A Permutative Cipher Technique (PCT) to Enhance the Security of Network Based Transmission

Jyotsna Kumar Mandal
Raiganj University
401 PUBLICATIONS **2,699** CITATIONS

A Permutative Cipher Technique (PCT) to Enhance the Security of Network Based Transmission

Manas Paul,[1] Jyotsna Kumar Mandal,[2]
[1]JIS College of Engineering, Kalyani, West Bengal, India, e-mail: manaspaul@rediffmail.com
[2]C.S.E. Dept., Kalyani University, Kalyani, West Bengal, India,E-mail:jkmandal@rediffmail.com

ABSTRACT
The permutative cipher technique (PCT) considers a message as binary string on which the principle of permutation of bits within a block follows. The input binary string breaks into manageable-sized chunks (8 - 8192 bits), build a secret key algorithm to perform bit shuffle recursively for a finite number of times on each chunk, generate encrypted binary stream.
The bit shuffle is symmetric in nature, i.e. bit shuffle is performed on each input chunk for n number of times the original stream is regenerated. Combination of values of chunks of a session generates the session key of the technique. To get the encrypted chunk, perform the bit shuffle operation to each input chunk for (n-m) times. Decoding is done following the same procedure after performing the bit shuffle operation to each encrypted chunk for m times.
A comparison of the proposed technique with existing and industrially accepted RSA and Triple DES has also been done in terms of frequency distribution and non homogeneity of source and encrypted files.

KEYWORDS
Permutative Cipher Technique (PCT), Session Key, RSA, Triple DES (TDES), Chi-Square.

1. INTRODUCTION

Network security is the most focused subject which is becoming more and more complex to maintain and Cryptography is the heart of security [10, 11, 12]. The people all over the world are engaged in communication through internet almost everyday. The important documents need to be secure from eavesdroppers. The encryption process [1, 2, 3, 4] makes the document into cipher text which will not be legible to them. Many algorithms are available, each of which has merits and demerits [5, 6, 7, 8, 9]. No single algorithm is sufficient for this application. As a result researchers are working in the field of cryptography to enhance the security further.

In this paper a new technique is proposed where the plain text is considered as a stream of bits and can be conceived as block of different length. The technique transforms each block into unintelligent form, the cipher text. The plain text can also be recovered using the same technique. The technique has been implemented using 'C' language [13, 14]. Section 2 deals with the technique. A proposal for generation of session key is described in Section 3. Results and analysis are given in section 4 and 5 respectively. Conclusions are drawn in section 6 and references are drawn in section 7.

2. PROPOSED TECHNIQUE

The technique considers the plain text as a stream of finite numbers of bits in the form of blocks with different lengths like 8 / 16 / 32 / 64 / …. / 2048 / 4096 / 8192 bits as follows

First n_1 number of bits is considered as x_1 no of blocks with length y_1 bits where $n_1 = x_1*y_1$, Next n_2 number of bits is considered as x_2 no of blocks with length y_2 bits where $n_2 = x_2*y_2$ and so on. Finally last n_m number of bits is considered as x_m no of blocks with length y_m bits where $n_m <= x_m*y_m$

If $n_m < x_m*y_m$, then padding up the plain text at the end with ($x_m*y_m - n_m$) bits. Table 2.1 shows the process of creation of blocks from a continuous stream.

Table: 2.1
Formation of blocks from stream of b its

No. of bits of plain text	No. of blocks	Length of each block (bits) (order of 2^k)	No. of blocks * size of blocks(in bits)
First n_1 bits	x_1	y_1	$n_1 = x_1*y_1$
Next n_2 bits	x_2	y_2	$n_2 = x_2*y_2$
Next n_3 bits	x_3	y_3	$n_3 = x_3*y_3$
..	..	..	..
Next n_{m-1} bits	x_{m-1}	y_{m-1}	$n_{m-1} = x_{m-1}*y_{m-1}$
Next n_m bits	x_m	y_m	$n_m <= x_m*y_m$

The technique generates a sequence number (1,3,5,7,…….) for each bit position for a fixed block starting from MSB towards LSB with length n using the following function:

$$f1(p) = (2n + 1 - 2p) * \mathrm{mod}(p+1,2) + p;$$

where, n = block length under consideration
p = position of the pth bit
mod(p+1,2) is a function which returns the remainder value when (p+1) is divided by 2
f1(p) is a function which returns the generating position of the pth bit

As for example if we generate the position of the 4th bit (i.e. p = 4) of a block of length 16 (i.e. n = 16), it may be done using the proposed function as

$$f1(4) = (2*16 + 1 – 2*4) * \mathrm{mod}(4+1,2) + 4$$

$= 25 * 1 + 4 \quad = 29$

Hence the sequence number of the 4th bit is generated as 29. Similarly, to generate the position of the 7th bit of a block of length 16, the generated value will be 7. So generating position of odd bits remain same and that for even bits changed to odd number whose value is greater than n but less than 2*n.
The positional orientation of bits is done to generate the next intermediate block of the encryption process by using the following function

$$f2(q) = (q+1) / 2 ;$$

where, q = generated bit position
f2(q) is a function which returns the bit position of the next stage of the encryption process

As for example, if the generated bit position of the kth bit is 29 (i.e. q=29), then the position of the kth bit of the previous block will be f2(29) into the next intermediate block where the value of f2(29) will be 15.
If n is the block length, f1(p) generates the consecutive odd integers from 1 to (2*n – 1). Using these generated numbers f2(q) generates the natural numbers from 1 to n. Table 2.2 shows the generating position of each bit of a block and new position for next intermediate block with block length 16

Table 2.2
Bit positions of each bit within a block

Original bit position for any block	Generated bit position using function f1(p)	Bit position for next intermediate block using function f2(p)
01	01	01
02	31	16
03	03	02
04	29	15
05	05	03
06	27	14
07	07	04
08	25	13
09	09	05
10	23	12
11	11	06
12	21	11
13	13	07
14	19	10
15	15	08
16	17	09

If the process is continued, the original bit stream of block length n (= 2^k) will be regenerated after (k+1) number of iterations. Any of the intermediate blocks generated in the process may be used as encrypted string. As the process is symmetric in nature, the same procedure may be followed for decryption to regenerate the plain text.

If j number of iteration (j < k+1) is done in encryption process, then during the decryption (k+1-j) no. of iteration is required to regenerate the plain text. The value of j and k are the secret key of the proposed technique. The same process may be repeated in cascaded manner for a varying block length which may enhance the security further. A session key may be generated (as described in section 3) for one time use in a session of transmission to ensure more security.
Section 2.1 describes the algorithm of the proposed technique and section 2.2 illustrates the complete technique with an example.

2.1 ALGORITHM

Step 1: Take a block with length n (= 2^k) bits from source string
Step 2: Generate the position of each bit using the following function

$$f1(p) = (2n + 1 - 2p) * mod(p+1,2) + p;$$

where, n = block length under consideration
p = position of the pth bit
mod(p+1,2) is a function which returns the remainder value when (p+1) is divided by 2
f1(p) is a function which returns the generating position of the pth bit
Step 3: To generate the next intermediate block, the positional orientation of bits are performed using the following function

$$f2(q) = (q+1) / 2 ;$$

where, q = generated bit position
f2(q) is a function which returns the bit position of the next stage of the encryption process
Step 4: If the steps 2 and 3 continued for (k+1) no. of iteration, source block will be regenerated.
Step 5: Any intermediate block of step 4 may be used as encrypted string.

2.2 EXAMPLE

To illustrate the proposed technique, consider a 16 bit source block as "1101001111001010". For each block, the position of each bit is regenerated first and then the positional orientations of bits are performed to generate next intermediate block using the proposed technique. Regenerate position of each bit and new bit position for next intermediate block is already shown in the figure 2.2. The source block, intermediate blocks and the final block which is nothing but source block is given in the figure 2.2.1.

1	1	0	1	0	0	1	1	1	1	0	0	1	0	1	0

The source stream "1101001111001010"

1	0	0	1	1	0	1	1	0	0	0	1	1	0	1	1

First intermediate block

1	0	1	1	0	0	1	1	1	0	1	0	1	0	1	0

Second intermediate block

1	1	0	1	1	1	1	1	0	0	0	0	1	0	1	0

Third intermediate block

1	0	1	1	0	0	1	1	0	0	0	0	1	1	1	1

Fourth intermediate block

1	1	0	1	0	0	1	1	1	1	0	0	1	0	1	0

Regenerated source block

Figure 2.2.1
Formation of the cycle of the process

Since the length of the source block is 16 (= 2^4), so after (4+1) i.e. 5th iteration the source block is regenerated. Any (1st or 2nd or 3rd or 4th) intermediate block may be used as encrypted block. Here if j no. of iteration is used to generate encrypted block, then (5 – j) no. of iteration is used to generate decrypted block using the same technique for single stage.

3. GENERATION OF SESSION KEY

A session key may be generated for one time use in a session of transmission to ensure more security which divides the input data each time of cascading dynamically into 16 portions of bit streams. Each portion of the bit stream is divided again into x_m no of blocks with block length y_m bits. Total length of the source stream can be expressed as

$$x_1*y_1 + x_2*y_2 + x_3*y_3 + \ldots.. + x_m*y_m$$
(if padding is not required)

and

$$x_1*y_1 + x_2*y_2 + x_3*y_3 + \ldots.. + x_m*y_m – \text{padded portion}$$
(if padding is required)

The session key contains the secret key value which consists of the value of x_m and y_m . Possible values of x_m and y_m (in bits) are shown with the help of two arrays b[] and s[] respectively as b[0] = 0 ; b[1] = a ; b[2] = a^2 ; ……. ; b[n] = a^n (where a (>1)is appositive integer)and s[0] = 0 ; s[1] = 8 = 2^3 ; s[2] = 16 = 2^4 ; ……. ; s[n] = $2^{(n+2)}$. Secret key contains total 16*c characters for c times of cascading. For each character (consists of 8 bits), first 4 bits contain the information of number of blocks and the last 4 bits contain the information of block length. If the first 4 bit is '0110' and the last 4 bit is '0101', whose corresponding decimal values are '6'and '5', then the character (whose 8 bit representation is '01100101' contains the information that there are b[6] i.e. a^6 no of blocks with block length s[5] i.e. 128. So the secret key is different for different session of the same bit stream. Hence this approach may enhance the security of the proposed technique.

4. RESULTS

In this section the results of implementations are given. The implementation is made through 'C' programming language. The fifteen files of different sizes, starting from 1 KB to 4.17 MB are taken for experiments.

4.1 STUDY OF ALGORITHMS ON .TXT FILE TYPES WITH VARYING SIZES

Table 4.1 represents size of .txt files before and after encryption, encryption time and decryption time using proposed algorithm

Table: 4.1
File size v/s Encryption & Decryption Time for .txt files for Proposed Algorithm

Source File Name	Source File Size before encryption (in Bytes)	Encrypted File Size (in Bytes)	Encryption Time (in sec)	Decryption Time (in sec)
s01.txt	1,696	1,707	0.00	0.00
s02.txt	8,181	8,184	0.05	0.00
s03.txt	19,684	19,689	0.05	0.05
eula.txt	41,842	41,844	0.11	0.11
s05.txt	70,815	70,842	0.22	0.22
s06.txt	149,848	149,859	0.38	0.44
s07.txt	343,587	343,713	0.99	1.10
s08.txt	639,159	639,204	1.76	1.98
s09.txt	982,732	983,484	2.69	2.75
s10.txt	1,395,453	1,396,026	3.74	4.34
s11.txt	1,739,050	1,739,349	4.83	4.78
s12.txt	2,109,551	2,111,346	5.71	5.77
s13.txt	2,790,946	2,791,368	7.91	7.86
s14.txt	3,284,369	3,285,090	9.23	9.17
s15.txt	3,985,411	3,985,548	10.99	11.04

Figure 4.1 shows the relationship between the encryption times against the source files and also between the decryption times against the source files using proposed algorithm.

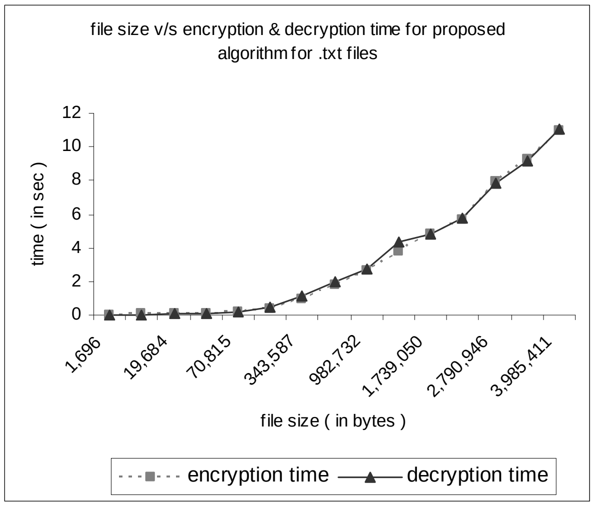

Figure: 4.1
File size v/s Encryption & Decryption Time for .txt files for Proposed Algorithm

Table 4.2 represents size of .txt files before and after encryption, encryption time and decryption time using RSA algorithm

Table: 4.2
File size v/s Encryption & Decryption Time for .txt files for RSA Algorithm

Source File Name	Source File Size before encryption (in Bytes)	Encrypted File Size (in Bytes)	Encryption Time (in sec)	Decryption Time (in sec)
s01.txt	1,696	8,734	00	00
s02.txt	8,181	41,528	00	00
s03.txt	19,684	99,888	00	00
eula.txt	41,842	231,243	00	01
s05.txt	70,815	385,560	00	00
s06.txt	149,848	830,986	01	01
s07.txt	343,587	1,913,110	02	02
s08.txt	639,159	3,556,458	03	04
s09.txt	982,732	5,359,344	04	07
s10.txt	1,395,453	7,891,421	07	09
s11.txt	1,739,050	9,663,598	08	11
s12.txt	2,109,551	11,868,835	10	13
s13.txt	2,790,946	15,787,222	14	18
s14.txt	3,284,369	18,328,943	17	21
s15.txt	3,985,411	22,552,576	20	25

Figure 4.2 shows the relationship between the encryption times against the source files and also between the decryption times against the source files using RSA algorithm.

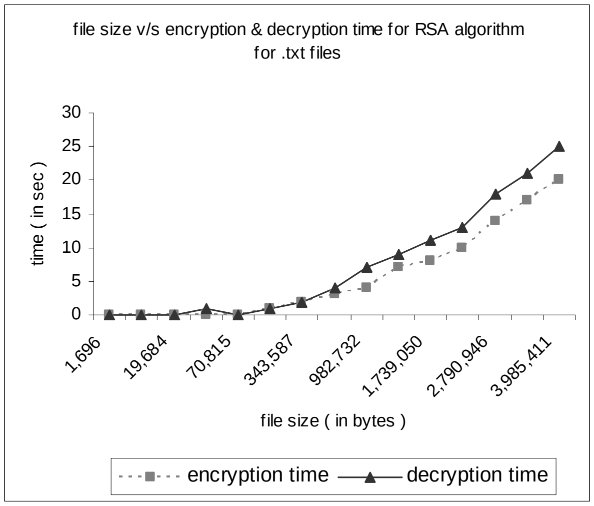

Figure: 4.2
File size v/s Encryption & Decryption Time for .txt files for RSA Algorithm

Table 4.3 represents size of .txt files before and after encryption, encryption time and decryption time using TDES algorithm

Table: 4.3
File size v/s Encryption & Decryption Time for .txt files for TDES Algorithm

Source File Name	Source File Size before encryption (in Bytes)	Encrypted File Size (in Bytes)	Encryption Time (in sec)	Decryption Time (in sec)
s01.txt	1,696	1,704	01	00
s02.txt	8,181	8,184	02	01
s03.txt	19,684	19,688	03	03
eula.txt	41,842	41,848	07	08
s05.txt	70,815	70,816	12	12
s06.txt	149,848	149,856	26	26
s07.txt	343,587	343,592	58	59
s08.txt	639,159	639,160	109	110
s09.txt	982,732	982,736	168	168
s10.txt	1,395,453	1,395,456	239	239
s11.txt	1,739,050	1,739,056	305	311
s12.txt	2,109,551	2,109,552	376	380
s13.txt	2,790,946	2,790,952	506	525
s14.txt	3,284,369	3,284,376	572	595
s15.txt	3,985,411	3,985,416	739	725

Figure 4.3 shows the relationship between the encryption times against the source files and also between the decryption times against the source files using TDES algorithm.

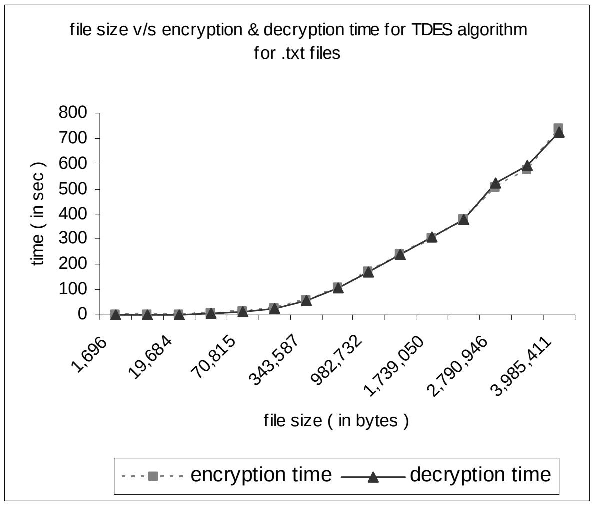

Figure: 4.3
File size v/s Encryption & Decryption Time for .txt files for TDES Algorithm

4.2 TESTS FOR NON-HOMOGENEITY ON .TXT FILE TYPES WITH DIFFERENT SIZES

Performing the test for goodness of fit (Pearson χ^2) between the source files and encrypted files, the Chi-Square values and the corresponding degree of freedom are calculated for three types of files of different sizes. The high range of the Chi-Square value gives the high degree of non-homogeneity between the source files and encrypted files.

Table 4.4 represents the Chi-Square values and the corresponding degree of freedom values for .txt source files and corresponding encrypted files using proposed algorithm.

Table: 4.4
Chi-Square values & degree of freedom values for .txt source and encrypted files for Proposed algorithm

Source File Name	Source File Size (in Bytes)	Calculated values(c) (degree of freedom)	Tabulated values (t)	c/t
s01.txt	1,696	3253 (79)	111.14	29
s02.txt	8,181	15802 (229)	281.71	56
s03.txt	19,684	38735 (255)	310.46	125
eula.txt	41,842	77943 (256)	311.56	250

Table 4.5 represents the Chi-Square values and the corresponding degree of freedom values for .txt source files and corresponding encrypted files using RSA algorithm.

Table: 4.5
Chi-Square values & degree of freedom values for .txt source and encrypted files for RSA algorithm

Source File Name	Source File Size (in Bytes)	Calculated values(c) (degree of freedom)	Tabulated values (t)	c/t
s01.txt	1,696	3342 (52)	78.62	42
s02.txt	8,181	16225 (55)	82.29	197
s03.txt	19,684	39055 (51)	77.39	505
eula.txt	41,842	80214 (114)	152.04	528

Table 4.6 represents the Chi-Square values and the corresponding degree of freedom values for .txt source files and corresponding encrypted files using TDES algorithm

Table: 4.6
Chi-Square values & degree of freedom values for .txt source and encrypted files for TDES algorithm

Source File Name	Source File Size (in Bytes)	Calculated values(c) (degree of freedom)	Tabulated values (t)	c/t
s01.txt	1,696	3229 (94)	128.80	25
s02.txt	8,181	15984 (222)	273.94	58
s03.txt	19,684	38444 (256)	311.56	123
eula.txt	41,842	76101 (256)	311.56	244

Figure 4.4 shows the relationship between the source file size and c/t (i.e. calculated Chi-Square value / tabulated Chi-Square value) for .txt files using all three algorithms (Proposed, RSA, TDES) for single stage.

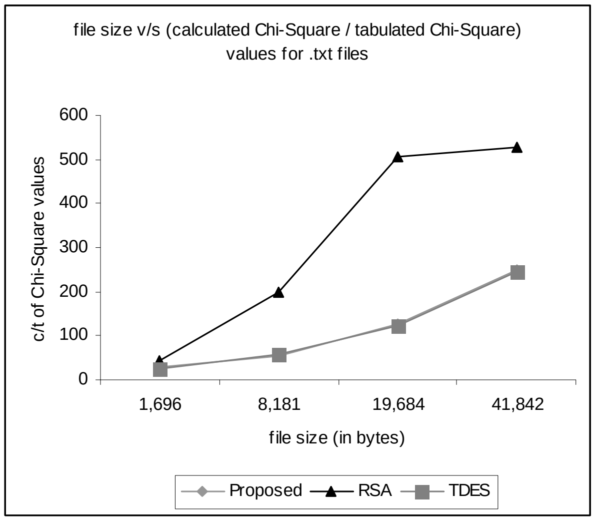

Figure: 4.4
File size v/s (calculated Chi-Square / tabulated Chi-Square) values for .txt files

5. ANALYSIS

Analyzing all the results given in section 4, following are the points obtained after comparing the three algorithms (Proposed, RSA, TDES)

i) For RSA algorithm, file size is increased by 5 to 6 times after encryption. But in case of proposed or TDES algorithm file size is remains same after encryption.

ii) For TDES algorithm, the encryption and decryption time both are 60 to 70 times more than that for proposed algorithm and for RSA algorithm, the encryption and decryption time both are 2 times than that for proposed algorithm i.e. proposed algorithm takes least time for encryption and decryption among all three algorithms.

iii) There is not much difference observed between the encryption time and decryption time, which indicates that the complexity of computation for all the processes is approximately similar.

iv) The graph shows that the encryption/decryption time increases with the increase of source file sizes. But the slope is higher for larger file size.

v) The calculated Chi-Square values are much higher than the tabulated Chi-Square values, which prove the high degree of nonhomogeneity between source and encrypted files for all three algorithms.

vi) For .txt files, the ratio of calculated Chi-Square value and tabulated Chi-Square value remains same for TDES and proposed algorithm but that for RSA algorithm is bit higher than other two algorithms. For .exe files, the ratio value remains same for all three algorithms.

vii) The degree of non homogeneity varies almost linearly with the size of the source file.

viii) In the proposed session key, the numbers of blocks taken as power of 3 including 0, which may limit the input file size to a certain extent (approximately 690 MB). However this value i.e. 3 may be changed to any higher integer value to accommodate the source file of any size.

CONCLUSION

The proposed technique is implemented for different types of files as .txt, .doc, .exe, .dll, .sys. Due to padding (padded with space character), the length of the encrypted file size may increase, but this is negligible compare to the total size of the source file. Since the block length and the no. of blocks are generated randomly, so the secret key is different for different session of the same bit stream.

The proposed technique has been implemented at bit level.

For the proposed session key, the technique may encrypt/decrypt file of size up to 690 MB, but if the no. of blocks may be power of higher integer value then it may accommodate input file of any size.

The same process may be repeated in cascaded manner for a varying block length which may enhance the security further which is the future scope of the work.

FUTURE SCOPE

The size of the chunk may be enhanced further beyond 8192 bits to make the algorithm more secured. Various other parametric tests may be performed to ensure the operability of the proposed scheme. Cascading of algorithm with some other technique may also be done. Cipher Block Chaining may also be performed to impart more security.

REFERENCES

[1] P.K. Jha, S. Shakya, & J.K. Mandal, Encryption through Cascaded Recursive Arithmetic Operation and Key Rotation of a session key,*ZERONE, Annual Technical Journal, Dept. of Computer and Electronic Eengineering, Volume 4, 2062/2005*, pp 51-58, The Institute of Engineering, Pulchowk, Kathmandu, Nepal, 2006.

[2] J.K. Mandal, et al, A 256-bit Recursive Pair Parity Encoder for Encryption, *Advances in Modeling, D; Computer Science & Statistics (AMSE), Vol. 9, No. 1*, pp. 1-14, France, 2004.

[3] J.K. Mandal, S. Mal, & S. Dutta, A Microprocessor Based Generalized Recursive Pair Parity Encoder for Secured Transmission, *Journal of Technology, Vol. XXXVII, No. 1-2*, pp. 11-20, India, July 2003.

[4] J.K. Mandal,, et al, Microprocessor-Based Bit Level Cryptosystem Through Arithmetic Manipulation of Blocks (AMB), *Journal of Institute of Engineering, Vol. 4, No. 1*,pp. 1-7, Nepal, December 2004.

[5] J.K. Mandal, S. Dutta, A Universal Bit-level Encryption Technique, *Seventh Vigyan Congress*, Jadavpur University, India, 28Feb to Ist March, 2000

[6] J.K. Mandal, P.K. Jha, Encryption Through Cascaded Recursive Key Rotation of a Session Key with Transposition and Addition of Blocks (CRKRTAB), Proceed. *National Conference of Recent Trends in Information Systems (ReTIS-06)*, IEEE Calcutta Chapter & Jadavpur University, 14-15 July, 2006.