CHAPTER 9

Decimal Equivalent Positional Substitution (DEPS)

9.1 Introduction

In this chapter, another novel microprocessor-based block cipher has been proposed in which the encryption is based on a substitution scheme, with the preferred name **D**ecimal **E**quivalent **P**ositional **S**ubstitution (**DEPS**). The original string of 512 bits is divided into a number of blocks each containing n bits, where n is any one of 8, 16, 32, 64, 128, 256, or 512 in each round. The equivalent decimal integer of the block under consideration is computed and checked whether it is even or odd. A '0' or '1' is pushed into the output string depending on whether the integral value is even or odd, respectively. Then the position of this decimal integral value in the series of natural even or odd numbers is ascertained. The process is carried out recursively with the positional values for a finite number of times, equal to the length of the source block. For example, the process is repeated eight times for an 8-bit block to produce an output string of 8 bits. During decryption, bits in the target block are considered along LSB-to-MSB direction after which we get an integral value, the binary equivalent of which is the source block.

The sweetness of the technique lies in its microprocessor-based implementation where no calculation is needed to ascertain whether the decimal value is even or odd and to find its position in the series of odd or even numbers.

9.2 The DEPS scheme

The input to this cipher is considered as a 512 bit binary string. In ***Round 1***, the 512-bit plaintext is divided into 64 blocks 8 bits. The algorithm is then applied to each of the blocks. In a particular round, for each block $S = s_0\ s_1\ s_2\ s_3\ s_4\ \ldots\ s_{L-1}$ of length L bits, the scheme is followed in a stepwise manner to generate the target block $T = t_0\ t_1\ t_2\ t_3\ t_4\ \ldots\ t_{L-1}$ of the same length (L). The process is repeated in ***Round 2***, ***Round 3***, and so on, each time doubling the block-size, the input to a particular round

being the output of the previous round. The last round will be *Round 7* with block-size 512. The scheme can be best understood by the pictorial example given in figure 9.1, where the step-by-step approach of generating the target block corresponding to an 8-bit source block 11010110 using this technique has been nicely illustrated.

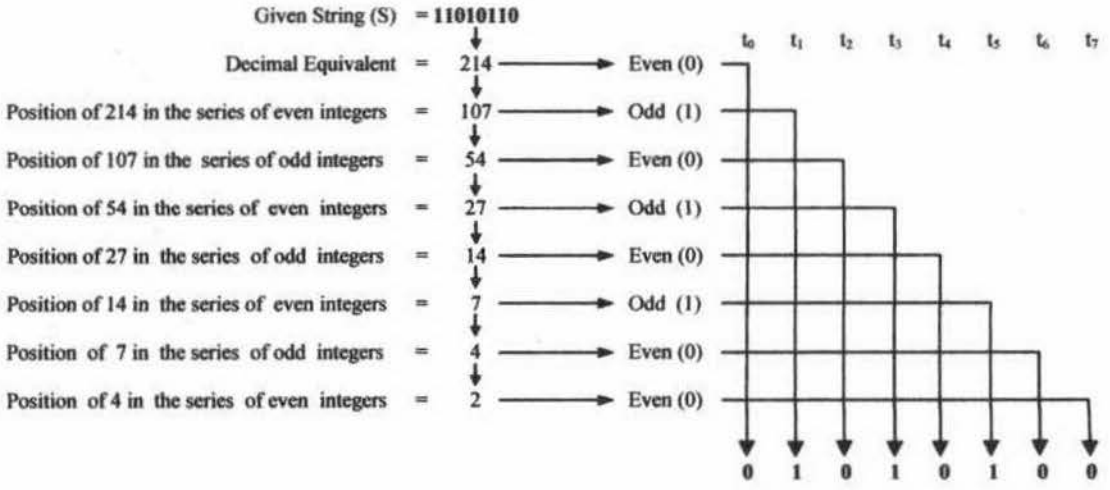


Figure 9.1: Cipher-text generation using DEPS on 8-bit data.

In the figure, the decimal equivalent of the given binary string is 214, which is an even number, and hence t_0=0. The position of 214 in the series of even integers is 214/2, i.e. 107, which is an odd number and hence t_1=1. The position of 107 in the series of odd integers is (107+1)/2, i.e. 54, which is an even number and hence t_3=0. The process is carried out eight times to get $T = t_0\, t_1\, t_2\, t_3\, t_4\, t_5\, t_6\, t_7$ since the length of the string is 8. The final value of T is 01010100. In general, if n is even, then its position in the series of even integers is $n/2$. If n is odd, then its position in the series of odd integers is $(n+1)/2$ or $\text{int}(n/2)+1$. Although it seems that lot of space and computations are needed to get the decimal equivalent of long binary strings, this requirement has been evaded in actual implementation by using an alternative method.

9.2.1 Algorithm for DEPS encryption

For a source block, $S = s_0\, s_1\, s_2\, s_3\, s_4 \ldots s_{L-1}$, of length L, the decimal equivalent D_L is computed and the target block, $T = t_0\, t_1\, t_2\, t_3\, t_4 \ldots t_{L-1}$, is generated by performing some steps. Pseudo-code has been used for the sake of brevity, so that the flow of control is quite clear.

set $P = 0$

LOOP: compute TEMP = remainder of $D_{L-P} / 2$

if TEMP = 0 then

compute $D_{L-P-1} = D_{L-P} / 2$

set: $t_P = 0$

else

compute $D_{L-P-1} = (D_{L-P} + 1) / 2$

set: $t_P = 1$

end if

set $P = P + 1$

if $P < (L - 1)$ then goto LOOP

end if

end

9.2.2 Algorithm for DEPS decryption

The encrypted message is decomposed into a finite set of blocks in the same manner as in encryption. For each encrypted block, $T = t_0\ t_1\ t_2\ t_3\ t_4 \ldots\ t_{L-1}$ of length L bits, the decryption algorithm is carried out to generate $S = s_0\ s_1\ s_2\ s_3\ s_4\ \ldots\ s_{L-1}$, the source block of the same length (L). Since in encryption, the position of an integer is computed, the opposite is done in decryption, i.e. the integer corresponding to a position is calculated.

set $P = L - 1$ and T = 1

LOOP: if $t_P = 0$ then

compute T = T^{th} number in the series of even integers

else

compute T = T^{th} number in the series of odd integers

end if

set $P = P - 1$

if $P \geq 0$ then goto LOOP

end if

compute $S = s_0\ s_1\ s_2\ s_3\ s_4\ \ldots\ s_{L-1}$, which is the binary equivalent of T

9.3 Example of DEPS

Considering the string "Local Area Network" as the plaintext (P), the corresponding string of bits (S) is obtained from the 8-bit ASCII code of each character, i.e. S = 01001100/01101111/01100011/01100001/01101100/00100000/
01000001/01110010/01100101/01100001/00100000/01001110/
01100101/01110100/01110111/01101111/01110010/01101011

9.3.1 The process of encryption

Without going through the proper rounds as proposed, just for the sake of example, the block-size has been chosen randomly. S is decomposed into a set of five blocks, namely S_1, S_2, S_3, S_4, and S_5, the first four being of size 32 bits and the last one of 16 bits. Hence,

S_1 = 01001100011011110110001101100001,
S_2 = 01101100001000000100000101110010,
S_3 = 01100101011000010010000001001110,
S_4 = 01100101011101000111011101101111, and
S_5 = 0111001001101011

For the block S_1, whose decimal value is $(1282368353)_{10}$, the process of encryption is as follows:

$1282368353^1 \Rightarrow 6411841771^1 \Rightarrow 3205920886^0 \Rightarrow 1602960443^1 \Rightarrow 801480222^0 \Rightarrow$
$400740111^1 \Rightarrow 200370056^0 \Rightarrow 100185028^0 \Rightarrow 50092514^0 \Rightarrow 25046257^1 \Rightarrow 12523129^1$
$\Rightarrow 6261565^1 \Rightarrow 3130783^1 \Rightarrow 1565392^0 \Rightarrow 782696^0 \Rightarrow 391348^0 \Rightarrow 195674^0 \Rightarrow 97837^1$
$\Rightarrow 48919^1 \Rightarrow 24460^0 \Rightarrow 12230^0 \Rightarrow 6115^1 \Rightarrow 3058^0 \Rightarrow 1529^1 \Rightarrow 765^1 \Rightarrow 383^1 \Rightarrow 192^0$
$\Rightarrow 96^0 \Rightarrow 48^0 \Rightarrow 24^0 \Rightarrow 12^0 \Rightarrow 6^0$, where the superscripts are the bits to be put into the target block,

Hence, T_1 = 11010100011110000110010111000000 is the target block generated corresponding to S_1.

Applying the same process, target blocks T_2, T_3, T_4 and T_5 corresponding to source blocks S_2, S_3, S_4 and S_5, respectively, are generated as

T_2 = 01110001011111011111101111001001,
T_3 = 01110001011111011111101111001001,
T_4 = 10001001000100011101000101011001, and
T_5 = 1110100110110001

Combining the target blocks in the same sequence, the target string of bits is generated as T = 11010100/01111000/01100101/11000000/01110001/01111101/
11111011/11001001/01110001/01111101/11111011/11001001/
10001001/00010001/11010001/01011001/11101001/10110001,
and the corresponding encrypted string of characters or the cipher-text (C), which is formed by substituting each 8-bit block by the corresponding ASCII character, will be "•9°═q}√╔M√yYė►╤YΘ□".

9.3.2 The process of decryption

During decryption, the cipher-text (C) is converted into the corresponding string of bits and broken into blocks accordingly. The blocks T_1, T_2, T_3, T_4 and T_5 are regenerated as

T_1 = 11010100011110000110010111000000,
T_2 = 01110001011111011111101111001001,
T_3 = 01110001011111011111101111001001,
T_4 = 10001001000100011101000101011001,
T_5 = 1110100110110001

Applying the process of decryption, the corresponding source blocks, namely S_1, S_2, S_3, S_4, and S_5, are generated. While scanning the block T_1 from the LSB towards MSB, the first bit obtained is 0, and hence even. The first even number is 2, which is assigned to S_1. The next bit is again 0, i.e. even. Since S_1=2 at this instance, the 2nd even number, i.e. 4, is assigned to S_1. The third bit being a 0, the 4th even number, i.e. 8, is assigned to S_1. Suppose at any instance, if S_1 = 57 and the next

scanned bit is 1, then the 57th odd number, i.e. 104, is assigned to S_1. Continuing in this fashion, the final value of S_1 is obtained as 1282368353 whose binary equivalent is 01001100011011110110001101100001. Similarly, S_2, S_3, S_4, and S_5 are computed and concatenated to form the final bit stream S. In general, the n^{th} even number is $2n$ and the n^{th} odd number is $2n$-1.

In actual implementation of the proposed scheme, the rounds for block-sizes 8, 16, 32, 64, 128, 256, and 512 are performed, iterating each round several times. The number of iterations in each round will form a part of the key that has been discussed in chapter 11.

9.4 Microprocessor-based implementation

As already mentioned before, the sweetness of the algorithm lies in its microprocessor-based implementation. Although it seems that lot of computation is required for determining the decimal equivalent of a binary string and subsequently ascertaining its position in the series of odd/even integers, the scheme has been implemented in an 8085 microprocessor-based system using a lucid approach. In actual implementation, the conversion from binary to decimal is not at all needed.

The shifted-out bit from the LSB determines whether the number is odd or even and this bit directly goes into the target block. The division operation, required for ascertaining the position of an integer in the series of even/odd numbers, has also been avoided. Since an arithmetic right-shift results in division by 2 and left-shift results in multiplication by 2, there is no need to calculate the position in the odd/even series.

During decryption the output string is initialised by 01H. The input string is shifted right. If the shifted-out bit is a '0', the output string is doubled, i.e. a '0' is shifted in from the LSB. If it is a '1', then the output bit is doubled and decremented by 1, i.e. a '0' is shifted in from the LSB and the resultant string is decremented once.

The routines for 16-bit DEPS encryption and decryption are too long. Hence, these are directly listed as programs at appropriate places in Appendix B. For block-

sizes higher than 16, small amendments, mostly comprising of modifications in register and memory initialisations, will be needed. The necessary amendments are given along with the respective programs. The routines for block-size 8 bits are presented in sections 9.4.1 and 9.4.2.

9.4.1 Routine for 8-bit DEPS encryption

This routine will perform 8-bit DEPS on a 512-bit block stored in the memory from a location, say F900H, onwards.

Step 1 : Load HL pair to point to memory location F900H
Step 2 : Load E with 40H
Step 3 : Load C with 00H and D with 08H
Step 4 : Clear Cy flag
Step 5 : Load A with the content of memory (location given by HL pair)
Step 6 : Rotate right with CARRY
Step 7 : If Cy = 0 then jump to step 9
Step 8 : Increment A
Step 9 : Store A into memory (location given by HL pair)
Step 10 : Move C to A
Step 11 : Rotate left with CARRY
Step 12 : Move A to C
Step 13 : Decrement D
Step 14 : Repeat from step 4 till D is zero
Step 15 : Load A with the content of memory (location given by HL pair)
Step 16 : Increment HL pair
Step 17 : Decrement E
Step 18 : Repeat from step 3 till E is zero
Step 19 : Return

9.4.2 Routine for 8-bit DEPS Decryption

Here also data-bytes are assumed to be stored in the memory from F900H onwards.

Step 1 : Load HL pair to point to memory location F900H
Step 2 : Load E with 40H
Step 3 : Load B with 01H and D with 08H
Step 4 : Load A with the content of memory (location given by HL pair)
Step 5 : Clear Cy flag
Step 6 : Rotate right with CARRY
Step 7 : Move A to C

Step 8 : Move B to A
Step 9 : If Cy = 0 then jump to step 14
Step 10 : Complement Cy
Step 11 : Rotate left with CARRY
Step 12 : Decrement A
Step 13 : Jump to step 15
Step 14 : Rotate left with CARRY
Step 15 : Move A to B
Step 16 : Move C to A
Step 17 : Decrement D
Step 18 : Repeat from step 5 till D is zero
Step 19 : Store B into memory (location given by HL pair)
Step 20 : Increment HL pair
Step 21 : Decrement E
Step 22 : Repeat from step 3 till E is zero
Step 23 : Return

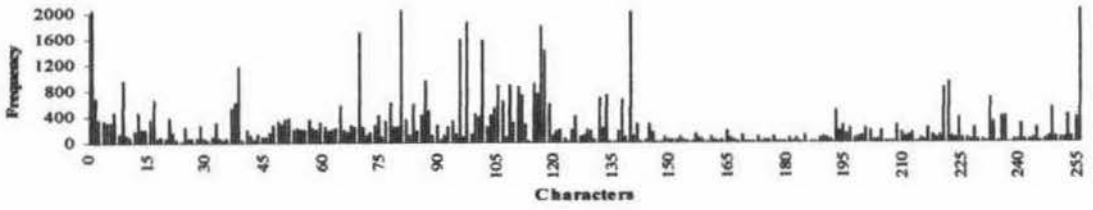
9.5 Results and comparisons

Like the previous algorithms, the strength and weaknesses of DEPS have also been tested in its weakest form, i.e., having just one pass (no iterations) in each round, so that the strength of the algorithm may increase during actual implementation.

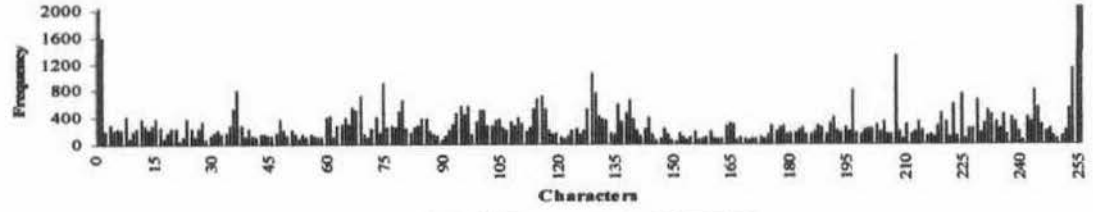
The same set of twenty files, and more precisely, five different files of varying sizes in each of the four categories, namely .dll, .exe, .jpg and .txt, were considered for the purpose of testing and were encrypted using the DEPS algorithm. The results of the tests have been compared with those of Triple DES.

9.5.1 Character frequency

The frequencies of all the 256 ASCII characters in the source file and the encrypted files are computed to compare how evenly the characters are distributed over the 0-255 region. Among the twenty files encrypted, the results of just one file in each of the four categories are shown here to make things simple. The variation of frequencies of all the 256 ASCII characters in the .dll source file and the ones obtained as results of encryption with DEPS and Triple DES are shown in figure 9.2. Likewise, figures 9.3 to 9.5 illustrate the comparative character-frequencies for files of the other three categories.



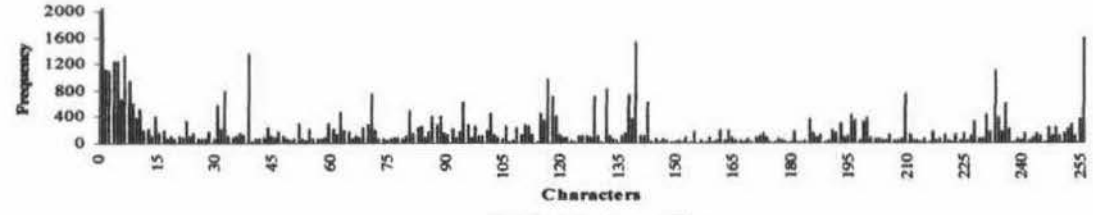
(a) Original .dll file



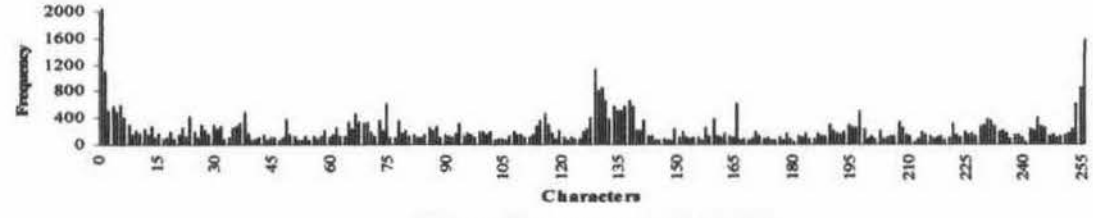
(b) .dll file ecrypted with DEPS

(c) .dll file encrypted with Triple DES

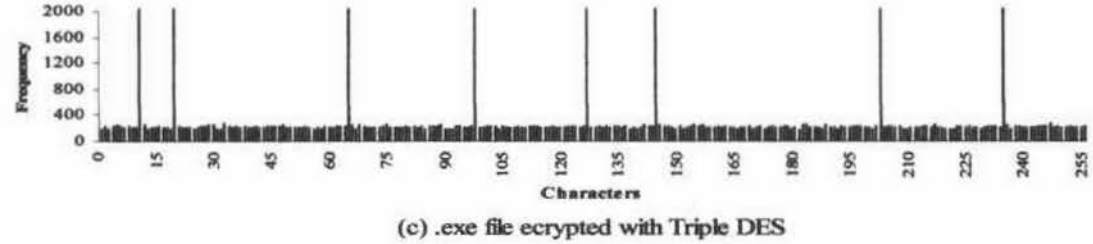
Figure 9.2: Character-frequencies in the source and encrypted .dll files



(a) Original .exe file

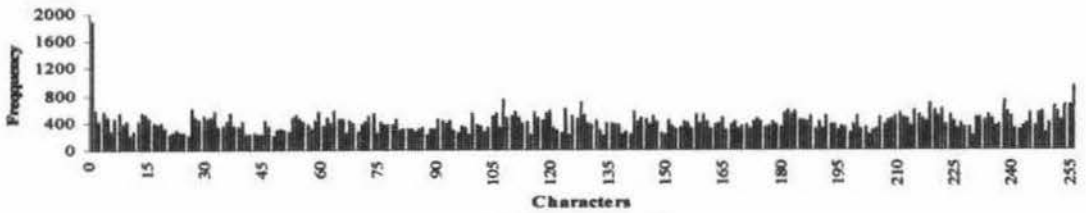


(b) .exe file encrypted with DEPS



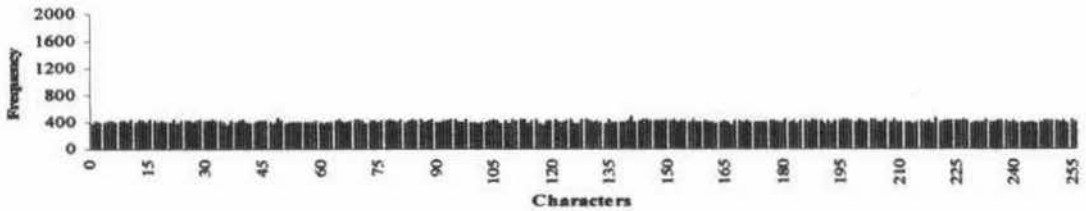
(c) .exe file ecrypted with Triple DES

Figure 9.3: Character-frequencies in the source and encrypted .exe files



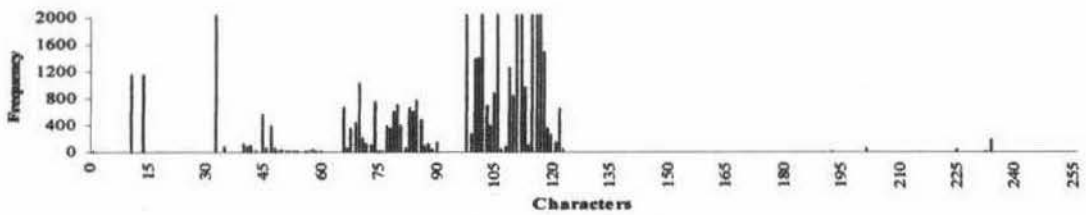
(a) Original .jpg file

(b) .jpg file encrypted with DEPS

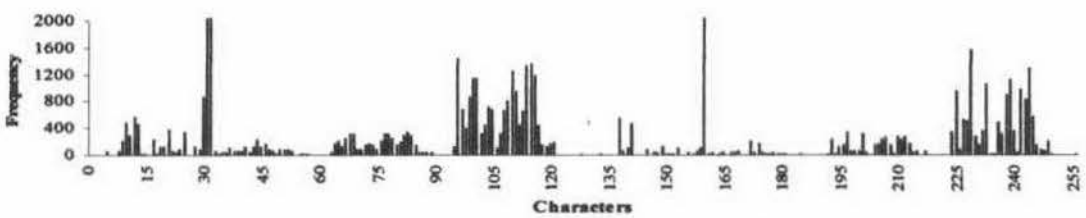


(c) .jpg file encrypted with Triple DES

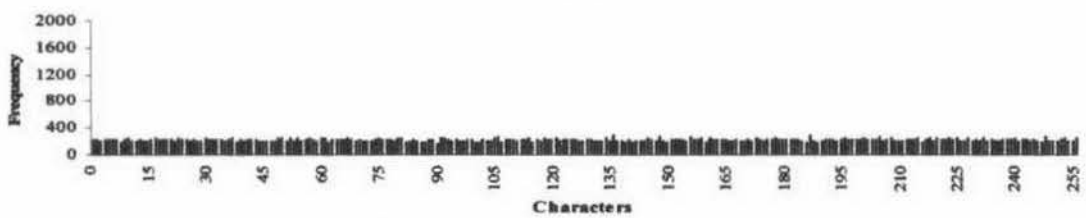
Figure 9.4: Character-frequencies in the source and encrypted .jpg files



(a) Original .txt file



(b) .txt file encrypted with DEPS



(c) .txt file encrypted with Triple DES

Figure 9.5: Character-frequencies in the source and encrypted .txt files

As usual, very high frequencies of few characters in some graphs have been truncated to make the low values bit visible. If this is not done, they will look like almost zero values.

In case of the .dll file, the result shown by DEPS is not as good as that of Triple DES. Nevertheless, the characters in the DES encrypted .dll file are not so much clustered in some particular regions. Some very high frequencies in the original file have been reduced and some very low frequencies have been boosted by considerable amounts.

The results for .exe file are quite similar to .dll file and needs no further explanations. The performances shown by both DEPS and Triple DES in case of .exe file are almost same as in .dll file.

The performance of DEPS in case of .jpg file is much better than in .dll and .exe files. All the characters are fairly distributed over the 0-255 region and this is quite comparable to Triple DES.

DEPS has not shown a good performance for .txt file compared to other proposed algorithms of this thesis. However, as in the case of transposition ciphers, this deficiency can be overcome by cascading with another cipher. Since DEPS is a substitution cipher, it can be cascaded with a transposition cipher like BET, SPOB etc.

9.5.2 Chi-Square test and encryption time

The quality of a cipher may be judged by analysing how different the original and encrypted files are. To check the heterogeneity between the original and encrypted pairs of all the twenty files, the most popular statistical tool, i.e. the χ^2-test, was performed as in the previous cases. Another aspect of quality is the encryption time. The χ^2 values and encryption times due to DEPS were compared with those of Triple DES. Each category of files has been dealt with separately. The comparative χ^2 values and encryption times for DEPS along with those for Triple DES in case of .dll files are listed in table 9.1. The same is visualised in figure 9.6.

Table 9.1: χ^2-test for DEPS with .dll files

Sl. No.	Original file	File size (bytes)	DEPS			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.dll	20480	0.989011	7012	183	06	29790	255
2	2.dll	53312	3.021978	22285	255	16	43835	255
3	3.dll	90176	5.164835	40456	255	26	66128	255
4	4.dll	118784	6.593406	126714	255	34	1211289	255
5	5.dll	204800	11.648351	1578332	255	69	2416524	255

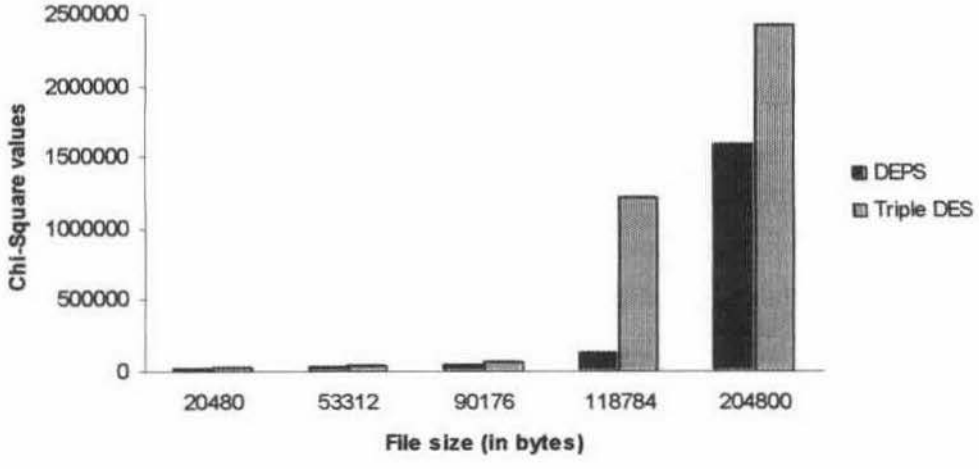


Figure 9.6: DEPS vs. Triple DES in χ^2-test of .dll files

Just like other proposed algorithms, the performance of DEPS in χ^2-test with .dll files is bit weak compared to Triple DES. Even then, very small encryption time and large χ^2 values with 255 degrees of freedom (DF) for almost all the .dll files indicate the strength of DEPS. Since Triple DES is quite complicated, it takes a long time to encrypt a file compared to DEPS. Table 9.2 and figure 9.7 give the results of the test for .exe files.

Table 9.2: χ^2-test for DEPS with .exe files

Sl. No.	Original file	File size (bytes)	DEPS			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.exe	23104	1.428571	7564	255	12	8772	255
2	2.exe	52736	2.967033	22863	255	15	43426	255
3	3.exe	131136	6.593406	900984	255	29	986693	255
4	4.exe	170496	10.164835	151307	255	49	475893	255
5	5.exe	200832	16.593407	2119639	255	58	1847377	255

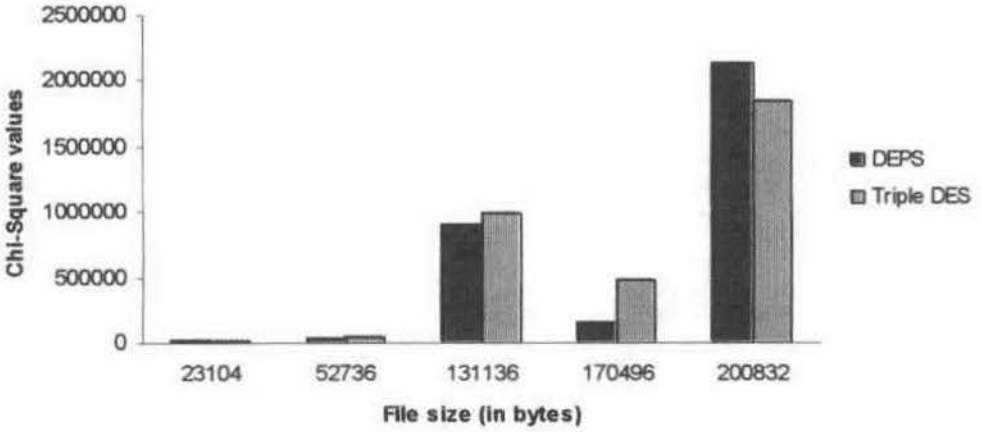


Figure 9.7: DEPS vs. Triple DES in χ^2-test of .exe files

The test results of DEPS for .exe files are better than those for .dll files, in fact, even better than Triple DES is case of some files,. Further, the encryption times are much less than that of Triple DES. The test results for .jpg files are listed in table 9.3 and illustrated by figure 9.8.

Table 9.3: χ^2-test for DEPS with .jpg files

Sl. No.	Original file	File size (bytes)	DEPS			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.jpg	28544	1.703297	4053	255	08	4331	255
2	2.jpg	71232	4.285714	2963	255	21	2916	255
3	3.jpg	105600	6.318681	4439	255	31	5227	255
4	4.jpg	160704	9.725274	23053	255	47	22314	255
5	5.jpg	216576	13.296702	31100	255	63	29824	255

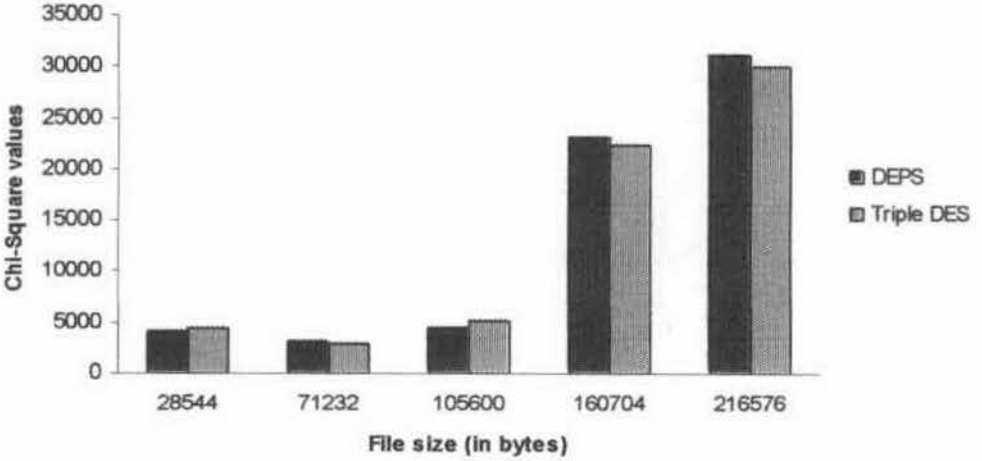


Figure 9.8: DEPS vs. Triple DES in χ^2-test of .jpg files

The performance of DEPS for .jpg files is much better than the same for .dll and .exe files. In case of three out of five files, the χ^2 values for DEPS are higher than those for Triple DES. High χ^2 values, all with 255 degrees of freedom (DF), and very small encryption time compared to Triple DES proves the strength of DEPS for .jpg files. The results for .txt files are given by table 9.4 and figure 9.9.

Table 9.4: χ^2-test for DEPS with .txt files

Sl. No.	Original file	File size (bytes)	DEPS			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	t1.txt	6976	0.439560	7895	86	02	10629	183
2	t2.txt	23808	1.428571	28461	168	07	32638	255
3	t3.txt	58688	3.516483	70257	182	17	82101	255
4	t4.txt	118784	7.142857	154803	215	35	170557	255
5	t5.txt	190784	11.538461	400590	205	55	430338	255

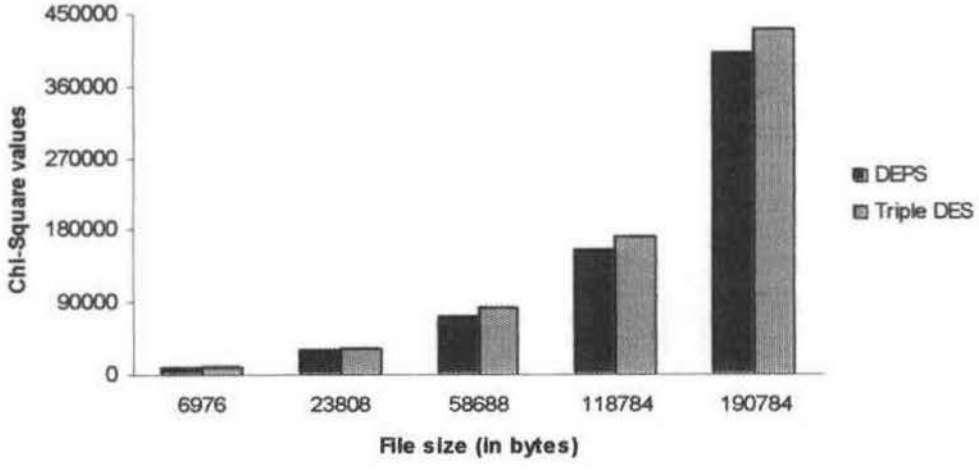


Figure 9.9: DEPS vs. Triple DES in χ^2-test of .txt files

The nature shown by DEPS for .txt files is very much similar to that shown by Triple DES. The only exception is that the degrees of freedom (DF) for DEPS in case of all the five files are bit lesser than those for Triple DES, but it does not have so much of significance.

9.5.3 Avalanche and runs

To examine the diffusion property of DEPS, a 32-bit binary string was repeatedly encrypted, first keeping the original string unaltered, and subsequently

each time complementing one bit of the plain-text. The differences between the cipher-texts were noted and the number of runs was also counted in each plain-text and the corresponding cipher-text. The difference of runs in each plain-text/cipher-text pair was also noted. Table 9.5 shows the results of this test for DEPS.

Table 9.5: Avalanche and runs in DEPS

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	FDDD340D	19	15	4
1ST	C145D450	FCDD340D	18	15	3
2ND	0145D450	FFDD340D	17	14	3
3RD	6145D450	F9DD340D	19	15	4
4TH	5145D450	F5DD340D	21	17	4
5TH	4945D450	EDDDC40D	21	17	4
6TH	4545D450	DDDD340D	21	17	4
7TH	4345D450	DDDD340D	19	17	2
8TH	4045D450	03DD340D	17	14	3
9TH	41C5D450	FDDC340D	17	13	4
10TH	4105D450	FDDF340D	17	13	4
11TH	4165D450	FDD9340D	19	15	4
12TH	4155D450	FDD5340D	21	17	4
13TH	414DD450	FDCD340D	19	15	4
14TH	4141D450	FDFD340D	17	13	4
15TH	4147D450	FD9D340D	17	15	2
16TH	4144D450	FD3D340D	19	15	4
17TH	41455450	FDDD350D	21	17	4
18TH	41459450	FDDD360D	19	15	4
19TH	4145F450	FDDD300D	17	14	3
20TH	4145C450	FDDD3D0D	17	13	4
21ST	4145DE50	FDDD240D	17	15	2
22ND	4145D050	FDDD0C0D	17	13	4
23RD	4145D650	FDDD540D	19	17	2
24TH	4145D550	FDDDD40D	21	15	6
25TH	4145D4D0	FDDD340F	19	15	4
26TH	4145D410	FDDD3409	17	13	4
27TH	4145D470	FDDD3403	17	15	2
28TH	4145D440	FDDD3415	17	13	4
29TH	4145D458	FDDD3435	19	17	2
30TH	4145D454	FDDD3435	21	17	4
31ST	4145D452	FDDD3475	21	17	4
32ND	4145D451	FDDD34F5	20	17	3

The table shows that DEPS can produce a good amount of effect in the cipher-text with a very small change in the plain-text. Almost all the cipher-texts have differences in at least two bytes compared to the first one. The difference of 4 runs between the plain-text and the cipher-text in most of the cases also proves the randomness property of DEPS. Hence, DEPS causes some amount of diffusion, although not as much as OMAT or MMAT. When cascaded with a transposition

cipher like DSPB, DEPS might show better results. This aspect has been treated in chapter 10.

9.6 Conclusion

DEPS is a simple, easy-to-implement, and an efficient system that takes little time to encode and decode though the block length is high. The encoded string will not generate any overhead bits. The technique appears to produce a computationally non-breakable cipher-text. The result of the frequency distribution test shows the fact that the cipher characters are distributed wide enough. The fact that the source and the encrypted files are non-homogeneous is established by the Chi-Square test. Since it does not involve a lot of arithmetic computations in its actual implementation, the proposed scheme is highly recommended for embedded systems with less powerful processors and very small memory units.