it has no precedent prime, allocation is temporarily suspended
15	1	Here 16 – 15 = 1, a non-prime
16	16	Being the position of the LSB, it is kept as it is
14	13	One allocation was suspended earlier; since the position 13 is only the unoccupied position so far, that allocation is made there

Now, the source blocks generated from the source stream are:

$S_1 = 0100010001100001$ $S_2 = 0111010001100001$

$S_3 = 1111111101000101$ $S_4 = 0110111001100011$

$S_5 = 0111001001111001$ $S_6 = 0111000001110100$

$S_7 = 0110100101101111$

It is seen that for the block of size 16, the number of iterations required to regenerate the source itself is 6. Table 2.3.3 to Table 2.3.9 show the formation of cycles for blocks S_1, S_2, S_3, S_4, S_5, S_6 and S_7 respectively. Now, for each of the blocks, an arbitrary intermediate block, as indicated in each table, is considered as the encrypted stream.

Table 2.3.3
Formation of Cycle for Block S_1

Source Block	0100010001100001
Block (I_{11}) after iteration 1	0010010001000101
Block (I_{12}) after iteration 2	0000010001101001
Block (I_{13}) after iteration 3	0110010001000001
Block (I_{14}) after iteration 4	0000010001100101
Block (I_{15}) after iteration 5	0010010001001001
Block after iteration 6 (Source Block)	0100010001100001

Encrypted Block

Table 2.3.4
Formation of Cycle for Block S_2

Source Block	0111010001100001
Block (I_{21}) after iteration 1	0010010001110101
Block (I_{22}) after iteration 2	0011010001101001
Block (I_{23}) after iteration 3	0110010001110001
Block (I_{24}) after iteration 4	0011010001100101
Block (I_{25}) after iteration 5	0010010001111001
Block after iteration 6 (Source Block)	0111010001100001

Encrypted Block

Table 2.3.5
Formation of Cycle for Block S_3

Source Block	1111111101000101
Block (I_{31}) after iteration 1	0000011111111111
Block (I_{32}) after iteration 2	1111110111001001
Block (I_{33}) after iteration 3	0100111101110111
Block (I_{34}) after iteration 4	1011011111001101
Block (I_{35}) after iteration 5	0100110111111011
Block after iteration 6 (Source Block)	1111111101000101

Encrypted Block

Table 2.3.6
Formation of Cycle for Block S_4

Source Block	0110111001100011
Block (I_{41}) after iteration 1	1010011011100101
Block (I_{42}) after iteration 2	0010110011101011
Block (I_{43}) after iteration 3	1110111001100001
Block (I_{44}) after iteration 4	0010011011100111
Block (I_{45}) after iteration 5	1010110011101001
Block after iteration 6 (Source Block)	0110111001100011

Encrypted Block

Table 2.3.7
Formation of Cycle for Block S_5

Source Block	0111001001111001
Block (I_{51}) after iteration 1	0111010010110101
Block (I_{52}) after iteration 2	0011100001111101
Block (I_{53}) after iteration 3	0111011000111001
Block (I_{54}) after iteration 4	0111000011110101
Block (I_{55}) after iteration 5	0011110000111101
Block after iteration 6 (Source Block)	0111001001111001

Encrypted Block

Table 2.3.8
Formation of Cycle for Block S_6

Source Block	0111000001110100
Block (I_{61}) after iteration 1	0011010000111100
Block (I_{62}) after iteration 2	0111000001111000
Block (I_{63}) after iteration 3	0111010000110100
Block (I_{64}) after iteration 4	0011000001111100
Block (I_{65}) after iteration 5	0111010000111000
Block after iteration 6 (Source Block)	0111000001110100

Encrypted Block

Table 2.3.9
Formation of Cycle for Block S_7

Source Block	01101001011011111
Block (I_{71}) after iteration 1	1110011100101101
Block (I_{72}) after iteration 2	0110000111101111
Block (I_{73}) after iteration 3	1110110100101101
Block (I_{74}) after iteration 4	0110001101101111
Block (I_{75}) after iteration 5	1110010110101101
Block after iteration 6 (Source Block)	01101001011011111

Encrypted Block

As indicated in tables 2.3.3 to 2.3.9, intermediate blocks I_{14} (0000010001100101), I_{21} (0010010001110101), I_{33} (0100111101110111), I_{42} (0010110011101011), I_{55} (0011110000111101), I_{62} (0111000001111000) and I_{72} (0110000111101111) are considered as the encrypted blocks, so that these, together with the last 8-bit block (01101110) that was kept as it is, form the encrypted stream as follows:
0000010001100101/0010010001110101/0100111101110111/0010110011101011/00111
10000111101/0111000001111000/0110000111101111/01101110, "/" being used as only the separator.

The encrypted stream can be rewritten as the series of bytes as follows:
00000100/01100101/00100100/01110101/01001111/01110111/00101100/11101011/
00111100/00111101/01110000/01111000/01100001/11101111/01101110.

Converting the bytes into the corresponding characters, the following text is obtained as the encrypted text, which is to be transmitted:

C =✖e$uOw,δ<=pxa□n

Now, the process for the decryption is the same as the encryption. After converting the ciphertext C into a stream of bits, it is to be decomposed into the 16-bit blocks (C_1, C_2, ..., C_7). For each block, the same process as the encryption process is to

be followed but for varying number of iterations. For example, while decrypting the encrypted block C_1, the number of iterations should be 6 – 4 = 2, as the encrypted block was obtained after 4 iterations and forming a cycle requires a total of 6 iterations. The same logic is to be applied for the remaining blocks. After obtaining the source blocks in this way, they are combined together along with the last 8-bit block, which was kept as it is, in the same sequence and thus the source stream of bit is obtained, from which the source text is regenerated.

2.4 Results

For the purpose of the practical implementation, like all the other techniques, RPSP technique has also been implemented on total 50 files. Out of these, there are *EXE*, *COM*, *DLL*, *SYS* and *CPP* files, each category being 10 in number.

Section 2.4.1 presents report of the encryption/decryption times and the Chi Square values. Section 2.4.2 presents result of frequency distribution tests. A comparative result with the RSA system is given in section 2.4.3.

2.4.1 Result for Encryption/Decryption Time and Chi Square Value

Section 2.4.1.1 shows the result on *EXE* files, section 2.4.1.2 shows the result on *COM* files, section 2.4.1.3 shows the result on *DLL* files, section 2.4.1.4 shows the result on *SYS* files and section 2.4.1.5 shows the result on *CPP* files. In all the cases, the sample blocks are taken of the same 64-bit size. In this situation, it is seen that the total number of iterations required to regenerate the source block is 84. During encrypting the source files, the number of iterations is the integral part of the half of the total number of iterations required to regenerate the blocks, i.e., (int) (84/2) = 42, in this case. The remaining iterations are performed during the decryption [44, 51, 55, 56].

Section 2.4.1.6 provides a report on the results for different block sizes to show how the encryption time changes accordingly for the same sample file.

2.4.1.1 Result for *EXE* Files

Table 2.4.1.1.1 gives the result of implementing the RPSP technique on different executable files. The result includes the source size, the time for encryption and the time for decryption. From the table, it is clear that the encryption and the decryption time increase with the increment in the size of the source file.

Ten executable files are taken. The size of the files varies from 11611 bytes to 59398 bytes. The encryption time varies from 5.2747 seconds to 25.2198 seconds, whereas the decryption time also varies from 5.2747 seconds to 25.2198 seconds. The values of the Chi Square test results vary from 2239 to19973 with the degree of freedom varies from 248 to 255. From these Chi Square values a high degree of non-homogeneity of each encrypted file is seen in comparison to the corresponding source file.

Table 2.4.1.1.1
Result for *EXE* Files

Source File	Encrypted File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	Chi Square Value	Degree of Freedom
TLIB.EXE	*A1.EXE*	37220	15.8791	15.8242	14479	255
MAKER.EXE	*A2.EXE*	59398	25.2198	25.2198	19973	255
UNZIP.EXE	*A3.EXE*	23044	9.8352	9.7802	2239	255
RPPO.EXE	*A4.EXE*	35425	15.0000	15.0000	5277	255
PRIME.EXE	*A5.EXE*	37152	15.8242	15.7692	5485	255
TCDEF.EXE	*A6.EXE*	11611	5.2747	5.2747	3791	254
TRIANGLE.EXE	*A7.EXE*	36242	15.4945	15.4945	5690	255
PING.EXE	*A8.EXE*	24576	10.4945	10.4396	4335	248
NETSTAT.EXE	*A9.EXE*	32768	13.9011	13.9011	8725	255
CLIPBRD.EXE	*A10.EXE*	18432	7.8571	7.8571	4964	255

Here since the operations while encrypting and decrypting are almost the same, the times needed for the encryption and the decryption are of little or no difference.

Figure 2.4.1.1.1 depicts the relationship between the source file size and the encryption time for *EXE* files. As is shown in the graph in figure 2.4.1.1.1, the encryption time increases almost linearly with the source file size.

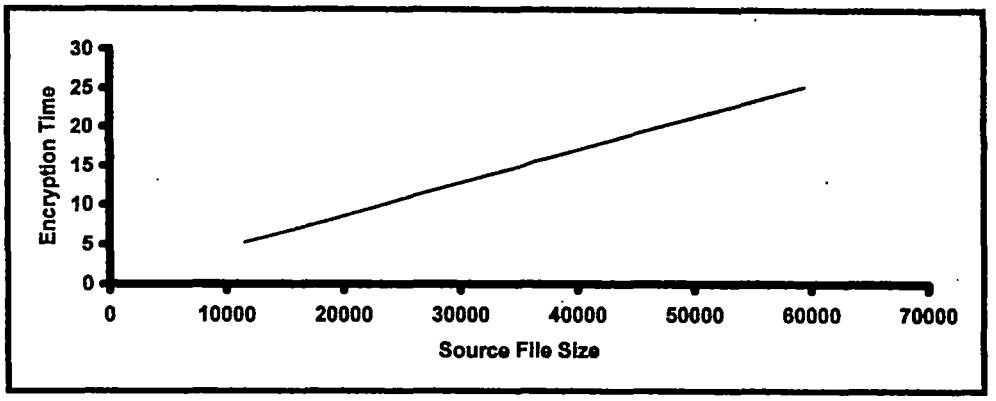

Figure 2.4.1.1.1
Graph to establish Relationship between Source Size and Encryption Time (For *EXE* Files)

2.4.1.2 Result for *COM* Files

Table 2.4.1.2.1 shows the result of implementing the technique on 10 sample *COM* files. Ten command files are taken. The size of the files varies from 5239 bytes to 29271 bytes. The encryption time varies from 2.2527 seconds to 12.4176 seconds, whereas the decryption time also varies from 2.2527 seconds to 12.4725 seconds. The values of the Chi Square test results vary from 731 to 5019 with the degree of freedom vary from 230 to 255. From these Chi Square values a high degree of non-homogeneity of each encrypted file is seen in comparison to the corresponding source file.

Table 2.4.1.2.1
Result for *.COM* Files

Source File	Encrypted File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	Chi Square Value	Degree of Freedom
EMSTEST.COM	*A1.COM*	19664	8.4066	8.4066	3890	255
THELP.COM	*A2.COM*	11072	4.7802	4.7253	2563	250
WIN.COM	*A3.COM*	24791	10.6044	10.5494	3887	252
KEYB.COM	*A4.COM*	19927	8.5165	8.4615	4023	255
CHOICE.COM	*A5.COM*	5239	2.2527	2.2527	1225	232
DISKCOPY.COM	*A6.COM*	21975	9.3407	9.3407	4221	254
DOSKEY.COM	*A7.COM*	15495	6.5934	6.5934	4105	253
MODE.COM	*A8.COM*	29271	12.4176	12.4725	5019	255
MORE.COM	*A9.COM*	10471	4.5055	4.4505	731	230
SYS.COM	*A10.COM*	18967	8.0769	8.0769	3131	254

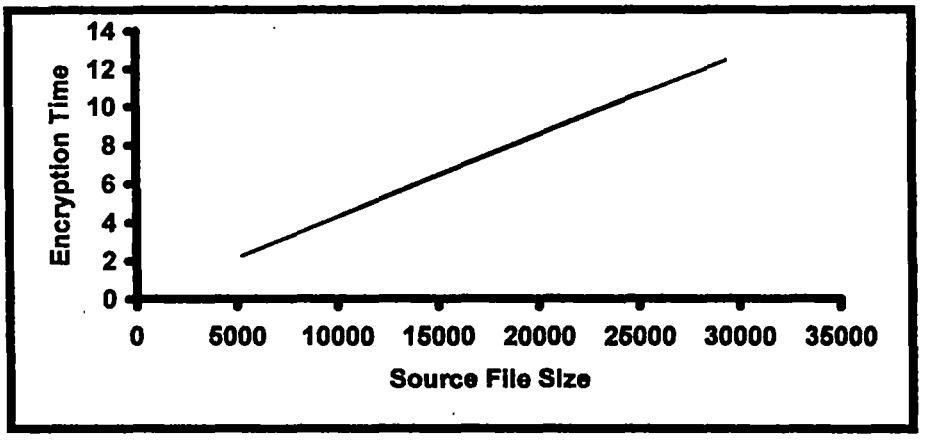

Figure 2.4.1.2.1
Graph to establish Relationship between Source Size and Encryption Time (For *COM* Files)

Figure 2.4.1.2.1 shows the graphical relationship between the source size and the encryption time for *COM* files. As shown in the figure, the relationship is linear. This means that the encryption time increases linearly with the size of the source file considered for encryption.

2.4.1.3 Result for DLL Files

Table 2.4.1.3.1 gives the result of implementing the technique on 10 sample *DLL* files. Ten dll files are taken. The size of the files varies from 3216 bytes to 58368 bytes. The encryption time varies from 1.4286 seconds to 24.8352 seconds, whereas the decryption time also varies from 1.3736 seconds to 24.8352 seconds. The values of the Chi Square test results vary from 872 to 23953 with the degree of freedom vary from 217 to 255. From these Chi Square values a high degree of non-homogeneity of each encrypted file is seen in comparison to the corresponding source file.

Table 2.4.1.3.1
Result for *.DLL* Files

Source File	Encrypted File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	Chi Square Value	Degree of Freedom
SNMPAPI.DLL	*A1.DLL*	32768	13.9560	13.9011	5987	253
KPSHARP.DLL	*A2.DLL*	31744	13.5165	13.5165	23318	254
WINSOCK.DLL	*A3.DLL*	21504	9.1758	9.1758	9648	252
SPWHPT.DLL	*A4.DLL*	32792	13.9560	13.9560	7781	255
HIDCI.DLL	*A5.DLL*	3216	1.4286	1.3736	872	217
PFPICK.DLL	*A6.DLL*	58368	24.8352	24.8352	12324	255
NDDEAPI.DLL	*A7.DLL*	14032	5.9890	5.9890	4693	249
NDDENB.DLL	*A8.DLL*	10976	4.7253	4.7253	8269	251
ICCCODES.DLL	*A9.DLL*	20992	8.9011	8.9560	9693	252
KPSCALE.DLL	*A10.DLL*	31232	13.2967	13.2967	23953	255

Figure 2.4.1.3.1 shows the relationship between the source file size and the encryption time for these *DLL* files. There exists a linear relationship between these two parameters. This proves that for DLL files also the encryption time increase linearly with the source file size.

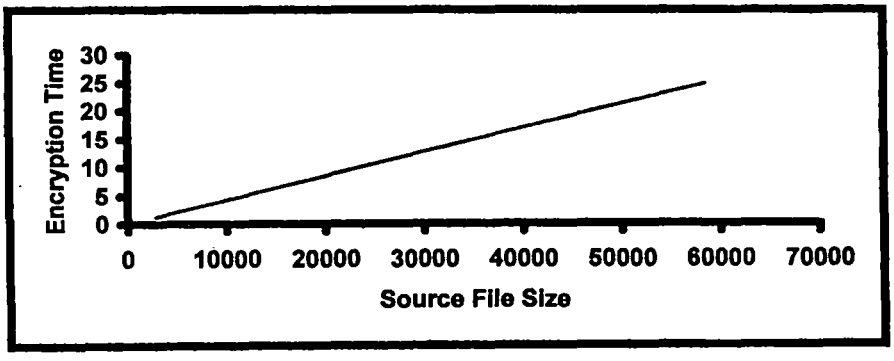

Figure 2.4.1.3.1
Graph to establish Relationship between Source Size and Encryption Time (For *DLL* Files)

2.4.1.4 Result for *SYS* Files

Table 2.4.1.4.1 shows the result after implementing the technique on 10 sample *SYS* files. Ten system files are taken. The size of the files varies from 1105 bytes to 33191 bytes. The encryption time varies from 1.1538 seconds to 14.1758 seconds,

whereas the decryption time also varies from 1.1538 seconds to 14.1209 seconds. The values of the Chi Square test results vary from 169 to 20163 with the degree of freedom vary from 165 to 255. From these Chi Square values a high degree of non-homogeneity of each encrypted file is seen in comparison to the corresponding source file.

Table 2.4.1.4.1
Result for *SYS* Files

Source File	Encrypted File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	Chi Square Value	Degree of Freedom
HIMEM.SYS	*A1.SYS*	33191	14.1758	14.1209	9189	255
RAMDRIVE.SYS	*A2.SYS*	12663	5.3846	5.3846	1425	241
USBD.SYS	*A3.SYS*	18912	8.0769	8.0220	9225	255
CMD640X.SYS	*A4.SYS*	24626	10.4945	10.4396	6248	255
CMD640X2.SYS	*A5.SYS*	20901	8.8462	8.9011	5759	255
REDBOOK.SYS	*A6.SYS*	5664	2.4725	2.4176	1946	230
IFSHLP.SYS	*A7.SYS*	3708	1.5934	1.5934	1040	237
ASPI2HLP.SYS	*A8.SYS*	1105	0.4945	0.4396	169	165
DBLBUFF.SYS	*A9.SYS*	2614	1.1538	1.1538	519	215
CCPORT.SYS	*A10.SYS*	31680	13.4615	13.4615	20163	255

Figure 2.4.1.4.1 establishes the graphical relationship between the source size and the decryption time for *SYS* files. It indicates that here also there exists a linear relationship between these two parameters. Therefore the decryption time increase linearly with the source file size.

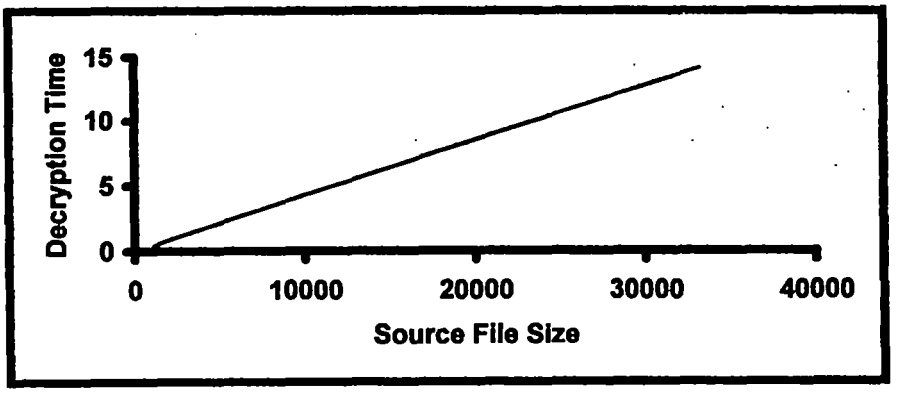

Figure 2.4.1.4.1
Graph to establish Relationship between Source Size and Decryption Time (For *SYS* Files)

2.4.1.5 Result for *CPP* Files

Table 2.4.1.5.1 summarizes the report of running the technique on 10 sample *CPP* files. Ten cpp files are taken. The size of the files varies from 4071 bytes to 14557 bytes. The encryption time varies from 0.5495 seconds to 13.6813 seconds, whereas the decryption time also varies from 0.5495 seconds to 13.6264 seconds. The values of the Chi Square test results vary from 415 to 101472 with the degree of freedom vary from 248 to 255. From these Chi Square values a high degree of non-homogeneity of each encrypted file is seen in comparison to the corresponding source file.

Table 2.4.1.5.1
Result for *CPP* Files

Source File	Encrypted File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	Chi Square Value	Degree of Freedom
BRICKS.CPP	*A1.CPP*	16723	7.1429	7.0879	26406	88
PROJECT.CPP	*A2.CPP*	32150	13.6813	13.6264	101472	90
ARITH.CPP	*A3.CPP*	9558	4.1209	4.0659	14157	77
START.CPP	*A4.CPP*	14557	6.2088	6.1538	36849	88
CHARTCOM.CPP	*A5.CPP*	14080	5.9890	5.9890	35754	84
BITIO.CPP	*A6.CPP*	4071	1.7582	1.7582	2628	70
MAINC.CPP	*A7.CPP*	4663	1.9780	1.9780	2380	83
TTEST.CPP	*A8.CPP*	1257	0.5495	0.5495	415	69
DO.CPP	*A9.CPP*	14481	6.1538	6.1538	31551	88
CAL.CPP	*A10.CPP*	9540	4.0659	4.0659	13499	77

Figure 2.4.1.5.1 establishes a graphical relationship between the source file size and the decryption time for these *CPP* files to draw the conclusion that here also there exists an almost linear relationship between these two values. Therefore for CPP files also the decryption time increases linearly with the source file size.

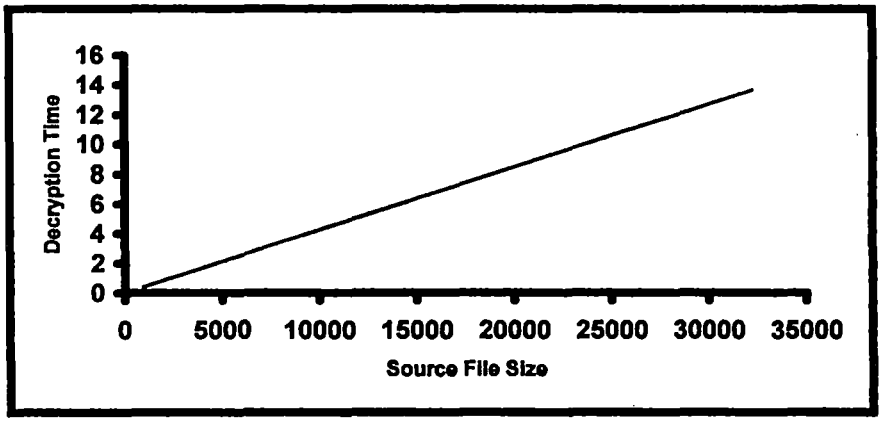

Figure 2.4.1.5.1
Graph to establish Relationship between Source Size and Decryption Time (For *CPP* Files)

2.4.1.6 Report on Variation of Encryption Time with Varying Block Sizes

For the purpose of observing the change in the encryption time with the change in the block size, one sample file CMD640X2.SYS has been considered. Table 2.4.1.6.1 shows the result.

If the unique block size of 8 bits is considered, the encryption time is observed as 0.3297 seconds. For the unique block size of 16 bits, the observed encryption time is 0.5495 seconds. The encryption time is observed as 2.7473 seconds for the unique block size of 32 bits. If the unique block length is considered as 64 bits, the encryption time is observed as 8.9011 seconds. For the block size of 128 bits, the encryption time is observed as 57.5824 seconds. If the block size is taken as 256 bits, the encryption time is seen as 276.8681 seconds. Finally, for the unique block size of 512 bits, the encryption time for encrypting the file CMD640X2.SYS is observed as 2545.9338 seconds.

Table 2.4.1.6.1
Encryption Time for Different Block Sizes for CMD640X2.SYS

Block Size	Encryption Time
8	0.3297
16	0.5495
32	2.7473
64	8.9011
128	57.5824
256	276.8681
512	2545.9338

Figure 2.4.1.6.1 illustrates graphically how the change in the encryption time is related to the block size. In this figure, the term "Real Curve" indicates the curve denoting the relationship between the source size in power of 2 form and the encryption time for the sample file considered, and the term "Exponential Trendline" indicates the typical exponential curve. The analogy between these two curves establishes the fact that there exists a tendency of the exponential change in the encryption time with N, where 2^N is the unique block size considered.

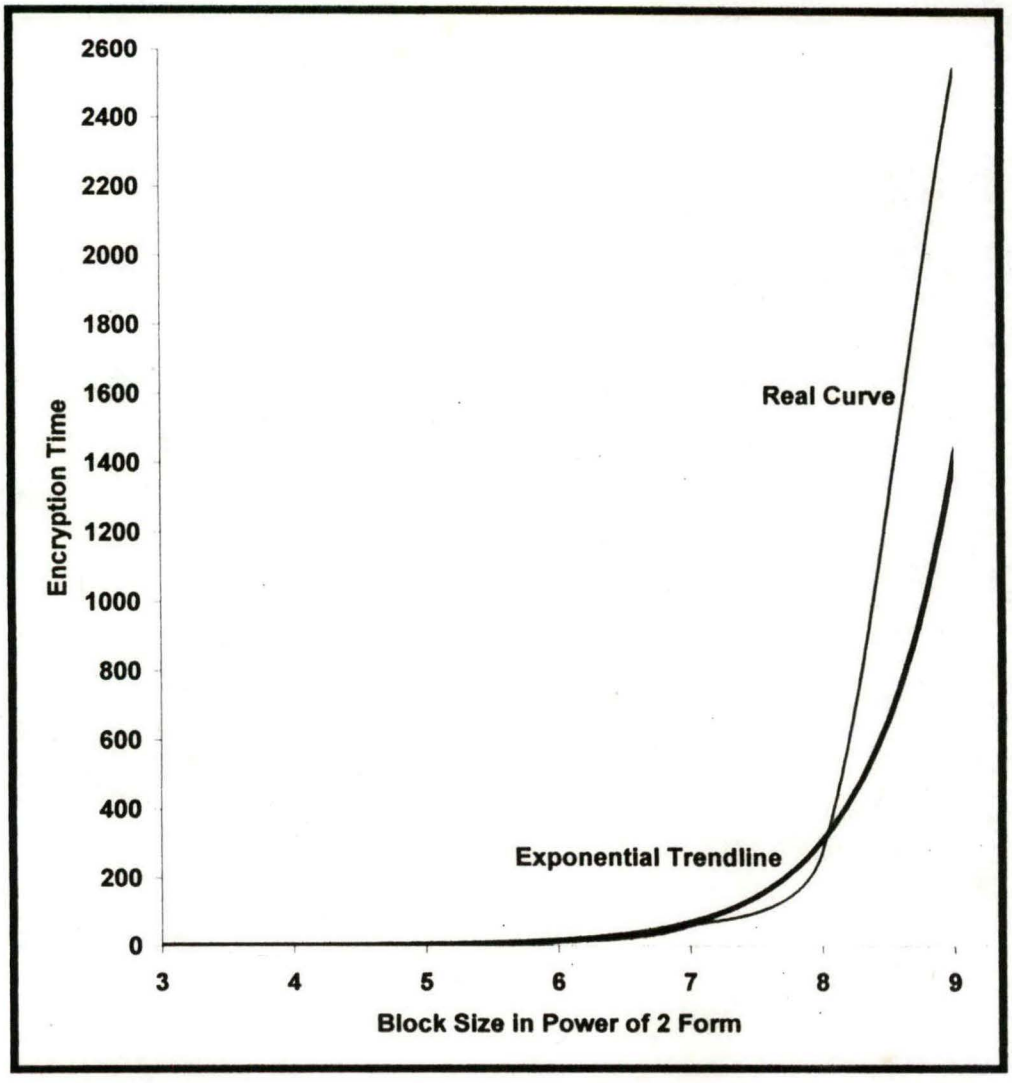

Figure 2.4.1.6.1
Tendency of Encryption Time to Change Exponentially with N, for Block Size of 2^N
(Result Observed for CMD640X2.SYS)

2.4.2 Result for Frequency Distribution Tests

The frequency of each of the 255 characters in the source file and the same in the encrypted file were calculated and compared to assess the efficiency of the proposed technique. All 50 sample files are considered for this purpose and the unique block size of 64 bits is considered while encrypting the files. Figure 2.4.2.1 to figure 2.4.2.5 show segments of graphical outcome only for arbitrarily chosen 5 files of different categories.

In each figure, red lines show the frequencies represented by the characters of the encrypted file whereas blue lines represent the frequencies of the characters of the source

file. From these figures it is very much clear that the source and the corresponding encrypted files are heterogeneous in nature. This can be interpreted that the proposed technique obtains a good quality of encryption.

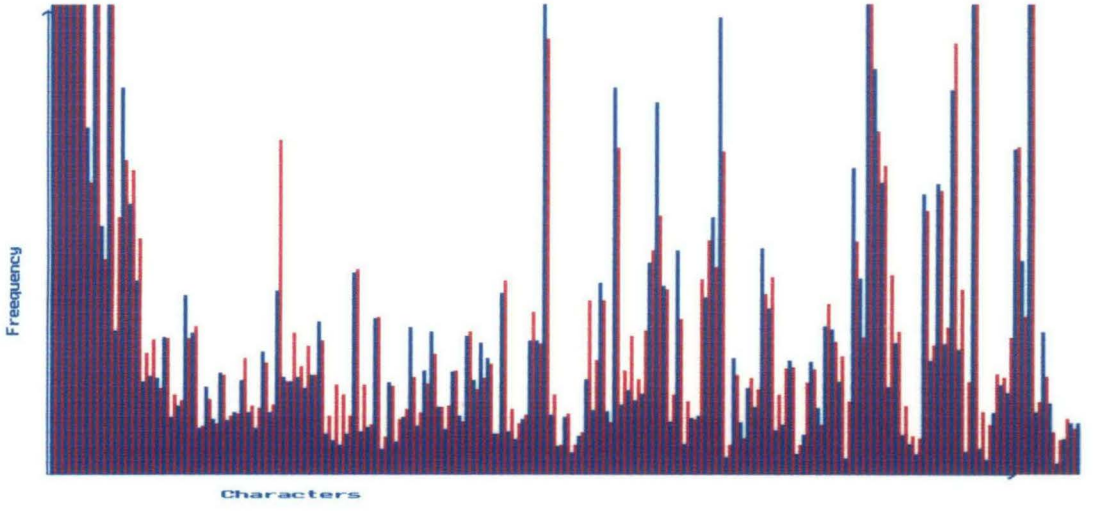

Figure 2.4.2.1
Frequency Distribution Chart for
RPPO.EXE and Encrypted FOX1.EXE

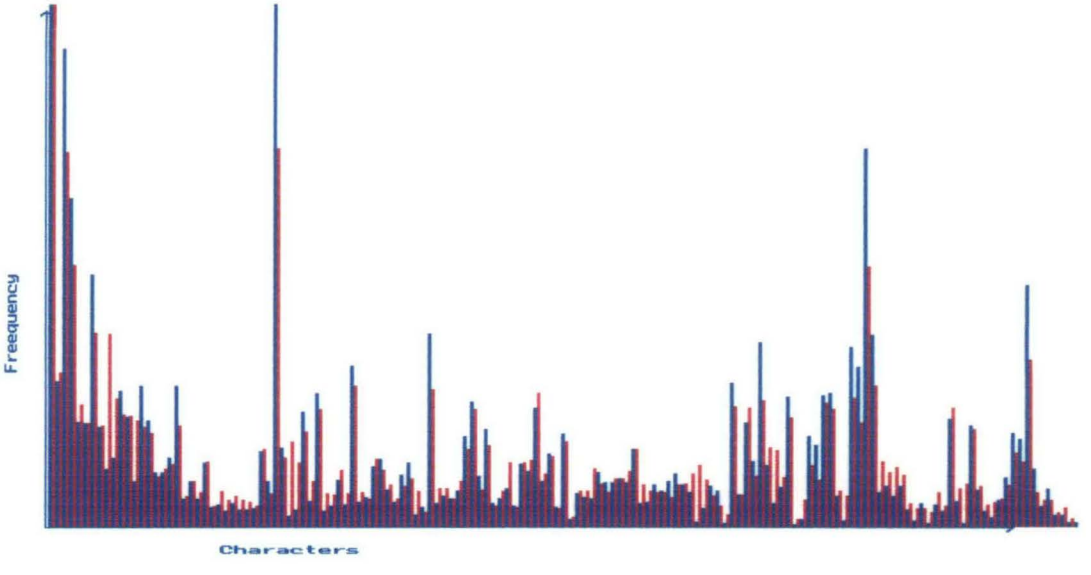

Figure 2.4.2.2
Frequency Distribution Chart for
DOSKEY.COM and Encrypted FOX1.COM

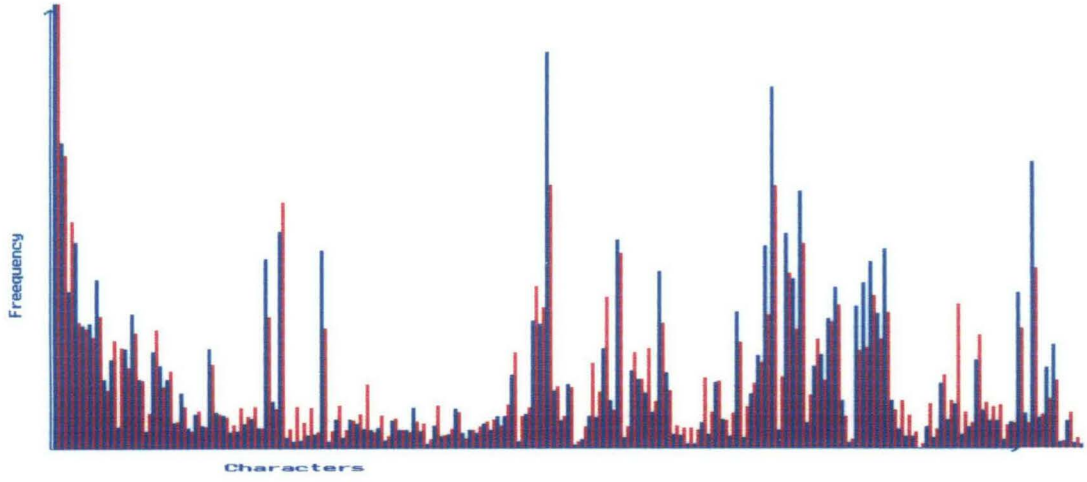

Figure 2.4.2.3
Frequency Distribution Chart for
NDDEAPI.DLL and Encrypted FOX1.DLL

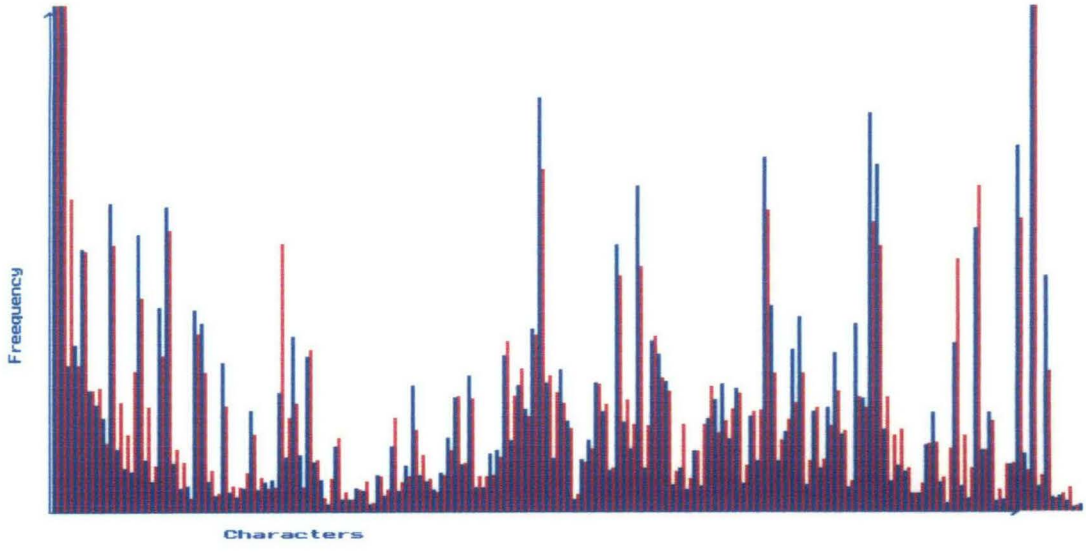

Figure 2.4.2.4
Frequency Distribution Chart for
USBD.SYS and Encrypted FOX1.SYS

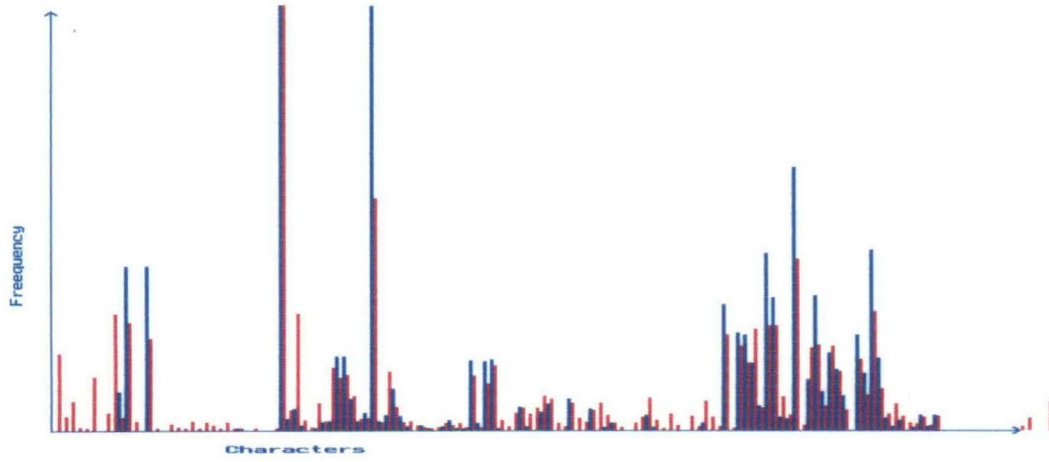

Figure 2.4.2.4
Frequency Distribution Chart for
BITIO.CPP and Encrypted FOX1.CPP

2.4.3 Comparison with RSA Technique

Chi square values between the sample .CPP files and corresponding three encrypted files, first using the RPSP technique with the unique block length of only 8 bits, second using the same with the unique block length of 64 bits, and the third using the existing RSA technique, are compared and the results are given in table 2.4.3.1 [7, 8].

When the proposed RPSP technique is implemented with 8-bit blocks, Chi Square values range from 1471 to 615796, and the same when is implemented for 64-bit blocks, Chi Square values range from 415 to 101472. On the other hand, if the same files are encrypted using the RSA technique, Chi Square values range from 3652 to 655734. In each case, the degree of freedom ranges from 69 to 90. From the table, it is also observed that the proposed technique produces the higher Chi Square values than the RSA technique for PROJECT.CPP (using 8-bit block), START.CPP (using 8-bit block), CHARTCOM.CPP (using 8-bit block), and MAINC.CPP (for 8-bit block).

Table 2.4.3.1
Comparison between Proposed RPSP and Existing RSA

Source File	Chi Square Values			Degree of Freedom
	For RPSP (8-bit Block)	For RPSP (64-bit Block)	For RSA	
BRICKS.CPP	109186	26406	200221	88
PROJECT.CPP	615796	101472	197728	90
ARITH.CPP	10842	14157	273982	77
START.CPP	80174	36849	49242	88
CHARTCOM.CPP	556552	35754	105384	84
BITIO.CPP	27462	2628	52529	70
MAINC.CPP	32724	2380	4964	83
TTEST.CPP	1471	415	3652	69
DO.CPP	80098	31551	655734	88
CAL.CPP	9540	13499	216498	77

Figure 2.4.3.1 is the diagrammatic representation of the comparison shown in table 2.4.3.1, where "red lines" stand for the RSA, "black lines" stand for the RPSP with 64-bit blocks, and "blue lines" stands for the RPSP with 8-bit blocks.

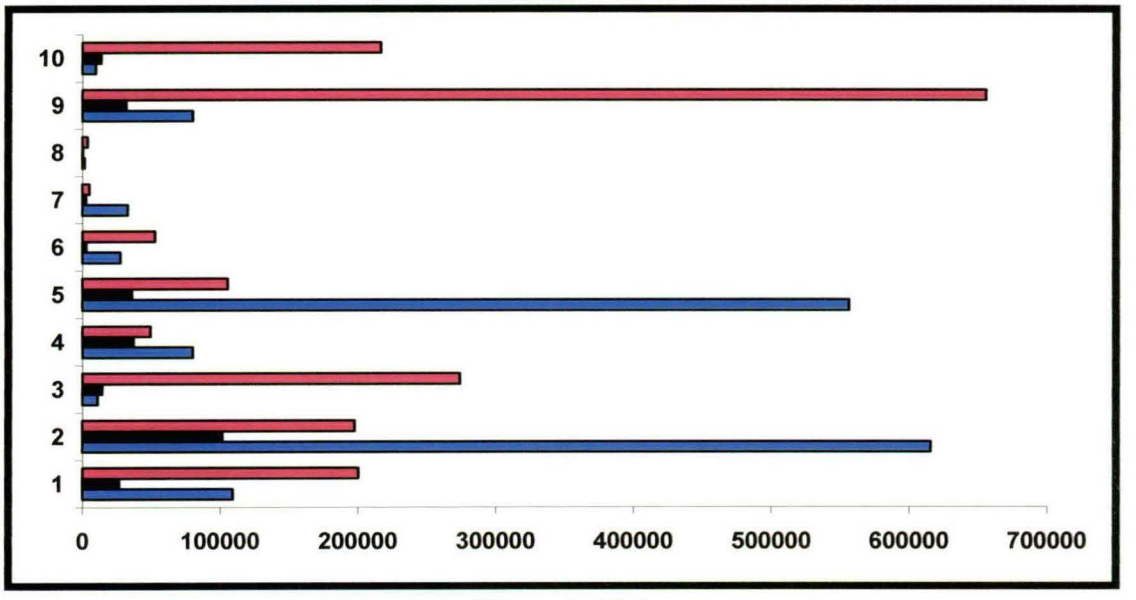

Figure 2.4.3.1
Graphical Comparison of Results of Chi Square Tests among Proposed RPSP (8-bit Block, 64-bit Block) and Existing RSA

2.5 Analysis

This section consists of analyses on three issues. One is the issue related to block size, discussed in section 2.5.2. Another issue is related to different factors that were considered to evaluate the technique, discussed in section 2.5.3. The last issue is the key of the technique, which is analyzed in chapter 8, a special chapter on the key generation. Prior to all these issues, the proof on the finiteness of the series of iterations has also been done in section 2.5.1.

2.5.1 Proof of Cycle Formation

The RPSP technique will not work if a source block is never regenerated. Now, the regeneration of a source block $S = s_0\ s_1\ s_2\ s_3\ s_4\ \ldots\ s_{n-1}$ means re-occupancy of the original positions by all the n bits of S. This re-occupancy is to be made after some positional reorientation of all the n bits in S. Now, if one or more bits never reoccupy their original positions, the source block will not be regenerated and hence the cycle will not be completed [3, 29, 30].

Now, it is to be proved that any bit s_i ($0 <= i <= n-1$) comes back into its original position (i) after a finite number of iterations.

Let, if possible, a bit s_i cannot reach its original position because it is stuck inside a loop shown in figure 2.5.1.1.

Iteration No. ⟶	**p**	**p+1**	**p+2**	**p+3**	**p+4**	**p+5**	**p+6**	**...**
Positions of s_i During different Iterations ⟶	**i**	**i_1**	**i_2**	**i_3**	**i_4**	**i_2**	**i_3**	**...**

Generation of an infinite loop

Figure 2.5.1.1
Generation of Infinite Loop (if possible) during Formation of Cycle

So, effectively what is happening here is two different bits, including s_i, are attempting to occupy the position i_2 during iteration no. (p+2); one from the position i_1, which is the bit s_i, and the other from the position i_4. Effectively the situation is like figure 2.5.1.2.

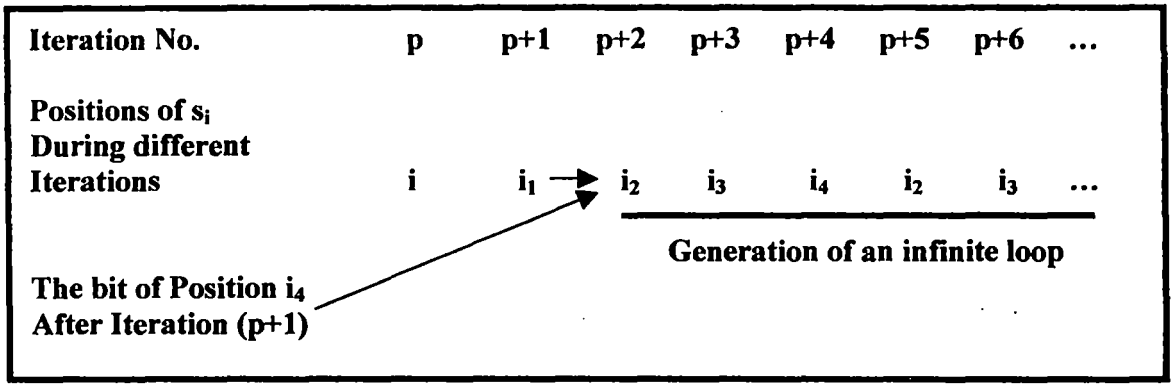

Figure 2.5.1.2
Two bits attempting to occupy the same position (if possible)

But as per the scheme is concerned, with respect to blocks of a fixed size, for each and every position in the intermediate and the final blocks there exists a unique position in the previous block, so that no two bits of two positions can be transferred to a single position. Therefore our assumption is contradicting the basic philosophy of the scheme. Hence our assumption is wrong.

So, it can be concluded that the process of regenerating the source block is absolutely a finite process.

2.5.2 Analysis on Block Size

To enhance the security, one criterion should be to choose the block size such a manner that it requires a huge number of iterations to complete the cycle. Now, generally the number of such iterations increases as the block size increases, but there exist many exceptions also in this regard. Table 2.5.2.1 shows how the number of such iterations changes with changes in block size [31, 32, 34].

Table 2.5.2.1
No. of Iterations (I) Required to form Cycles for Blocks of different Sizes (n)

n	I	n	I	n	I	n	I	n	I
1	1	7	6	13	14	19	22	25	30
2	1	8	5	14	20	20	60	26	12
3	2	9	5	15	18	21	26	27	18
4	3	10	6	16	6	22	6	28	30
5	5	11	8	17	12	23	16	29	18
6	5	12	7	18	14	24	14	30	18

The graph shown in figure 2.5.2.1 to represent table 2.5.2.1 depicts the clear picture in this regard. In this graph, block sizes only in the range of 1 to 30 have been considered. The most satisfactory result appears when the size of a block is taken as 20 because it requires a total of 64 iterations to complete the cycle.

But in practical case it is suggested to take the block size as of 2^n (n = 1,2,3...) form, preferably of at least 64-bit size. In that case we can observe a steady increase in the number of iterations to complete the cycle, as the value of n increases. Table 2.5.2.2 gives this information in a tabular presentation and figure 2.5.2.2 shows this relationship graphically.

Figure 2.5.2.1 shows that there exists a non-linear relationship between the number of iterations required to complete the cycle and the unique block size if the block size ranges from 1 to 30.

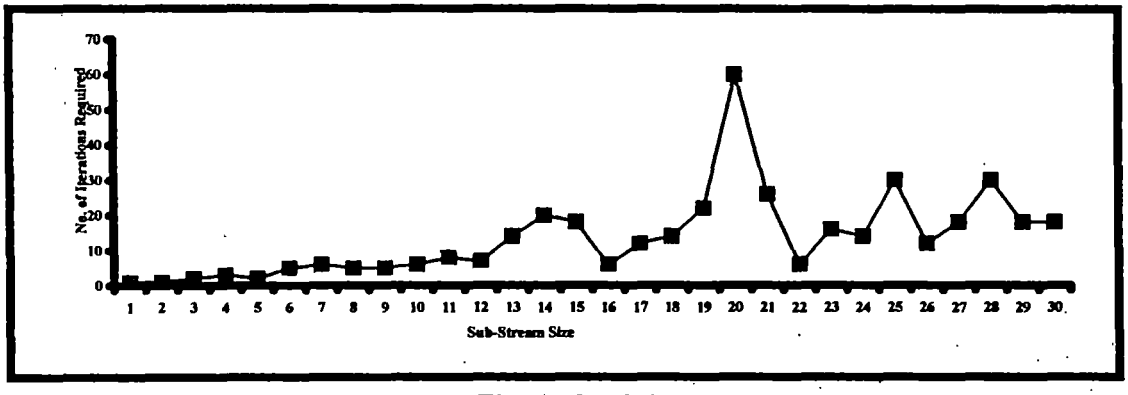

Figure 2.5.2.1
Relationship between Block Size and Number of Cycles to Form a Cycle

Table 2.5.2.2 shows a steady increment in the number of iterations to complete the cycle if the block size is considered in the form of 2^n. It is seen that for the values of n ranging from 0 to 8, numbers of iterations required are respectively 1, 1, 3, 5, 6, 30, 84, 456, and 1820.

Table 2.5.2.2
No. of Iterations (I) Required to form Cycles for Blocks of different Sizes (2^n)

n	I	n	I	n	I
0	1	3	5	6	84
1	1	4	6	7	456
2	3	5	30	8	1820

The steady increment in the number of iterations required completing the cycle is shown pictorially in figure 2.5.2.2.

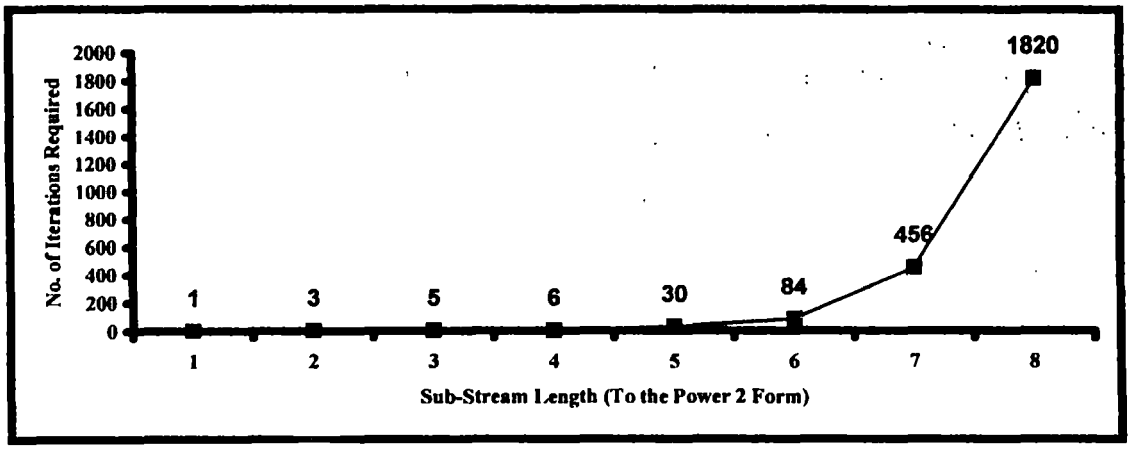

Figure 2.5.2.2
Graphical Relationship between Number of Iterations (I) Required to form Cycles for Blocks of different Sizes (2^n)

Now, the RPSP technique may be implemented in an intelligent way by making the blocks to be of different sizes. In that case, different blocks will require different number of iterations to complete the cycle. Naturally the total number of iterations to regenerate the entire stream of bits also will be different.

In this regard, let us consider a simple example. Say, there is a stream of bits of size 128 bits. Using this intelligent approach, the stream is being decomposed into blocks B_1 (64 bits), B_2 (20 bits), B_3 (19 bits), B_4 (13 bits) and B_5 (12 bits). Now, following table 2.5.2.1 and table 2.5.2.2, we get the information that B_1 requires 84 iterations, B_2 requires

60 iterations, B_3 requires 22 iterations, B_4 requires 14 iterations and B_5 requires 7 iterations. Therefore to regenerate the entire stream of bits, the minimum number of iterations required is 4620, which is obtained by taking the LCM of 84, 60, 22, 14 and 7; whereas a fixed block size of 64 bits would have required only 84 iterations to regenerate the entire stream. Thus by allowing blocks to be of varying sizes; a far better security can be achieved.

2.5.3 Analysis on Factors Considered for Evaluation Purpose

From the results shown in section 2.4, it is clear that for a set of fixed blocks, the execution time (the encryption as well as the decryption) varies almost linearly with the source file size. Now, since any type of source file is being considered simply as a stream of bits, the encryption (or the decryption) time does not depend on the type of the source file. Hence if the technique is applied for different types of files, all being of around the same size, the encryption (or the decryption) time will be more or less the same for a particular block size or for a particular set of different blocks. Table 2.5.3.1 establishes this fact and it is graphically shown in figure 2.5.3.1.

In table 2.5.3.1, seven files of almost the same size ranging from 31232 bytes to 33191 bytes have been considered. It is observed that using the proposed RPSP technique if these files are encrypted, the encryption time ranges from 13.2967 seconds to 14.1758 seconds, and the decryption time ranges from 13.2967 seconds to 14.1209 seconds. This means that for these files of almost similar sizes, the encryption/decryption times are also almost similar. Graphically this fact is established in figure 2.5.3.1

Table 2.5.3.1
Result of Encryption/Decryption Time for Different Types of Files of Almost Same Sizes

File Name	File Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In seconds)
PROJECT.CPP	32150	13.6813	13.6264
CCPORT.SYS	31680	13.4615	13.4615
HIMEM.SYS	33191	14.1758	14.1209
KPSCALE.DLL	31232	13.2967	13.2967
SPWHPT.DLL	32792	13.9560	13.9560
SNMPAPI.DLL	32768	13.9560	13.9011
NETSTAT.EXE	32768	13.9011	13.9011

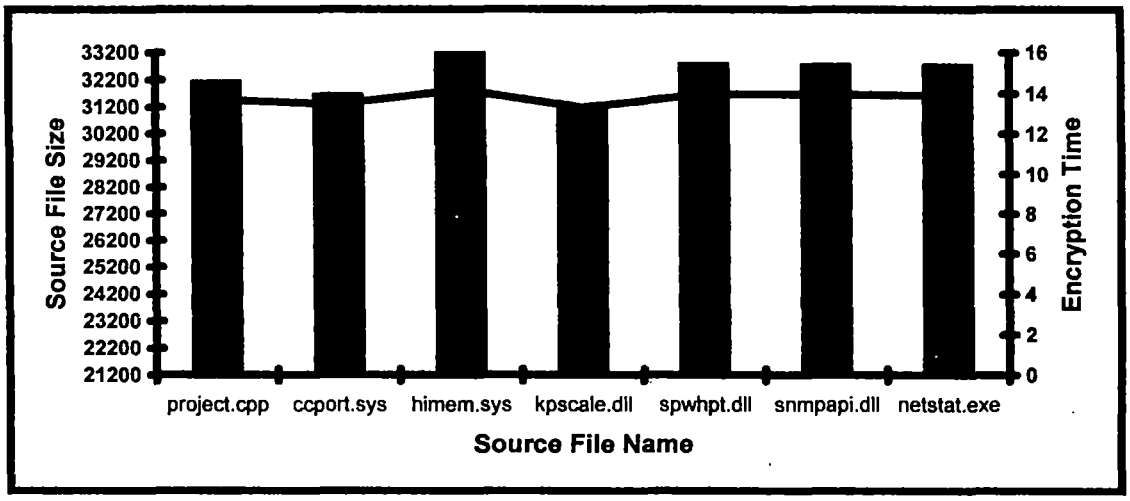

Figure 2.5.3.1
Graphical Relationship among Encryption Times of Files of Different Categories but almost of Same Sizes

Results obtained in section 2.4 on chi square values suggest the fact it is exactly file-dependent since there exists no fixed relationship between the file size and the chi square value. Out of all the results obtained, only those for almost the same file sizes have been considered to analyze their chi square values. Table 2.5.3.2 enlists these results and the graphical relationship is shown in figure 2.5.3.2.

In table 2.5.3.2, seven files have been considered, the sizes ranging from 32150 bytes to 37220 bytes. Five out of these files are with the degree of freedom as 255, one with 253, and the remaining with 90. For these sample files of almost similar sizes, Chi Square values range from 5485 to 14479, which means that in spite of the fact the files considered are of almost similar sizes, there Chi Square values vary to a large extend. It indicates that the Chi Square value is dependent to the content of the file, not to the size of the file.

In figure 2.5.3.2, the black pillars, which are almost of the similar heights, stand for the sizes of sample files, and the corresponding adjacent blue pillars, the heights of which vary to a large extend, stand for the corresponding Chi Square values.

Table 2.5.3.2
Result of Chi Square Values for
Different Types of Files of Almost Same Sizes

File Name	File Size	Chi Square Value	Degree of Freedom
TLIB.EXE	37220	14479	255
PRIME.EXE	37152	5485	255
TRIANGLE.EXE	36242	5690	255
SPWHPT.DLL	32792	7781	255
SNMPAPI.DLL	32768	5987	253
HIMEM.SYS	33191	9189	255
PROJECT.CPP	32150	101472	90

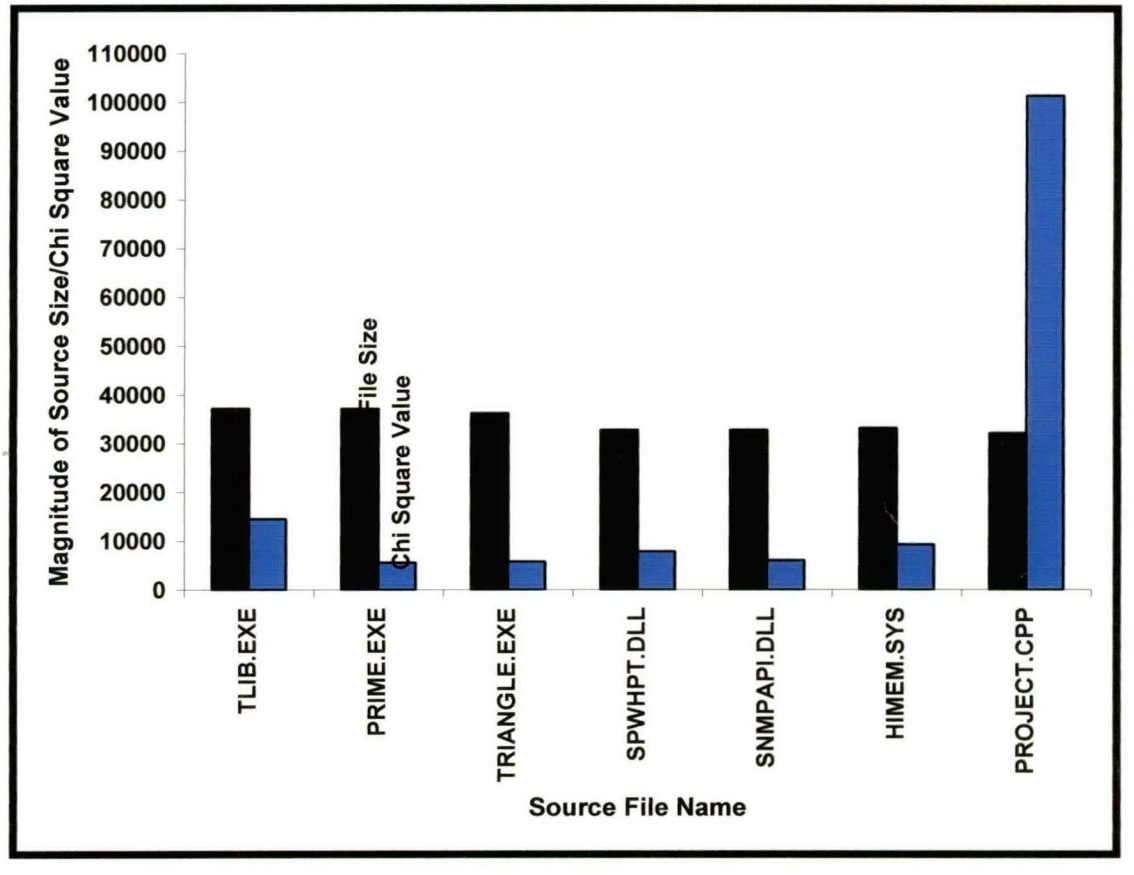

Figure 2.5.3.2
Graphical Relationship between Chi Square Values of
Files of Different Categories but almost of Same Sizes

Results of the frequency distribution tests presented in section 2.4.2 suggest the existence of a wide range of distribution of characters in all encrypted files, which is so effective in making it difficult to break the ciphertext using cryptanalysis.

One point to be noted in section 2.4.3 is that the better result is available in terms of the chi square values for lower block size. But only judging from this angle it cannot be concluded that it is a preferable option to choose smaller blocks, since having a unique size of blocks of only 8 bits makes the implementation of the encryption process so easy.

2.6 Conclusion

Analyzing the proposed RPSP technique from different perspectives, it can be pointed out that it is the structure of the scheme that helps in forming a large key space, which in turn helps ensuring the security of a very satisfactory level. If blocks of varying sizes are constructed from a source stream of bits, and for each block arbitrary number of iterations are performed during the process of encryption, and if all these information are accommodated in the secret key, then for ensuring the correct decryption, a reasonably long key space is required. One proposal for such a key is presented in figure 8.2.1.1 in chapter 8.

Also, section 2.5.1 shows the finiteness in regenerating the source block, which ensures that whatever is the size of a block, after a finite number of iterations it is regenerated, and since this finite number of iterations does not follow any mathematical policy it ensures a better security.

Therefore it can be concluded that the RPSP encryption policy itself can produce a satisfactory degree of information security, and, as it is discussed in chapter 9, when it participates in the cascaded approach of encryption, it further enhances the performance a lot.