CHAPTER 8

Bit-pair Operation and Separation (BOS)

8.1 Introduction

A new microprocessor-based block cipher has been proposed in this chapter in which the encryption is done through **B**it-pair **O**peration and **S**eparation (**BOS**). Like in other proposed algorithms, the plain-text in BOS is considered as a string of binary bits, which is then divided into blocks of $n = 2^k$ bits each, where k is 3, 4, 5, 6, and so on. Within each block, two adjacent bits are paired and two different operations are performed in each pair. The result of the first operation is placed at the front and that of the second operation at the rear. The encryption is started with block-size of 8 bits and repeated for several times and the number of iterations forms a part of the key. The whole process is repeated several times, doubling the block-size each time. The same process is used for decryption.

8.2 The BOS technique

Though the technique may be applied to larger string sizes also, a 512-bit binary string has been used as the plain-text in this implementation. The input string, S, is first broken into a number of blocks, each containing n bits where $n = 2^k$ and k may be one of 3,4,5,6,7,8,9, and so on, starting with $k = 3$. Hence, $S = S_1S_2S_3.....S_m$, where $m = 512/n$. The BOS operation is applied to each block. The process is repeated, each time doubling the block size till $n = 512$.

The decryption is carried out by reiterating the same algorithm. Section 8.2.1 explains the operations in detail.

8.2.1 The algorithm for BOS

After breaking the input stream into several blocks of size 8, the following operations are performed starting from the most significant side:

Round 1: In each block $S_i = (B_1,B_2,B_3,B_4,B_5,B_6,B_7,B_8)$, each pair of two adjacent bits are grouped together viz. (B_1,B_2), (B_3,B_4), (B_5,B_6), and (B_7,B_8). If the resultant block is denoted by $S'_i = (C_1,C_2,C_3,C_4,C_5,C_6,C_7,C_8)$, then the two proposed operations on the pair (B_1,B_2) will give two resultant bits which are then separately placed as C_1 and C_5. Similarly, the pair (B_3,B_4) will give C_2 and C_6. Symbolically, the pair (B_i,B_{i+1}) will give the bits $C_{(i+1)/2}$ and $C_{(i+1)/2+n/2}$ where n is the block size. The two operations are carried out according to the truth table given in table 8.1. Henceforth, the bits $C_1,C_2,C_3,\ldots\ldots,C_{n/2}$ will be called ***front bits*** and the bits $C_{(n/2)+1},C_{(n/2)+2},\ldots\ldots,C_n$ will be called ***rear bits***. Simply speaking, the result of the first operation is appended to the front bits and the result of the second operation is appended to the rear bits.

This round is repeated for a finite number of times and the number of iterations will form a part of the key, which will be discussed in chapter 11. Experiments have shown that, if the block-size is $n = 2^k$, then the original block is obtained after $3*k$ iterations. Some intermediate encrypted block may be taken as the cipher-text where the number of iterations is less than $3*k$.

Round 2: The same operations as in *Round 1* are performed with block-size 16.

In this fashion several rounds are completed till we reach ***Round 7*** where the block-size is 512 and we get the encrypted bit-stream.

During decryption, the remaining iterations out of $3*k$ iterations in each round are carried out for each block-size to get the original string, but the block-size is halved in each round starting from 512 down to 8, i.e. the reverse as that of encryption.

8.2.2 The bit-pair operations

As discussed in *Round 1*, the two operations are illustrated in Table 8.1. If the pair (B_i,B_{i+1}) is considered for the operations, then the resultant *front bit* will be $C_{(i+1)/2}$ and the *rear bit* will be $C_{(i+1)/2+n/2}$. If the two bits of the pair are same, then the *front bit* will be 0 else it will be 1. This is same as an XOR operation, but has been

implemented using IF-THEN-ELSE structure for C language program. If the bits are same and both are 0's then the *rear bit* will be 0 and if both are 1's then the *rear bit* will be 1. If the bits are different, then (0,1) will produce 0 whereas (1,0) will give 1 as the *rear bit*. A close study will reveal that whatever may be the front bit $C_{(i+1)/2+n/2} = B_i$. This observation is important for microprocessor-based implementation.

Table 8.1: Computation of front and rear bits

For Front Bit		For Rear Bit			
Bit Pair	Front Bit	Same Bits	Rear Bit	Different Bits	Rear Bit
Same	0	0,0	0	0,1	0
Different	1	1,1	1	1,0	1

8.3 Example of BOS

A single block of 8 bits, say S = 11011000, is considered here for an example. As discussed in section 8.2, the operations are performed to give the first encrypted block, say S^1. Figure 8.1 shows the formation of the front and rear bits.

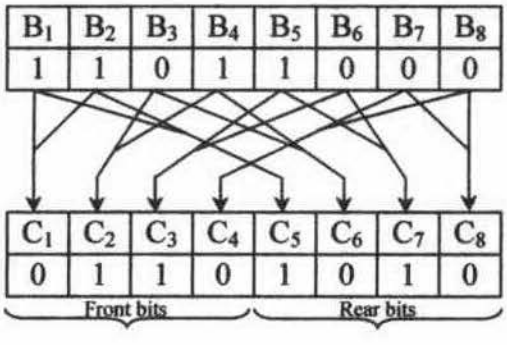


Figure 8.1: Computation of the Front and Rear bits in the first iteration.

The process is repeated to produce S^2, S^3, S^5, and so on. Since the block-size is 8, which is equal to 2^3, the final block will be S^9 (3*3=9), which is same as the original block S. Any one of the intermediate blocks, say S^4, may be considered as the cipher-text. Table 8.2 illustrates all the iterations needed to get back the original block for an 8-bit block. If m is the maximum number of iterations and S^i is the cipher-text, then the remaining $(m-i)$ iterations will be required during decryption to get back the original block. Hence a separate algorithm for decryption is not required for BOS. Moreover, the decryption key can be easily obtained from the encryption key.

Table 8.2: Iterations for an 8-bit block

Bit positions →	1	2	3	4	5	6	7	8
S (Original block)	1	1	0	1	1	0	0	0
S^1	0	1	1	0	1	0	1	0
S^2	1	1	1	1	0	1	1	1
S^3	0	0	1	0	1	1	0	1
S^4 (Cipher-text)	0	1	0	1	0	1	1	0
S^5	1	1	1	1	0	0	0	1
S^6	0	0	0	1	1	1	0	0
S^7	0	1	0	0	0	0	1	0
S^8	1	0	0	1	0	0	0	1
S^9 (Same as original)	1	1	0	1	1	0	0	0

Similarly, for a 16-bit string, the whole process will have two rounds, the first with block-size 8 and then with block-size 16. Round 1 can be repeated any number of times less than the maximum number of iterations, i.e. the number of iterations should be less than of 9 in Round 1. Similarly, in Round 2, since $16 = 2^4$, the number of iterations should be less than 3*4, i.e. 12. The number of iterations required to generate the original block for all block-sizes up to 512 have been listed in table 8.3.

Table 8.3: Number of iterations needed to form a cycle

Round	Block size	Maximum Iterations
1	8	09
2	16	12
3	32	15
4	64	18
5	128	21
6	256	24
7	512	27

8.4 Microprocessor-based implementation

In order to realize the scheme in an Intel 8085 microprocessor-based system, a 512-bit data is stored in memory (say FA00H onwards) and the routines for 8, 16, 32, 64, 128, 256, and 512 bits are applied, and the result is stored F900H onwards. The routine for 8-bit BOS is given in section 8.4.1 and the same for 16-bit block-size is given in section 8.4.2. The routines for higher block-sizes will be almost same as 16-bit BOS with a very few modifications. The modifications needed for each block-size have been listed in table 8.4.

8.4.1 Routine for 8-bit BOS encryption/decryption

This routine will perform 8-bit BOS on a 512-bit block stored in the memory from FA00H onwards and stores the result from F900H onwards. The stack is maintained from FB00H onwards.

Step 1 : Load SP to point to memory location FB00H
Step 2 : Load HL pair to point to memory location F900H
Step 3 : Push the content of HL pair into the stack
Step 4 : Load HL pair to point to memory location FA00H
Step 5 : Load A with 40H
Step 6 : Store the content of A into memory at FC00H
Step 7 : Load E with 04H
Step 8 : Clear B and C
Step 9 : Load A with the content of memory (location given by HL pair)
Step 10 : Rotate left without CARRY
Step 11 : Move the content of A to memory (location given by HL pair)
Step 12 : Move B to A
Step 13 : Rotate left with CARRY
Step 14 : Move A to B
Step 15 : Load A with the content of memory (location given by HL pair)
Step 16 : Rotate left without CARRY
Step 17 : Move the content of A to memory (location given by HL pair)
Step 18 : Move C to A
Step 19 XOR A with B
Step 20 : Rotate left with CARRY
Step 21 : Load D with 04H
Step 22 : Rotate left without CARRY
Step 23 : Decrement D
Step 24 : Repeat from step 22 till D is zero
Step 25 : OR A with B
Step 26 : Move A to D
Step 27 : Swap HL pair with stack top
Step 28 : Load A with the content of memory (location given by HL pair)
Step 29 : Rotate left without CARRY
Step 30 : OR A with D
Step 31 : Move the content of A to memory (location given by HL pair)
Step 32 : Swap HL pair with stack top
Step 33 : Decrement E
Step 34 : Repeat from step 8 till E is zero
Step 35 : Increment HL pair
Step 36 : Swap HL pair with stack top
Step 37 : Increment HL pair
Step 38 : Swap HL pair with stack top
Step 39 : Load A from FC00H
Step 40 : Decrement A
Step 41 : Store the content of A into memory at FC00H
Step 42 : Repeat from step 7 till A is zero
Step 43 : Return

8.4.2 Routine for 16-bit (and higher) BOS encryption/decryption

The same assumptions as in 8-bit BOS are made for this routine.

Step 1 : Load A with 20H
Step 2 : Store the content of A into memory at FC00H
Step 3 : Load A with 01H
Step 4 : Store the content of A into memory at FC01H
Step 5 : Load SP to point to memory location FB00H
Step 6 : Load HL pair to point to memory location F900H
Step 7 : Push the content of HL pair into the stack
Step 8 : Load HL pair to point to memory location FA00H
Step 9 : Load A with 01H
Step 10 : Store the content of A into memory at FC02H
Step 11 : Load E with 04H
Step 12 : Clear B and C
Step 13 : Load A with the content of memory (location given by HL pair)
Step 14 : Rotate left without CARRY
Step 15 : Move the content of A to memory (location given by HL pair)
Step 16 : Move B to A
Step 17 : Rotate left with CARRY
Step 18 : Move A to B
Step 19 : Load A with the content of memory (location given by HL pair)
Step 20 : Rotate left without CARRY
Step 21 : Move the content of A to memory (location given by HL pair)
Step 22 : Move C to A
Step 23 : Rotate left with CARRY
Step 24 : XOR A with B
Step 25 : Move A to D
Step 26 : Swap HL pair with stack top
Step 27 : Load A with the content of memory (location given by HL pair)
Step 28 : Clear Cy
Step 29 : Rotate left with CARRY
Step 30 : OR A with D
Step 31 : Move the content of A to memory (location given by HL pair)
Step 32 : Load A from FC01H
Step 33 : Increment HL pair
Step 34 : Decrement A
Step 35 : Repeat from step 33 till A is zero
Step 36 : Load A with the content of memory (location given by HL pair)
Step 37 : Clear Cy
Step 38 : Rotate left with CARRY
Step 39 : OR A with B
Step 40 : Load A from FC01H
Step 41 : Decrement HL pair
Step 42 : Decrement A
Step 43 : Repeat from step 41 till A is zero
Step 44 : Swap HL pair with stack top

Step 45 : Decrement E
Step 46 : Repeat from step 12 till E is zero
Step 47 : Load A from FC02H
Step 48 : Increment HL pair
Step 49 : Decrement A
Step 50 : Store the content of A into memory at FC02H
Step 51 : Repeat from step 11 till A is zero
Step 52 : Load A from FC01H
Step 53 : Add A to A (to make it double)
Step 54 : Swap HL pair with stack top
Step 55 : Increment HL pair
Step 56 : Decrement A
Step 57 : Repeat from step 55 till A is zero
Step 58 : Swap HL pair with stack top
Step 59 : Load A from FC00H
Step 60 : Decrement A
Step 61 : Store the content of A into memory at FC00H
Step 62 : Repeat fro step 9 till A is zero
Step 63 : Return

Table 8.4: Amendments for BOS with higher block-sizes

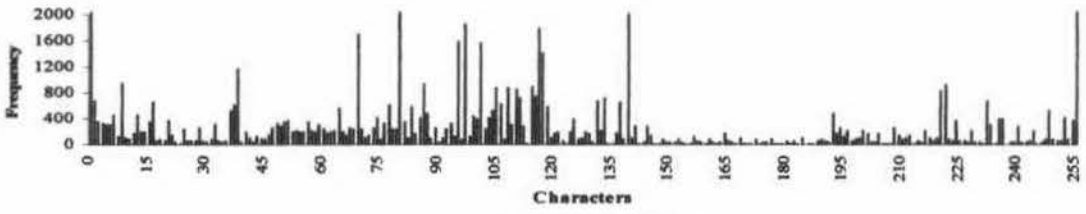
Steps	To be changed	Block-size				
		32 bit	64 bit	128 bit	256 bit	512 bit
Step 1	20H	10H	08H	04H	02H	01H
Step 4	01H	02H	04H	08H	10H	20H
Step 9	01H	02H	04H	08H	10H	20H

8.5 Results and comparisons

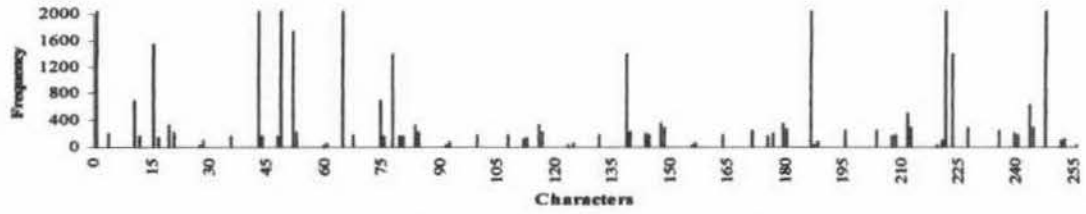
BOS was tested using the methods already discussed in section 1.8.3. Just one pass (no iterations) in each round was used to test it in its weakest form, so that the strength of the BOS algorithm may increase during actual implementation. The results of the tests have been compared with those of Triple DES. As in previous cases, the same files, five each in four different categories, namely .dll, .exe, .jpg and.txt, were taken for encryption using BOS and Triple DES for the purpose of testing.

8.5.1 Character frequency

Among the twenty files encrypted, the results of just one file in each category are shown here for the sake of brevity. Figures 8.2 through 8.5 show the frequencies of all the 256 characters in the source and the encrypted files of all four categories.



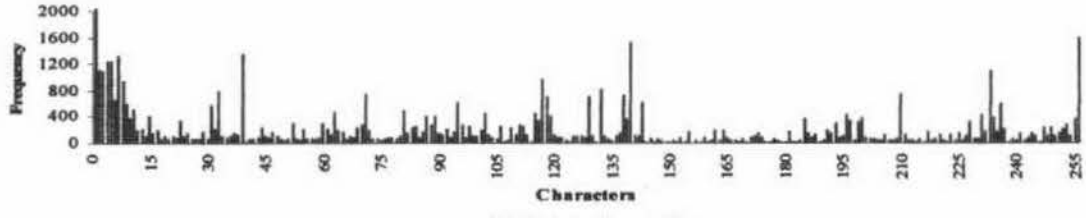
(a) Original .dll file



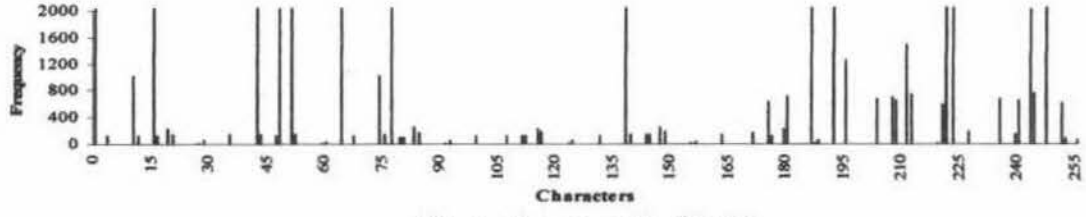
(b) .dll file encrypted with BOS

(c) .dll file encrypted with Triple DES

Figure 8.2: Character-frequencies in the source and encrypted .dll files



(a) Original .exe file

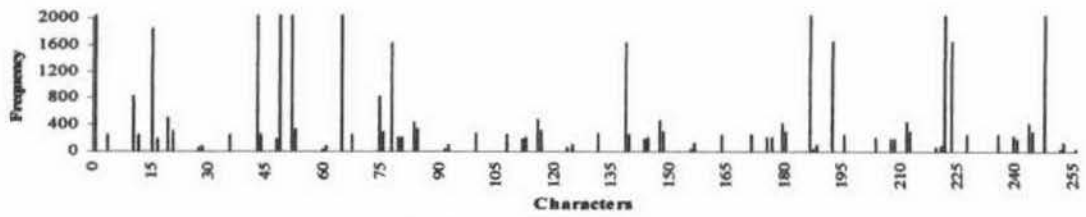


(b) .exe file encrypted with BOS

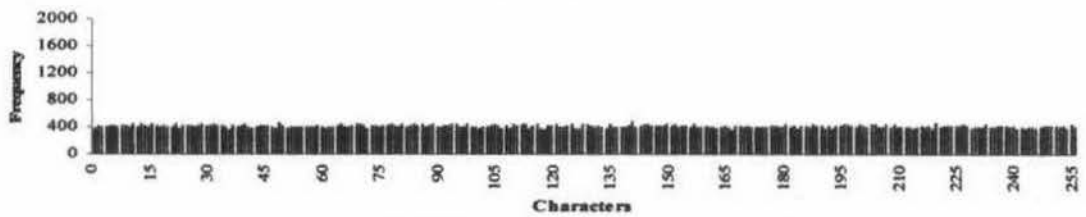
(c) .exe file ecrypted with Triple DES

Figure 8.3: Character-frequencies in the source and encrypted .exe files

(a) Original .jpg file

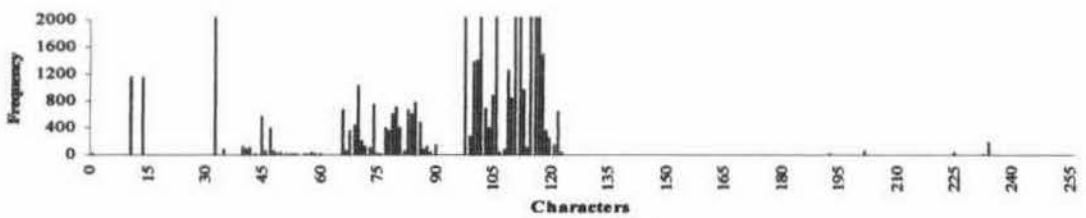


(b) .jpg file encrypted with BOS

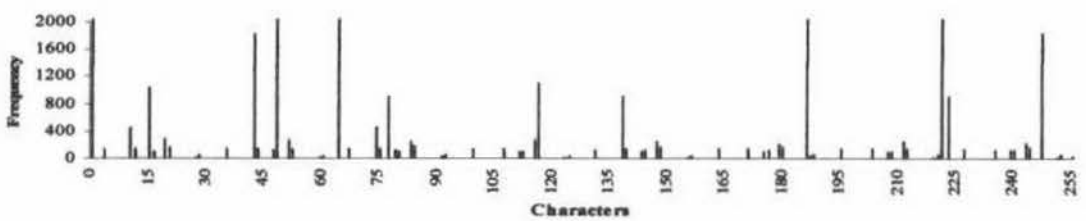


(c) .jpg file encrypted with Triple DES

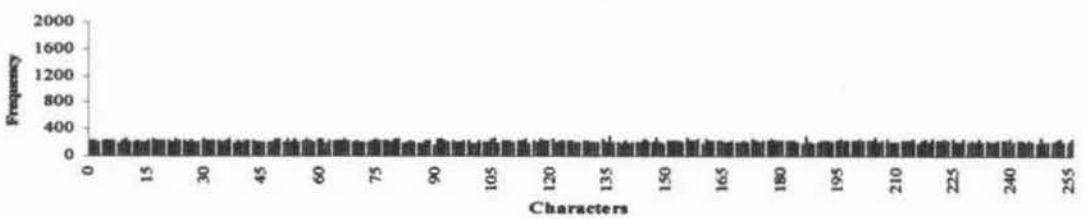
Figure 8.4: Character-frequencies in the source and encrypted .jpg files



(a) Original .txt file



(b) .txt file encrypted with BOS



(c) .txt file encrypted with Triple DES

Figure 8.5: Character-frequencies in the source and encrypted .txt files

Although very high frequencies of few characters in the graphs have been truncated to make the low values quite visible, some of the very low frequencies in the files encrypted with BOS are not still visible.

BOS has shown quite similar results for all four categories of files. Hence, it is not necessary to explain its performance in each category separately. It may seem that BOS has not shown a good performance with regard to character frequency distribution, but a close look at the graphs will tell a different story. The frequencies of all the 256 characters have changed a lot during encryption in all the four cases. Some characters with very low frequencies in the original file have shown high frequencies in the encrypted files. Likewise, some characters with high frequencies have changed to quite low frequencies. It should be noted that the most important plus point for BOS is that it has shown the same type of performance for all categories of files, which is not true for other proposed algorithms. The results for BOS are not that bad compared to Triple DES.

8.5.2 Chi-Square test and encryption time

As usual, the χ^2-test was performed to check the heterogeneity between the original and encrypted pairs of all the twenty files. The χ^2 values and encryption times due to BOS were also compared with those of Triple DES. Each category of files has been dealt with separately. The comparative χ^2 values and encryption times for BOS along with those for Triple DES in case of .dll files are listed in table 8.5. The comparisons in table 8.5 can be visualised in figure 8.6.

Table 8.5: χ^2-test for BOS with .dll files

Sl. No.	Original file	File size (bytes)	BOS			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.dll	20480	0.109890	13177	125	06	29790	255
2	2.dll	53312	0.219780	46062	237	16	43835	255
3	3.dll	90176	0.329670	311218	246	26	66128	255
4	4.dll	118784	0.384615	415291	255	34	1211289	255
5	5.dll	204800	0.714286	1569635	255	69	2416524	255

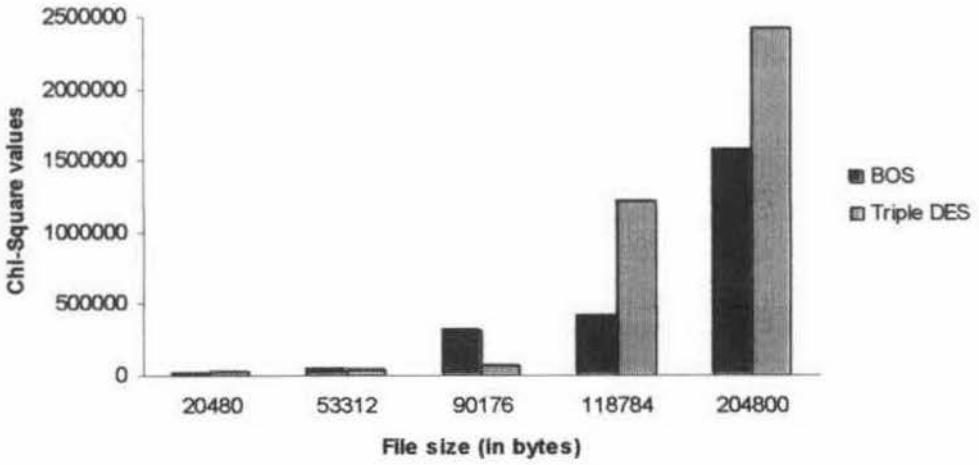


Figure 8.6: BOS vs. Triple DES in χ^2-test of .dll files

For .dll files, the performance of BOS in χ^2-test is quite comparable to Triple DES. While the degrees of freedom (DF) for some files are bit low, the corresponding χ^2 values for BOS are quite high. For large files also, they are not so low compared to Triple DES. Very small encryption time and large χ^2 values indicate the strength of BOS. The results of the test for .exe files are given in table 8.6 and figure 8.7.

Table 8.6: χ^2-test for BOS with .exe files

Sl. No.	Original file	File size (bytes)	BOS			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.exe	23104	0.109890	30676	250	12	8772	255
2	2.exe	52736	0.164835	46235	227	15	43426	255
3	3.exe	131136	0.384615	2321449	250	29	986693	255
4	4.exe	170496	0.494505	1958490	255	49	475893	255
5	5.exe	200832	0.604396	2004973	252	58	1847377	255

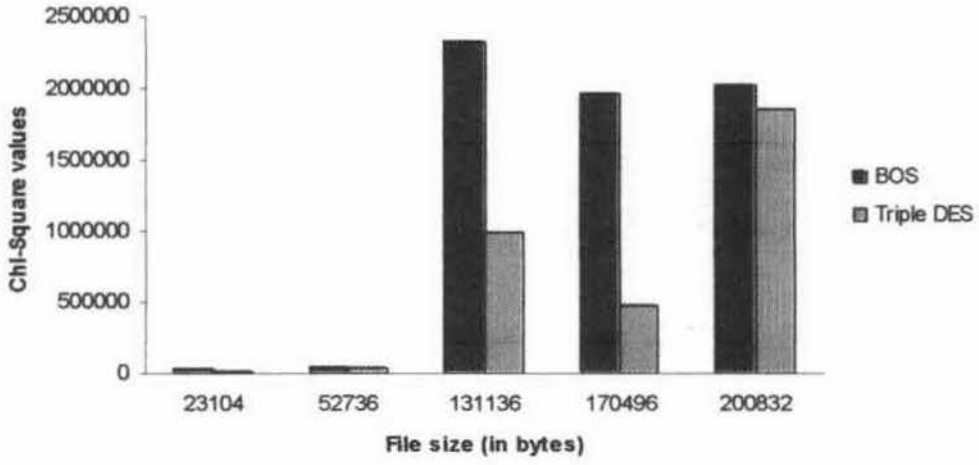


Figure 8.7: BOS vs. Triple DES in χ^2-test of .exe files

The test results of BOS for .exe files are much better than those for .dll files. In fact, in case of .exe files, the χ^2 values for BOS are even better than Triple DES. Moreover, the encryption times are much less than that of Triple DES. The test results for .jpg files are listed in table 8.7 and illustrated by figure 8.8.

Table 8.7: χ^2-test for BOS with .jpg files

Sl. No.	Original file	File size (bytes)	BOS			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.jpg	28544	0.109890	41118	255	08	4331	255
2	2.jpg	71232	0.219780	101874	255	21	2916	255
3	3.jpg	105600	0.384615	514189	255	31	5227	255
4	4.jpg	160704	0.494505	531483	255	47	22314	255
5	5.jpg	216576	0.714286	3198284	255	63	29824	255

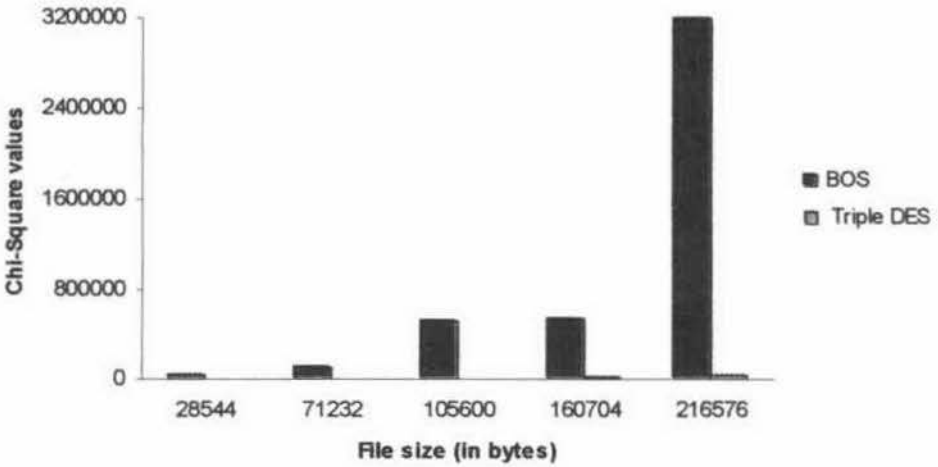


Figure 8.8: BOS vs. Triple DES in χ^2-test of .jpg files

The results of BOS for .jpg are even better than .exe files. The χ^2 values for BOS are much higher than Triple DES. The results for .txt files are given by table 8.8 and figure 8.9.

Table 8.8: χ^2-test for BOS with .txt files

Sl. No.	Original file	File size (bytes)	BOS			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	t1.txt	6976	0.054945	5158	66	02	10629	183
2	t2.txt	23808	0.109890	42571	114	07	32638	255
3	t3.txt	58688	0.164835	105767	130	17	82101	255
4	t4.txt	118784	0.329670	409852	134	35	170557	255
5	t5.txt	190784	0.549451	7499376	131	55	430338	255

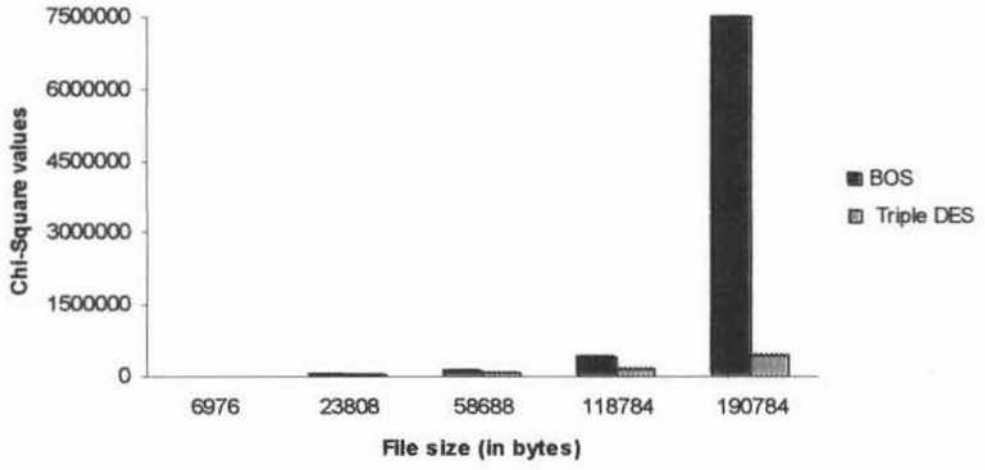


Figure 8.9: BOS vs. Triple DES in χ^2-test of .txt files

The nature shown by BOS for .txt files is quite similar to that shown for .jpg files and, hence, needs no further explanations. To sum up, high χ^2 values with high degrees of freedom (DF) along with very small encryption time compared to Triple DES makes BOS very much feasible for implementation into the intended targets.

8.5.3 Avalanche and runs

This test has been performed to examine the effect created in the cipher-text by a small change in the plain-text. To do this, a 32-bit binary string was repeatedly encrypted using BOS, first keeping the original string unaltered, and subsequently each time complementing one bit of the plain-text. The differences between the cipher-texts were examined and the number of runs was also computed in each pair of plain-text and the corresponding cipher-text. The difference of runs in each plain-text/cipher-text pair was noted. The results of this test for BOS are listed in table 8.9.

BOS has shown a very good performance in this test. The table reveals that BOS can produce a good amount of effect in the cipher-text with a very small change in the plain-text, which is evident from the difference between any two consecutive cipher-texts. Hence BOS causes a sufficient amount of diffusion and, in addition, there are a lot of differences in runs between the plain-text and the cipher-text in most of the cases.

Table 8.9: Avalanche and runs in BOS

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	284EAA66	19	21	2
1ST	C145D450	E44E66C6	18	16	2
2ND	0145D450	A04E22C6	17	16	1
3RD	6145D450	E84E6AC6	19	18	1
4TH	5145D450	A84E2AC6	21	20	1
5TH	4945D450	E44EAAC6	21	20	1
6TH	4545D450	A04EAAC6	21	20	1
7TH	4345D450	E84EAAC6	19	20	1
8TH	4045D450	A84EAAC6	17	22	5
9TH	41C5D450	2C4EAAC6	17	21	4
10TH	4105D450	0A4E88C6	17	17	0
11TH	4165D450	284EAAC6	19	21	2
12TH	4155D450	084E8AC6	21	17	4
13TH	414DD450	2C4EAAC6	19	21	2
14TH	4141D450	0A4EAAC6	17	21	4
15TH	4147D450	284EAAC6	17	21	4
16TH	4144D450	084EAAC6	19	21	2
17TH	41455450	2882AA0A	21	21	0
18TH	41459450	28C6AA4E	19	21	2
19TH	4145F450	288EAA06	17	19	2
20TH	4145C450	28CEAA46	17	21	4
21ST	4145DE50	2882AAC6	17	21	4
22ND	4145D050	28C6AAC6	17	21	4
23RD	4145D650	288EAAC6	19	21	2
24TH	4145D550	28CEAAC6	21	21	0
25TH	4145D4D0	288EAAA6	19	23	4
26TH	4145D410	286CAAE4	17	21	4
27TH	4145D470	288EAAA6	17	23	6
28TH	4145D440	286EAAE6	17	21	4
29TH	4145D458	288EAAC6	19	21	2
30TH	4145D454	286CAAC6	21	21	0
31ST	4145D452	288EAAC6	21	21	0
32ND	4145D451	286EAAC6	20	21	1

8.6 Conclusion

The proposed scheme takes very little time to encode and decode though the block length is high. No overhead bits are generated within the encoded string. The block length may be further increased beyond 512 bits with very little modifications in the algorithms, which may enhance security. Selecting the bit pairs in random order, rather than taking adjacent ones, may also enhance security. Due to its strength and simplicity, BOS may be very much feasible for implementation into the intended targets.