Block Exchange Technique (BET)

3.1 Introduction

A new microprocessor-based block cipher has been proposed in this chapter in which the encryption is done through **B**lock **E**xchange **T**echnique (**BET**). Like in the previous proposed algorithms, the plain-text in BET is considered as a string of binary bits, which is then divided into blocks of 8, 16, 32, 64, 128, 256, and 512 bits. Each block is then divided into four sets of bits. For example, when the block-size is 16 bits, then each set within a block will have 4 bits. Among the four set of bits, two middle sets are exchanged, i.e. set 2 and set 3 of the four sets are swapped. During the exchange, the corresponding bits in each set are swapped. 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, till it reaches 512 bits. The same process is used for decryption.

3.2 The BET scheme

A 512-bit binary string has been used as the plain-text in this implementation, but the technique may be applied to larger string sizes also. The input string, S, is first broken into a number of blocks, each containing n bits. Hence, $S = S_1S_2S_3.....S_m$, where $m = 512/n$. Starting with $n = 8$, the BET operation is applied to each block. The process is repeated, each time doubling the block size till $n = 512$. As in the previous algorithms, the original string can be obtained by reiterating the rounds. Hence the same process is followed for decryption. Section 3.2.1 explains the rounds of BET in detail.

3.2.1 The Algorithm for BET

After breaking the input string of 512 bits into blocks of size 8, the following operations are performed.

Round 1: The block $S_i = (B_1,B_2,B_3,B_4,B_5,B_6,B_7,B_8)$, is divided into four sets of bits, each containing 2 bits, viz. (B_1,B_2), (B_3,B_4), (B_5,B_6), and (B_7,B_8). The two middle blocks are then swapped bit-by-bit. As a result B_3 is swapped with B_5, and B_4 is swapped with B_6. Hence, a new block $S'_i = (B_1,B_2,B_5,B_6,B_3,B_4,B_7,B_8)$ is formed by the swapping process. This new block is used as the input to the next round. 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.

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***, in which the block-size is 512 and we get the encrypted bit-stream.

During decryption, the remaining out of the maximum number of iterations in each round to form a cycle, 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.

3.3 Example of BET

Since a large plain-text will make things complicated, only a 32-bit string is considered for illustration. Further, the characters a, b, c etc. are used in place of 0's and 1's to visualise the movement of bits properly. The process of encryption for the plain-text S = abcdefghijklmnopqrstuvwxyzϕλπθξψ is shown in a stepwise manner.

Round 1: Block-size = 8, number of blocks = 4

Input:

B_1	B_2	B_3	B_4
abcdefgh	ijklmnop	qrstuvwx	yzϕλπθξψ

Output:

B_1	B_2	B_3	B_4
abefcdgh	ijmnklop	qruvstwx	yzπθϕλξψ

Round 2: Block-size = 16, number of blocks = 2

Input:

B_1	B_2
abefcdghijmnklop	qruvstwxyzπθϕλξψ

Output:

B_1	B_2
abefijmncdghklop	qruvyzπθstwxϕλξψ

Round 3: Block-size = 32, number of blocks = 1

Input:

B_1
abefcdghijmnklopqruvstwxyzπθϕλξψ

Output:

B_1
abefcdghqruvstwxijmnklopyzπθϕλξψ

Since only 32-bit string has been considered, it is not possible to proceed further and just three rounds are performed. The output from *Round 3*, say S', is the encrypted stream, i.e. S' = abefcdghqruvstwxijmnklopyzπθϕλξψ.

3.3.1 A discussion

The original string will be regenerated after 4 iterations for the string in the above example. To compute the number of iterations to form a cycle for each block-size, a string of 512 bits is taken and encrypted repeatedly with a particular block size, each time comparing the string generated with the original one. If the original and the encrypted strings are different, the encryption is iterated again, and if they are same, the number of iterations for that block-size is noted. Table 3.1 shows the number of iterations required for different block-sizes to complete a cycle.

Table 3.1: Number of iterations for a complete cycle

Block-size	8	16	32	64	128	256	512
Maximum iterations required for a cycle	2	4	4	8	8	8	8

It is evident from the table that if the block-size is increased, the number of iterations to complete a cycle also increases, but no direct relation between the block-size and the number of iterations could be established for the proposed algorithm.

3.4 Microprocessor-based implementation

An efficient algorithm has been developed considering the mapping of bits rather than following the actual swapping procedure. The algorithm is faster and suitable for implementation using an Intel 8085 microprocessor-based system. A 512-bit data is assumed to be stored in memory from some location onwards. The memory allocations for the implementation of the algorithm are given in the respective sections.

3.4.1 Routines for block-size 8 bits

For the routines implementing 8-bit round of the proposed algorithm, the various blocks of the main memory are reserved for program, data, tables etc. as follows:

Program area : F800H onwards
Data area : F900H onwards
Result area : FA00H onwards
Table area (for look-up tables) :
Table 1 : FB00H onwards (for number of iterations)
Table 2 : FC00H onwards (for type of iterations)
Table 3 : FD00H onwards (for masking information)
Stack area : FFA0H

The main routine for 8-bit BET is given in section 3.4.1.1. This main routine calls the subroutine 'BET8' to apply the mapping on the binary string being considered. The subroutine 'BET8' takes the value in A as a parameter. It consults the various tables (table1 from FB00H onwards, table2 from FC00H onwards, table3 from FD00H onwards) and sets the corresponding bit in the result area from FA00H onwards.

3.4.1.1 Routines for 8-bit BET

Main routine:

Step 1 : Initialize SP with the highest memory location available
Step 2 : Load HL pair with F900H and DE pair with FA00H
Step 3 : Initialize some memory location as a counter with 40H
Step 4 : Load A with the content of memory (location given by HL pair)
Step 5 : Push HL and DE pairs into the stack
Step 6 : Call BET8
Step 7 : Pop HL and DE pairs from stack
Step 8 : Load A with the content of memory (location given by HL pair)
Step 9 : AND 10H with A
Step 10 : ADD B to A
Step 11 : Move A to B
Step 12 : Load A with the content of memory (location given by HL pair)
Step 13 : AND 01H with A
Step 14 : ADD B to A
Step 15 : Store the content of A into memory location pointed to by DE pair
Step 16 : Increment both HL and DE pairs
Step 17 : Decrement the counter variable in memory
Step 18 : Repeat from step 4 till the counter is zero
Step 19 : Return

Subroutine 'BET8':

Step 1 : Load HL pair with FB00H and DE pair with FD00H
Step 2 : Move 00H to B
Step 3 : Initialize some memory location as a counter with 06H
Step 4 : Load C with the content of memory (location given by HL pair)
Step 5 : Increment H (not HL pair)
Step 6 : Load A with the content of memory (location given by HL pair)
Step 7 : Rotate right without CARRY
Step 8 : If Cy=1 then rotate right else rotate left without CARRY
Step 9 : Swap HL pair with DE pair
Step 10 : Rotate A either left or right depending on the step 8
Step 11 : Decrement C
Step 12 : Repeat from step 8 till C is zero
Step 13 : Add B to A
Step 14 : Move A to B
Step 15 : Decrement H (not HL pair)
Step 16 : Increment both HL and DE pairs
Step 17 : Decrement the counter variable in memory
Step 18 : Repeat from step 4 till the counter is zero
Step 19 : Return

Look-up Tables

Table 1 (FB00H onwards): 03H, 01H, 02H, 02H, 01H, 03H

Table 2 (FC00H onwards): 00H, 01H, 00H, 01H, 00H, 01H

Table 3 (FD00H onwards): 02H, 04H, 08H, 10H, 20H, 40H

3.4.2 Routines for block-size 16 bits (and higher)

The routines of BET implementing rounds for 16-bit and higher block-sizes need the various blocks of the main memory to be reserved for program, data, tables etc. as follows:

Program area : F800H onwards
Data area : F900H onwards
Result area : FA00H onwards
Table area (for look-up tables) :
  Table 1 : FB00H onwards (for bit mapping)
  Table 2 : FC00H onwards (for counter values)
Stack area : FFA0H

The routines needed for 16-bit block-size are listed section 3.4.2.1. The main routine calls several routines to apply the mapping on the binary string being considered.

3.4.2.1 Routines for 16-bit (and higher) BET

Main routine:

Step 1 : Load BC pair with F900H
Step 2 : Load D with 02H (i.e. block-size/8)
Step 3 : Initialize SP with the highest memory location available
Step 4 : Push HL pair, BC pair, DE pair and PSW into stack
Step 5 : Call OPPR
Step 6 : Load HL pair with FC00H
Step 7 : Load A with the content of memory (location given by HL pair)
Step 8 : Increment A 02H times
Step 9 : Move the content of A to memory (location given by HL pair)

Step 10 : Pop HL pair, BC pair, DE pair and PSW from stack
Step 11 : Increment BC pair the no of times as the value in A
Step 12 : Decrement D
Step 13 : Repeat from step 3 till D is zero
Step 14 : Return

Subroutine 'OPPR':

This routine checks each an every individual bit of the input data, and depending on the bit, it calls subroutine 'BSTR' for the conversion from input data stream to output data stream.

Step 1 : Clear L and load D with 02H (for 16-bit block-size)
Step 2 : Load E with 08H and H with 01H
Step 3 : Load A with the memory content pointed to by BC
Step 4 : AND A with H
Step 5 : If A is not zero then call BSTR
Step 6 : Increment L
Step 7 : Move H to A
Step 8 : Rotate A right without CARRY
Step 9 : Move A to H
Step 10 : Decrement E
Step 11 : Repeat from step 3 till E is zero
Step 12 : Increment BC pair
Step 13 : Decrement D
Step 14 : Repeat from step 2 till D is zero
Step 15 : Return

Subroutine 'BSTR':

This routine generates the result by consulting Table1 (from FB00H onwards) and sets the corresponding bit into the result area from FA00H onwards.

Step 1 : Push HL pair, BC pair, DE pair and PSW into stack
Step 2 : Load A from FC00H
Step 3 : Move A to B
Step 4 : Load H with 0FBH
Step 5 : Load A with the content of memory (location given by HL pair)
Step 6 : Subtract 08H from A
Step 7 : If Cy=1 jump to step 10
Step 8 : Increment B
Step 9 : Repeat from step 6
Step 10 : Add 08H to A
Step 11 : Move A to C

Step 12 : Load H with 0FAH
Step 13 : Move B to L
Step 14 : Load A with the content of memory (location given by HL pair)
Step 15 : Compare A with 00H
Step 16 : If Z=0 jump to 22
Step 17 : Load A with 01H
Step 18 : Rotate A right without CARRY
Step 19 : Decrement C
Step 20 : If Z=0 repeat from step 19
Step 21 : Jump to step 24
Step 22 : Load A with 01H
Step 23 : OR A with M
Step 24 : Move the content of A to memory (location given by HL pair)
Step 25 : Pop HL pair, BC pair, DE pair and PSW from stack
Step 26 : Return

Look-up Tables

Table 1 (FB00H onwards): 00H, 08H, 01H, 09H, 02H, 0AH, 03H, 0BH,
04H, 0CH, 05H, 0DH, 06H, 0EH, 07H, 0FH

Table 2 (FC00H onwards): 00H

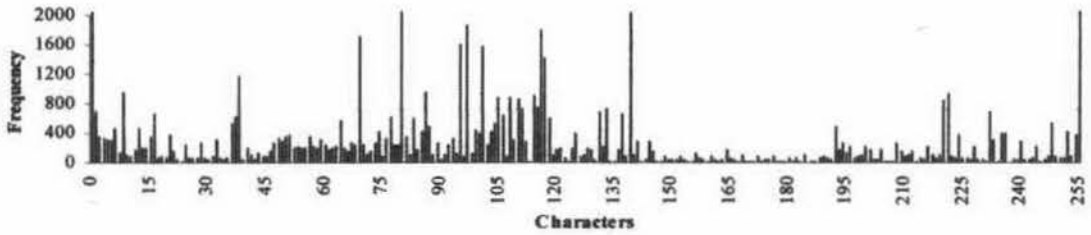
The routines for higher block-sizes, like 32, 64, 128 etc., are almost same with a slight modification in each case, and hence, not listed here.

3.5 Results and comparisons

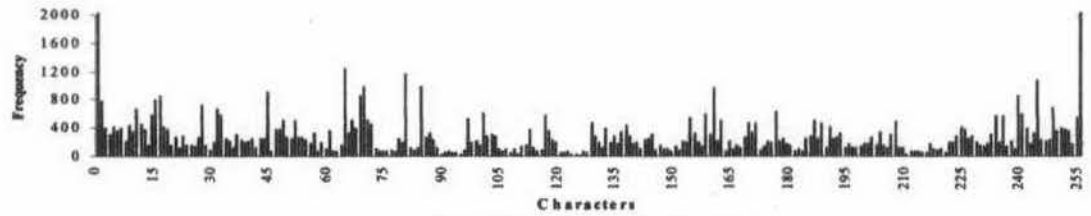
The methods already discussed in section 1.8.3 were used to test the strength of BET. Just one pass (no iterations) in each round was used to test it in its weakest form, so that its strength may increase during actual implementation. Triple DES has been used as a benchmark to compare the results of BET. As in previous cases, the same set of files, five in each of the four categories, namely .dll, .exe, .txt and .jpg, were taken for encryption using BET and Triple DES for the purpose of testing.

3.5.1 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 3.1 through 3.4 show the frequencies of all the 256 characters in the source and the encrypted files of all four categories.



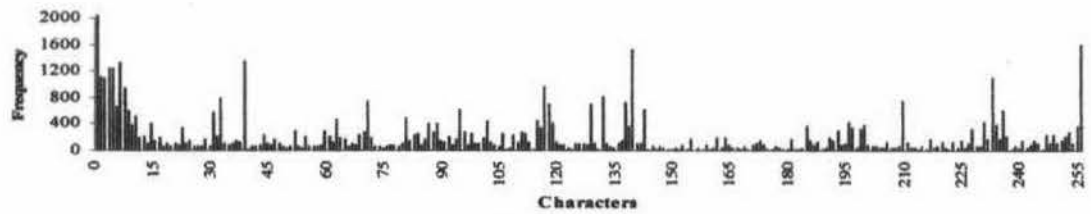
(a) Original .dll file



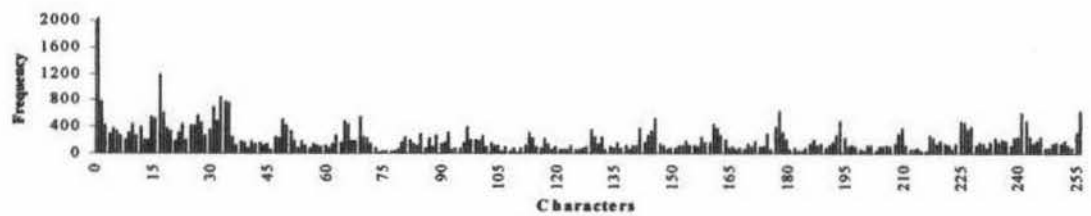
(b) .dll file encrypted with BET

(c) .dll file encrypted with Triple DES

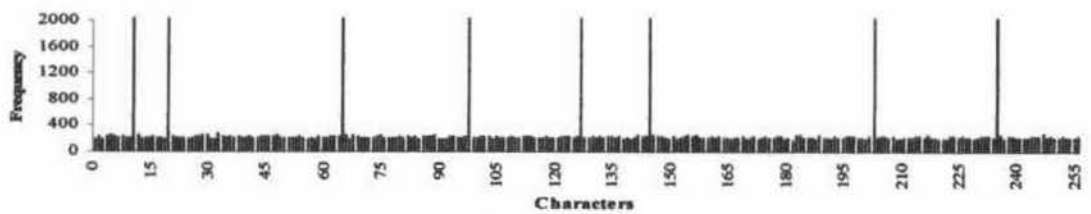
Figure 3.1: Character-frequencies in the source and encrypted .dll files



(a) Original .exe file



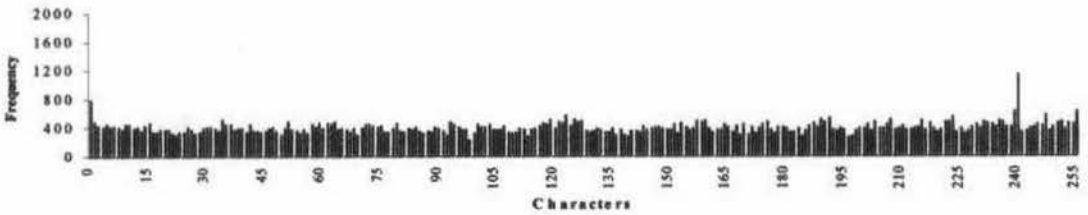
(b) .exe file encrypted with BET



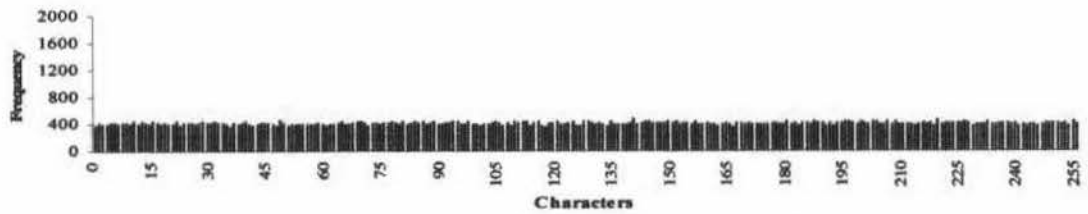
(c) .exe file ecrypted with Triple DES

Figure 3.2: Character-frequencies in the source and encrypted .exe files

(a) Original .jpg file

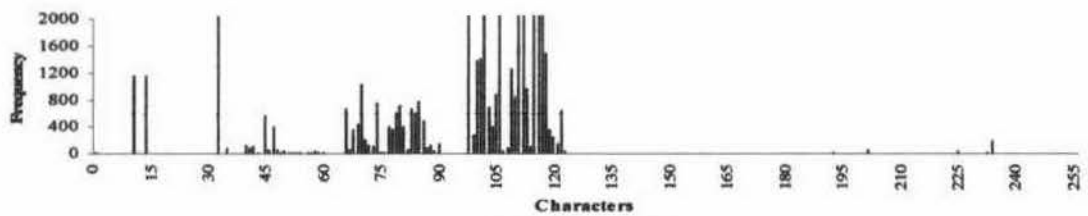


(b) .jpg file encrypted with BET

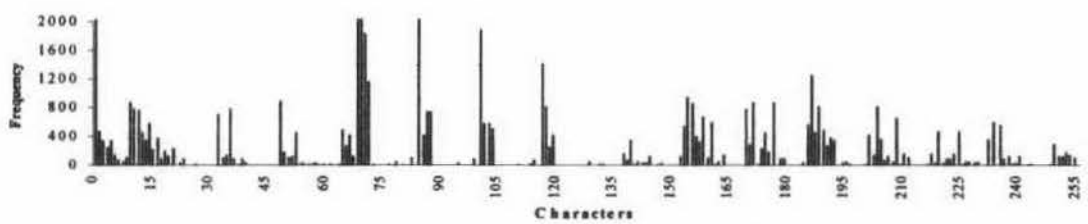


(c) .jpg file encrypted with Triple DES

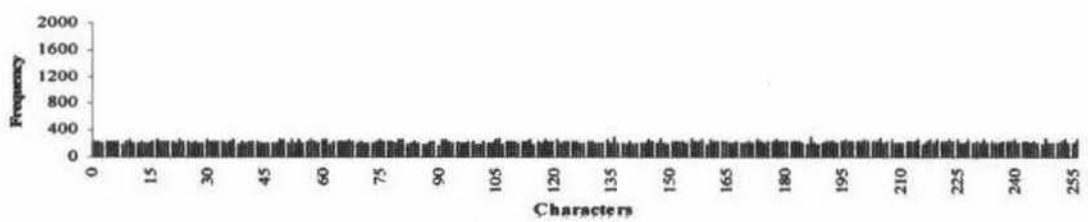
Figure 3.3: Character-frequencies in the source and encrypted .jpg files



(a) Original .txt file



(b) .txt file encrypted with BET



(c) .txt file encrypted with Triple DES

Figure 3.4: Character-frequencies in the source and encrypted .txt files

Some very high frequencies of few characters in the graphs have been truncated to make the low values quite visible. If this is not done, then the very low frequencies would not figure in the graphs and may be thought of as zero frequencies.

The characters in the original .dll file are clustered in some regions and almost negligible in some portions. After encryption with BET the characters are more or less evenly distributed throughout the character space. The characters with ASCII value 0 and 255 are the only exceptions, which have quite high frequencies and are truncated in the figure. In the result of Triple DES, most of the characters are distributed evenly in the character space, but some of them have abruptly high frequencies.

Similar explanations hold true for the .exe file also, because the results are almost similar to that of the .dll file. Even then the performance of BET is bit better fro .exe file than .dll file as is evident from the figure.

The distribution of characters in the .jpg file encrypted with BET is very good, as is evident from the frequency graphs. The frequencies obtained from BET and Triple DES are more or less the same. The characters are homogeneously distributed in the character space for both BET and Triple DES encrypted .jpg files.

Since the non-printable characters are absent in the original .txt file, the frequencies of almost half of the total 256 characters are nil. The characters are fairly distributed over the character space after the files are encrypted by BET and the same is comparable with Triple DES.

3.5.2 Chi-Square test and encryption time

To check the heterogeneity between the original and encrypted pairs of all the twenty files, the mostly used χ^2-test was performed. Higher the χ^2 values, more heterogeneous will be the two files being compared. The χ^2 values from the character frequencies and encryption times due to BET were also compared to those of Triple DES. Each category of files has been dealt with separately. The comparative χ^2 values and encryption times for BET along with those for Triple DES in case of .dll files are listed in table 3.2. The comparisons in the table can be visualised in figure 3.5.

Table 3.2: χ^2-test for BET with .dll files

Sl. No.	Original file	File size (bytes)	BET			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.dll	20480	0.054945	5471	169	06	29790	255
2	2.dll	53312	0.164835	17601	255	16	43835	255
3	3.dll	90176	0.274725	40798	255	26	66128	255
4	4.dll	118784	0.384615	1920600	255	34	1211289	255
5	5.dll	204800	0.604396	7376322	255	69	2416524	255

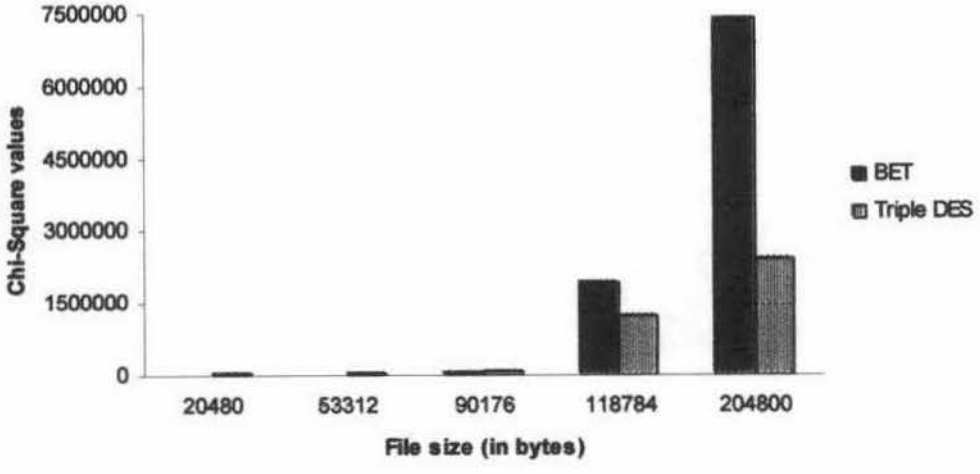


Figure 3.5: BET vs. Triple DES in χ^2-test of .dll files

For .dll files, the performance of BET in χ^2-test is quite good and comparable to Triple DES. The degree of freedom (DF) for all the files, except for one, is 255, and the corresponding χ^2 values are quite high. For large files, they are even higher than that of Triple DES. Very small encryption time and large χ^2 values indicate the strength of BET. The results of the test for the .exe files are given in table 3.3 and figure 3.6.

Table 3.3: χ^2-test for BET with .exe files

Sl. No.	Original file	File size (bytes)	BET			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.exe	23104	0.054945	8107	255	12	8772	255
2	2.exe	52736	0.164835	19221	255	15	43426	255
3	3.exe	131136	0.384615	899616	255	29	986693	255
4	4.exe	170496	0.549451	131095	255	49	475893	255
5	5.exe	200832	0.604396	2217990	255	58	1847377	255

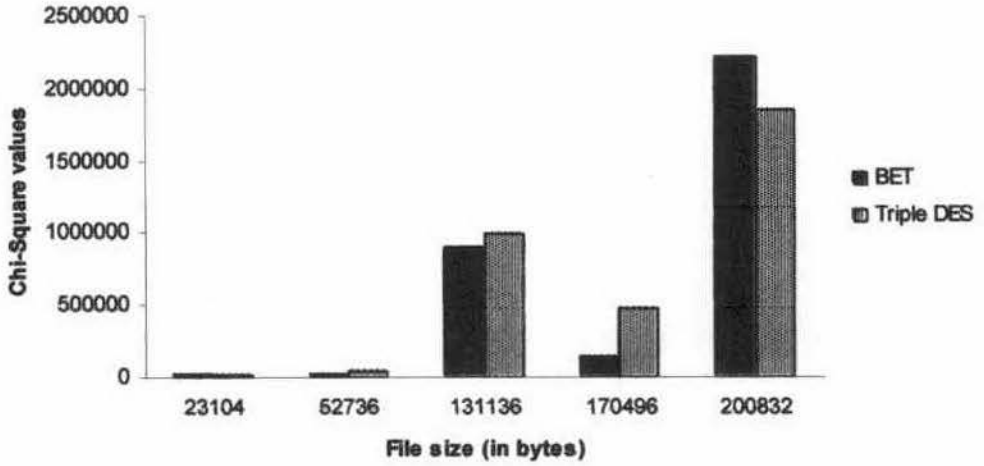


Figure 3.6: BET vs. Triple DES in χ^2-test of .exe files

The test results of BET for .exe files are almost similar to those for .dll files. In case of all the .exe files, except two files, the $\chi 2$ values for BET are at par with Triple DES. Moreover, the encryption times are much less than that of Triple DES. The test results for .jpg files are listed in table 3.4 and illustrated by figure 3.7.

Table 3.4: χ^2-test for BET with .jpg files

Sl. No.	Original file	File size (bytes)	BET			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.jpg	28544	0.054945	4196	255	08	4331	255
2	2.jpg	71232	0.219780	3604	255	21	2916	255
3	3.jpg	105600	0.329670	4724	255	31	5227	255
4	4.jpg	160704	0.439560	14853	255	47	22314	255
5	5.jpg	216576	0.659341	20862	255	63	29824	255

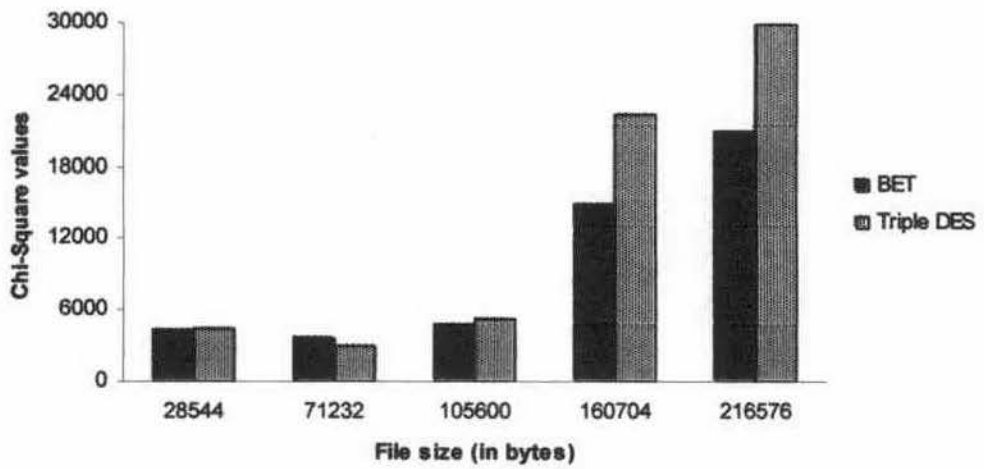


Figure 3.7: BET vs. Triple DES in χ^2-test of .jpg files

The results of BET for .jpg are not good as .dll and .exe files. Even then, they are not so low as compared to Triple DES. The values grow with the file size. The results for .txt files are given by table 3.5 and figure 3.8.

Table 3.5: χ^2-test for BET with .txt files

Sl. No.	Original file	File size (bytes)	BET			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	t1.txt	6976	0.054945	10028	85	02	10629	183
2	t2.txt	23808	0.109890	32962	151	07	32638	255
3	t3.txt	58688	0.219780	81943	178	17	82101	255
4	t4.txt	118784	0.384615	175305	193	35	170557	255
5	t5.txt	190784	0.549451	406733	194	55	430338	255

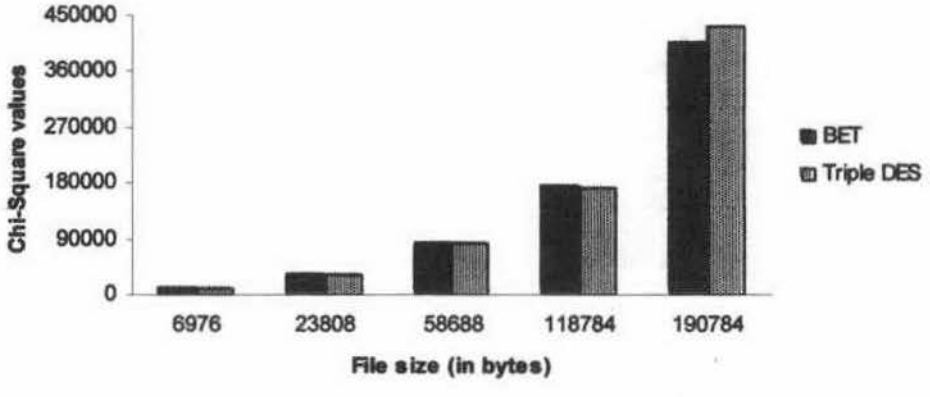


Figure 3.8: BET vs. Triple DES in χ^2-test of .txt files

The nature shown by BET for .txt files is very much similar to that shown by Triple DES. The $\chi 2$ values for BET are almost equal to those for Triple DES, although the degrees of freedom (DF) are bit less. High $\chi 2$ values along with very small encryption time compared to Triple DES makes BET very much feasible for implementation into the intended targets.

3.5.3 Avalanche and runs

As already done for PPO, a 32-bit binary string was repeatedly encrypted using BET, first keeping the original string unaltered, and subsequently each time complementing one bit of the plain-text. The plain-texts, corresponding cipher-texts and the number of runs along with their differences are listed in table 3.6.

Table 3.6: Avalanche and runs in BET

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	5445C150	19	18	1
1ST	C145D450	D445C150	18	17	1
2ND	0145D450	1445C150	17	16	1
3RD	6145D450	5445C150	19	20	1
4TH	5145D450	5445C151	21	19	2
5TH	4945D450	5445C158	21	18	3
6TH	4545D450	5445C154	21	20	1
7TH	4345D450	7445C150	19	16	3
8TH	4045D450	4445C150	17	16	1
9TH	41C5D450	5445C1D0	17	16	1
10TH	4105D450	5445C110	17	16	1
11TH	4165D450	5645C150	19	18	1
12TH	4155D450	5545C150	21	20	1
13TH	414DD450	5C45C150	19	16	3
14TH	4141D450	5045C150	17	16	1
15TH	4147D450	5445C170	17	16	1
16TH	4144D450	5445C140	19	16	3
17TH	41455450	54454150	21	20	1
18TH	41459450	54458150	19	18	1
19TH	4145F450	5447C150	17	16	1
20TH	4145C450	5444C150	17	18	1
21ST	4145DE50	544DC150	17	18	1
22ND	4145D050	5441C150	17	16	1
23RD	4145D650	5445E150	19	18	1
24TH	4145D550	5445D150	21	20	1
25TH	4145D4D0	54C5C150	19	18	1
26TH	4145D410	5405C150	17	16	1
27TH	4145D470	5445C350	17	18	1
28TH	4145D440	5445C050	17	16	1
29TH	4145D458	5445C950	19	20	1
30TH	4145D454	5495C550	21	20	1
31ST	4145D452	5465C150	21	21	0
32ND	4145D451	5455C150	20	20	0

BET not has shown a very good performance in this test. The difference between any two consecutive cipher-texts is not as large as desired, but the difference between any plain-text and the corresponding cipher-text is quite high. There are differences in runs between the plain-text and the cipher-text in most of the cases.

3.6 Conclusion

BET does not generate any overhead bits within the encoded string and it takes very little time to encode and decode though the block length is high. It can also be cascaded with a substitution cipher like OMAT (to be discussed later) to enhance its strength, which is discussed in chapter 10. Its strength and simplicity make BET very much feasible for implementation into the intended targets.