

An Approach towards Design and Implementation of Microprocessor-Based Cryptosystems for Secured Transmission

**By
SHARAD SINHA**

**Dept. of Computer Science & Application
University of North Bengal**

515018



**Thesis submitted to the University of North Bengal
for the fulfillment of the Degree of Doctor of Philosophy
(Computer Science)**

2007

An Approach towards
Design and Implementation of
Microprocessor-Based Cryptosystems
for Secured Transmission

Ref. -
004.6
5617a

STOCKTAKING - 2011

BY
SHARAD SINHA

Dept. of Computer Science & Application
University of North Bengal

212074
02 FEB 2009



This thesis submitted to the University of North Bengal
for the fulfillment of the Degree of Doctor of Philosophy
(Computer Science)

2007



ACKNOWLEDGMENTS

It really gives me an immense pleasure in expressing my sincerest gratitude to Prof. Jyotsna Kumar Mandal, my guide and mentor, who was solely responsible for enabling to undertake this daunting task. He was always by my side, coaxing and cajoling me, all through my research work. I cannot thank enough his wife, Mrs. Shyamali Mandal, and daughter, little Khushi, for their love and support during my frequent visits to their place that had become, and will always remain, my second home.

Secondly, my wonderful parents deserve my heartfelt gratitude, not that they would expect it from me, for all that they have done for me. I vividly remember what my father once told me, "*Son, I wouldn't mind going without a shirt on my back in order to see you reach the pinnacle of your academic career*". That became the '*mantra*', which inspired and goaded me into striving hard towards my goal, milestone after milestone.

Neelima, my better half, gets the sole credit for expediting my work by acting as a catalyst. In the same vein, I owe the rest of the members of my family an apology and a big thanks for sacrificing their leisure hours and creating an atmosphere congenial for my work. Even the kids, Aditya and Shraman deserve a special mention here.

I am highly obliged to the Head and my colleagues at the Dept. of Computer Science and Application, University of North Bengal, for their support and best wishes.

I would like to express my thanks to Prof. Satadal Mal for his valuable help, suggestions and words of encouragement. I simply cannot forget my dear students - Sujit Ghosh, Kaushik Sarkar and Dhruba Saha, for their contributions during coding and analysis. I am really thankful to them.

During the time I was working on my thesis, I had the opportunity of meeting some eminent people at the national and international seminars, conferences and symposiums that I attended. The interactions and discussions with them opened up my mind's eye to a larger spectrum of ideas that helped in moulding my work, giving it a new direction.

Friends of my circle and relatives deserve my thanks for their encouragement and assistance that made it possible for my work to see the light of the day.

Lastly, I extend my apology to all such individuals, who I might, unwittingly, have forgotten to mention here and thank the almighty God for being with me throughout this endeavour.

Sharad Sinha
(Sharad Sinha)

DECLARATION

I do hereby declare that neither the thesis, titled "*An approach Towards Design and Implementation of Microprocessor-Based Cryptosystems for Secured Transmission*" submitted by me, nor any part thereof has been submitted anywhere for any degree whatsoever.

Sharad Sinha
(Sharad Sinha)

DECLARATION

I do hereby declare that neither the thesis, titled "*An Approach Towards Design and Implementation of Microprocessor-based Cryptosystems for Secured Transmission*" submitted by *Mr. Sharad Sinha*, or any part thereof has been submitted for any degree whatsoever.

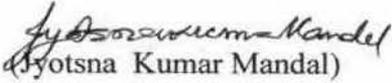

(JYOTSNA KUMAR MANDAL)
SUPERVISOR

TO WHOM IT MAY CONCERN

This is to certify that the matter contained in this thesis is a genuine piece of research work executed by **Mr. Sharad Sinha**, Lecturer, Dept. of Computer Science and Application, University of North Bengal. He has completed this research work, titled **"AN APPROACH TOWARDS DESIGN AND IMPLEMENTATION OF MICROPROCESSOR-BASED CRYPTOSYSTEMS FOR SECURED TRANSMISSION"**, under my supervision fulfilling the requirements of the regulations relating to the nature and the prescribed period of the research work.

His name has been registered for the degree of Ph. D. (Computer Science) in the University of North Bengal with effect from July 19, 2004 according to the letter from the Registrar with ref. no. 168/Ph.D/CompSc/1655(R)04 dated 12.08.2004.

So far my knowledge is concerned, he was very sincere and laborious in his work and bears a good moral character. I wish him every success in life.


(Jyotsna Kumar Mandal)
Supervisor

Chapter 1: Prime Position Orientations of Bits (PPO)	1
1.1 Introduction	1
1.2 Cryptanalysis	4
1.3 Security of algorithms	6
1.4 Classical cryptosystems	7
1.4.1 Substitution ciphers	8
1.4.2 Transposition ciphers	9
1.4.3 Product ciphers	9
1.5 Few popular algorithms	9
1.5.1 Caesar cipher	10
1.5.2 Rail Fence cipher	10
1.5.3 Vigenère cipher	11
1.5.4 One-Time Pad	12
1.5.5 Rotor Machines	13
1.5.6 Data Encryption Standard (DES)	13
1.5.7 RSA	14
1.6 Confusion and diffusion	15
1.7 Block cipher modes of operation	16
1.7.1 Electronic Code Book (ECB) mode	17
1.7.2 Cipher Block Chaining (CBC) mode	17
1.7.3 Cipher Feed-back (CFB) mode	19
1.7.4 Output Feed-back (OFB) mode	20
1.8 Proposal of the thesis	21
1.8.1 Embedded systems	22
1.8.2 Proposed algorithms	24
1.8.2.1 Prime Position Orientations (PPO)	26
1.8.2.2 Block Exchange Technique (BET)	26
1.8.2.3 Selective Positional Orientation of Bits (SPOB)	27
1.8.2.4 Modulo-Arithmetic Technique (MAT)	27
1.8.2.5 Overlapped Modulo-Arithmetic Technique (OMAT)	27
1.8.2.6 Modified Modulo-Arithmetic Technique (MMAT)	28
1.8.2.7 Bit-pair Operation and Separation (BOS)	28
1.8.2.8 Decimal Equivalent Positional Substitution (DEPS)	28
1.8.3 Methods of evaluation	29
1.8.3.1 Character frequency distribution	29
1.8.3.2 Heterogeneity of the source and encrypted files	29
1.8.3.3 Avalanche test	30
1.8.3.4 Runs test	30
1.8.4 Microprocessor-based implementation	30
1.9 Structure of the thesis	31
Chapter 2: Prime Position Orientations (PPO)	32
2.1 Introduction	32
2.2 The PPO scheme	32
2.3 Example	34
2.4 Discussion	35
2.5 Microprocessor-based implementation	40
2.5.1 Routines for 8-bit block-size	40
2.5.1.1 Routines for 8-bit RSPB	41
2.5.1.2 Routine for 8-bit LSPB	42
2.5.1.3 Routine for 8-bit DSPB	42
2.5.2 Routine for 16-bit (and higher) block-sizes	43
2.5.2.1 Routines for 16-bit (and higher) RSPB	44
2.5.2.2 Routine for 16-bit (and higher) LSPB	46
2.5.2.3 Routine for 16-bit (and higher) DSPB	46

2.6	Results and comparisons	46
2.6.1	Character frequency	47
2.6.2	Chi-Square test and encryption time	52
2.6.3	Avalanche and runs	56
2.7	Conclusion	59

Chapter 3: Block Exchange Technique (BET) 60

3.1	Introduction	60
3.2	The BET scheme	60
3.2.1	Algorithm for BET	60
3.3	Example of BET	61
3.3.1	A discussion	62
3.4	Microprocessor-based implementation	63
3.4.1	Routines for block-size 8 bits	63
3.4.1.1	Routines for 8-bit BET	64
3.4.2	Routine for block-size 16 bits (and higher)	65
3.4.2.1	Routines for 16-bit (and higher) BET	65
3.5	Results and comparisons	67
3.5.1	Character frequency	67
3.5.2	Chi-Square test and encryption time	70
3.5.3	Avalanche and runs	73
3.6	Conclusion	74

Chapter 4: Selective Positional Orientation of Bits (SPOB) 75

4.1	Introduction	75
4.2	The SPOB scheme	75
4.2.1	The rounds of SPOB	75
4.3	Example and discussion	76
4.4	Microprocessor-based implementation	78
4.4.1	Look-up tables for 8-bit SPOB	78
4.4.2	Look-up tables for 16-bit (and higher) SPOB	79
4.5	Results and comparisons	79
4.5.1	Character frequency	79
4.5.2	Chi-Square test and encryption time	82
4.5.3	Avalanche and runs	85
4.6	Conclusion	86

Chapter 5: Modulo-Arithmetic Technique (MAT) 87

5.1	Introduction	87
5.2	The Modulo-Arithmetic Technique	87
5.2.1	Algorithm for MAT	88
5.2.2	The modulo-addition operation	88
5.3	Example of MAT	89
5.4	Microprocessor-based implementation	90
5.4.1	Routine for 8-bit MAT encryption	90
5.4.2	Routine for 8-bit MAT decryption	91
5.4.3	Routine for 16-bit (and higher) MAT encryption	92
5.4.4	Routine for 16-bit (and higher) MAT decryption	92
5.5	Results and comparisons	93
5.5.1	Character frequency	93
5.5.2	Chi-Square test and encryption time	96
5.5.3	Avalanche and runs	99
5.6	Conclusion	100

Chapter 6: Overlapped Modulo-Arithmetic Technique (OMAT)	101
6.1 Introduction	101
6.2 The Overlapped Modulo-Arithmetic Technique	101
6.3 Example of OMAT	102
6.4 Microprocessor-based implementation	103
6.4.1 Routine for 8-bit OMAT encryption	103
6.4.2 Routine for 8-bit OMAT decryption	104
6.4.3 Routine for 16-bit (and higher) OMAT encryption	104
6.4.4 Routine for 16-bit (and higher) OMAT decryption	105
6.5 Results and comparisons	105
6.5.1 Character frequency	105
6.5.2 Chi-Square test and encryption time	108
6.5.3 Avalanche and runs	111
6.6 Conclusion	112
Chapter 7: Modified Modulo-Arithmetic Technique (MMAT)	113
7.1 Introduction	113
7.2 The Modified Modulo-arithmetic Technique	113
7.3 Example of MMAT	114
7.4 Microprocessor-based implementation	115
7.4.1 Routine for 8-bit MMAT encryption	115
7.4.2 Routine for 8-bit MMAT decryption	116
7.4.3 Routine for 16-bit (and higher) MMAT encryption	117
7.4.4 Routine for 16-bit (and higher) MMAT decryption	117
7.5 Results and comparisons	118
7.5.1 Character frequency	118
7.5.2 Chi-Square test and encryption time	121
7.5.3 Avalanche and runs	124
7.6 Conclusion	125
Chapter 8: Bit-pair Operation and Separation (BOS)	126
8.1 Introduction	126
8.2 The BOS technique	126
8.2.1 The algorithm for BOS	126
8.2.2 The bit-pair operations	127
8.3 Example of BOS	128
8.4 Microprocessor-based implementation	129
8.4.1 Routine for 8-bit BOS encryption/decryption	130
8.4.2 Routine for 16-bit BOS (and higher) encryption/decryption	131
8.5 Results and comparisons	132
8.5.1 Character frequency	132
8.5.2 Chi-Square test and encryption time	135
8.5.3 Avalanche and runs	138
8.6 Conclusion	139
Chapter 9: Decimal Equivalent Positional Substitution (DEPS)	140
9.1 Introduction	140
9.2 The DEPS scheme	140
9.2.1 Algorithm for DEPS encryption	141
9.2.2 Algorithm for DEPS decryption	142
9.3 Example of DEPS	143
9.3.1 The process of encryption	143
9.3.2 The process of decryption	144

9.4	Microprocessor-based implementation	145
9.4.1	Routine for 8-bit DEPS encryption	146
9.4.2	Routine for 8-bit DEPS decryption	146
9.5	Results and comparisons	147
9.5.1	Character frequency	147
9.5.2	Chi-Square test and encryption time	150
9.5.3	Avalanche and runs	153
9.6	Conclusion	155
Chapter 10: Cascaded Techniques: Product Ciphers		156
10.1	Introduction	156
10.2	OMAT and BET	156
10.2.1	Character frequency	156
10.2.2	Chi-Square test and encryption time	159
10.2.3	Avalanche and runs	162
10.3	MMAT and DSPB	163
10.3.1	Character frequency	163
10.3.2	Chi-Square test and encryption time	163
10.3.3	Avalanche and runs	168
10.4	Conclusion	169
Chapter 11: Proposed Key Formats		170
11.1	Introduction	170
11.2	Format 1	171
11.3	Format 2	172
11.4	Format 3	174
11.5	Format 4	176
11.6	Conclusion	177
Chapter 12: Concluding Discussions		179
12.1	Introduction	179
12.2	Comparison of character frequencies	185
12.3	Comparison of χ^2 values	186
12.4	Comparison of encryption times	186
12.5	Conclusions	186
Appendix A: C Source Codes		188
Appendix B: 8085 Assembly Language Programs		268
Appendix C: Figures and Tables		306
References		
<i>List of Publications by the Author</i>		

Overview

1.1 Introduction

A few years back people used to say that we are in the age of computers, but now they say that this is the age of computer networks, and more precisely the Internet. The main reason for connecting different computers to form a network is to transmit data for whatever purpose it may be. Thus, this is one form of communication and today most of the computers throughout the world are sharing information. The most widely used method of communicating over distances is through electrical signal, either over cables or through free space using satellite channels or other forms of wireless transmission.

In general, three types of problems are associated with data transmission. The first one is handling of a large volume of data and the second one is ensuring error-free transmission. The third one is the most important aspect of data transmission: a lack of security exists when a volume of data is transferred from its source to the destination if no measure is taken for its security. Data transmission runs a risk of making sensitive information vulnerable to unauthorised access. For one reason or the other, most of the data being transmitted must be kept secret from eavesdroppers.

A very important reason to encode data or messages is to keep them secret. **Cryptology**, the study of systems for secret communications [1,2,3,5], consists of two complementary fields of study: **cryptography**, the design of secret communications systems, and **cryptanalysis**, the study of ways to compromise secret communications systems. An attempted cryptanalysis is called an **attack**. Cryptography, **secret** (crypto-) **writing** (-graphy), is the art or science encompassing the principles and methods of transforming an intelligible message into one that is unintelligible, and then retransforming that message back to its original form [10]. Until late 1970s, cryptographic technology was exclusively used for military and diplomatic purposes. More recently, the widespread use of computers has led to the emergence of a variety

of important new applications of cryptology. Today, business sectors and private individuals have recognised the need to protect valuable information in computer communication networks against unauthorised interception. Cryptography is concerned with developing algorithms which may be used to [1,2,5,6]:

- conceal the context of some message from all except the sender and recipient (**privacy** or **secrecy**) to prevent **eavesdropping**, and/or
- verify the correctness of a message to the recipient (**authentication**) to prevent **tampering**

From e-mail to cellular communication, from secured web access to digital cash, cryptography is an essential part of today's information systems. It helps to provide accountability, fairness, accuracy and confidentiality. It can prevent fraud in electronic commerce and assure the validity of financial transactions. It can prove one's identity and protect one's anonymity. It can keep away vandals from altering one's web page and prevent industrial competitors from reading one's confidential documents. As commerce and communications continue to move to computer networks as e-commerce, cryptography is becoming vital issue in communication. These electronic commerce schemes may fall fraud through forgery, misrepresentation, denial of service and cheating if we do not add security to these systems. In fact, computerization makes the risks even greater by allowing attacks that are impossible in non-automated systems. Only strong cryptography can protect against these attacks.

Everything that goes into providing a means for secure communication are collectively called a **cryptosystem** [4,8,38,39]. A structure of a typical cryptosystem is given in figure 1.1. The **sender** sends a message, called the **plain-text**, to the **receiver** by transforming the plain text into a secret form suitable for transmission, called the **cipher-text**, using a cryptographic algorithm, called the **encryption method** (or a **cipher**), and some **key** parameters. To read the message, the receiver must have a matching cryptographic algorithm, called the **decryption method**, and some **key** parameters, which will transform the cipher-text back into the

plain-text, the message. Keys are important parts of cryptographic algorithms. A cryptographic key is somewhat like a physical key used to lock and unlock a door [7].

Mathematically, a cryptosystem is a 5-tuple (E, D, M, K, C) , where M is the set of plain-texts, K the set of keys, C is the set of cipher-texts, $E: M \times K \rightarrow C$ is the set of enciphering functions, and $D: C \times K \rightarrow M$ is the set of deciphering functions. In other words, $E_K(M) = C$ and $D_K(C) = M$ where K is the key [6]. The key may be same or different for encryption and decryption depending on the type of the algorithm.

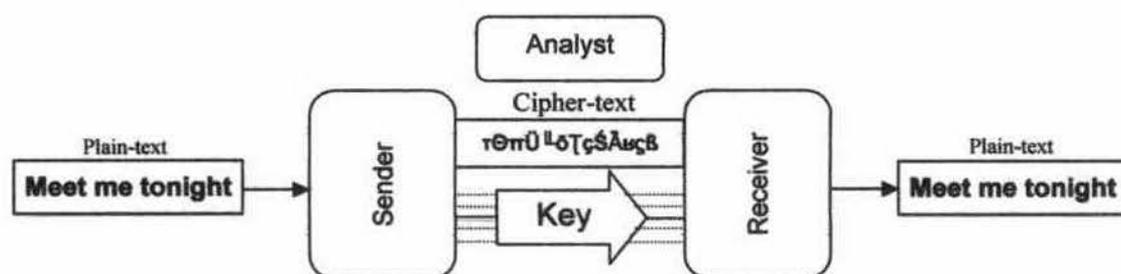


Figure 1.1: A structure of a typical cryptosystem

There are two general types of key-based algorithms: **symmetric** and **asymmetric** [1,2,6,7,9]. In symmetric algorithms, the decryption key can be calculated from the encryption key and vice versa. In most symmetric algorithms, the encryption key and the decryption key are the same. These algorithms are also called **secret-key/single-key/one-key** algorithms. Asymmetric algorithms (also called **public-key** algorithms) are designed so that the encryption key is different from the decryption key. Furthermore, the decryption key cannot be calculated from the encryption key. These algorithms are called "public-key" because the encryption can be made public and only a specific person with the corresponding decryption key (often called **private-key**) can decrypt a message. Sometimes, messages may be encrypted with the private key and decrypted with the public key like in the case of digital signatures.

Symmetric algorithms can be divided into two categories [1]:

- (i) **Stream algorithms** or **stream ciphers**: operate on the plain-text a single bit (or sometimes a byte) at a time.

- (ii) **Block algorithms** or **block ciphers**: operate on the plain-text in groups of bits. A typical block size is 64 bits – large enough to preclude analysis and small enough to be workable.

The goal of cryptography, as already stated, is to keep information secret by enciphering. Standard cryptographic practice is to assume that an adversary, who wishes to break a cipher, has complete access to the communications between the sender and the receiver and knows the algorithm used to encrypt the plain-text, but not the specific cryptographic key. The process of trying to break a cipher is termed as 'cryptanalysis'.

1.2 Cryptanalysis

Cryptanalysis is the science of recovering the plain-text or the key [1,2]. It also may find weaknesses in a cryptosystem. An attempted cryptanalysis is called an '**attack**'. A fundamental assumption in cryptanalysis is that the secrecy must reside entirely on the key.

There are four general types of cryptanalytic attacks [1]. Of course, each of them assumes that the cryptanalyst has complete knowledge of the encryption algorithm used.

- a) ***Cipher-text only attack***: The adversary (or cryptanalyst) has only the cipher-texts of several plain-texts. The goal is to find as many plain-texts as possible, or yet better to deduce the key(s) too.
- b) ***Known plain-text attack***: The cryptanalyst has, not only the cipher-texts of several plain-texts, but also the corresponding plain-texts. The goal is to deduce the key(s) used to encrypt the plain-texts or an algorithm to decrypt any new cipher-texts that have been encrypted using the same key(s).
- c) ***Chosen plain-text attack***: The cryptanalyst may ask that specific plain-texts be encrypted. The chosen plain-texts might yield more information about the key. After getting the corresponding cipher-texts, the goal is to deduce the key(s) used to encrypt the chosen

plain-texts or an algorithm to decrypt any new messages encrypted with the same key(s).

- d) ***Adaptive chosen plain-text attack:*** It is a special case of chosen plain-text attack. In addition to having the freedom to choose the plain-text that is encrypted, the cryptanalyst can also modify his choice based on the results of previous encryption.

There are at least three other types of cryptanalytic attacks [1].

- a) ***Chosen cipher-text attack:*** The cryptanalyst can choose different cipher-texts to be decrypted and has access to the decrypted plain-text. For example, the cryptanalyst has access to a tamperproof box that does automatic decryption. The goal is to deduce the key. A chosen cipher-text attack may be used in combination with a chosen plain-text attack, which is sometimes known as ***chosen text attack***.
- b) ***Chosen key attack:*** This attack does not mean that the cryptanalyst can choose the key, but has some knowledge about the relationship between different keys. This type of attack is not very practical.
- c) ***Rubber-hose cryptanalysis:*** The cryptanalyst threatens, blackmails, bribes, or tortures someone until the key is surrendered.

Attacks use both mathematics and statistics. The statistical methods make use of the assumptions about the statistics of the plain-text language and examine the cipher-text to correlate its properties. Keeping the algorithm's insides secret does not improve the security of the cryptosystem. It is good to let the academic community analyze the strength of the cryptosystem. The best algorithms available today are the ones that have been made public, have been attacked by the world's best cryptographers for years, and are still unbreakable. Most of those who claim to have an unbreakable cipher, simply because they cannot break it, are fools. Good cryptographers rely on peer review to separate the good algorithms from the bad.

1.3 Security of algorithms

Different cryptosystems have different degrees of security and depends on how hard they are to break. If an algorithm resists all attacks, it can be considered secure in practice. An algorithm is said to be 'probably' safe [3] if –

- a) the cost of breaking the algorithm is greater than the value of the encrypted data.
- b) the time required to break the algorithm is longer than the time the encrypted data must remain secret.
- c) the amount of data encrypted with a single key is less than the amount of data necessary to break the algorithm.

The word 'probably' has been used because there is always a chance of breakthroughs in cryptanalysis. On the other hand, the value of most data decreases over time.

Using any kind of cryptanalytic attack, an algorithm may be broken in one of the following ways listed in decreasing order of severity [1,2].

- a) **Total break:** A cryptanalyst finds the key K , such that $D_K(C) = P$.
- b) **Global deduction:** A cryptanalyst finds an alternative algorithm, say A , equivalent to $D_K(C)$, without knowing K .
- c) **Instance (or local) deduction:** A cryptanalyst finds the plain-text of an intercepted cipher-text.
- d) **Information deduction:** A cryptanalyst gains some information about the key or plain-text. This information could be a few bits of the key, some information about the form of the plain-text, and so on.

In most cases, it is not possible to prove that a cipher is secure. Ciphers are either **unconditionally** or **computationally** secure [1].

An algorithm is **unconditionally secure** if, no matter how much cipher-text or computing power is available, there is not enough information to recover the plain-text. In fact, given infinite resources, the one-time pad is the only known unconditionally secure cipher which requires keys as long as messages.

An algorithm is considered **computationally secure** (or strong) if the cryptanalyst's task is made computationally feasible. In other words, a computationally secure algorithm cannot be broken with available resources, either current or future. This means that it requires an infeasible amount of computing resources to break. The theory of computational complexity is still inadequate to demonstrate the computational infeasibility of any cryptosystem.

As an alternative, cipher security is established on the basis of certification methods which involve subjecting the cipher to various cryptanalytic attacks under circumstances considered most favourable to the cryptanalyst. The cryptosystem is then certified to be secure from the specific type of attack.

1.4 Classical cryptosystems

Cryptography had emerged as an 'art' (of hiding), but with the advent of computers it has become a 'science'. More recently, with the explosive growth of computers and the Internet, cryptography has become the only alternative to protect information during transmission. Modern cryptosystems have evolved from the secret codes of decades past, brilliantly augmented with a deep knowledge of modern mathematics. It is worth looking into the past systems before moving forward with the purpose of this thesis.

Almost all the early cryptosystems are symmetric, that is, having the same key for encryption and decryption. Hence, the term 'classical' has been used synonymously with the word 'symmetric'. Before computers, cryptography consisted of encryption by manipulating characters. Modern algorithms work on bits instead of characters. Things are more complex today, but the philosophy remains the same.

Different classical ciphers either substituted characters for one another or transposed characters with one another. The better algorithms did both. Hence they have been categorised accordingly as substitution ciphers, transposition ciphers and product ciphers.

1.4.1 Substitution ciphers

A substitution cipher replaces (or substitutes) each character/bit of the plain-text by another character/bit to form the cipher-text [1,2,6]. The receiver has to invert the substitution on the cipher-text to recover the plain-text.

In classical cryptography, substitution ciphers are grouped into four different types [1].

- a) **Monoalphabetic**: In a monoalphabetic cipher, or a simple substitution cipher, there is a one-to-one correspondence between a character and its substitute. The cryptograms in newspapers are examples of simple substitution ciphers.
- b) **Homophonic**: It is like a simple substitution cipher except that a single character of the plain-text can map to one of the several characters of the cipher-text. The Beale cipher of the 1880s is an example of homophonic substitution cipher.
- c) **Polygram**: In this type of substitution ciphers, a group of characters are substituted at the same time by another group of characters using a key. For example, 'ABC' could correspond to 'HJL' whereas 'ABB' could be replaced by 'VGT'. When ' N ' characters are substituted at a time, the cipher is called an N -gram substitution cipher. The Playfair cipher is a 2-gram substitution cipher and was used by the British during World War I. Hill cipher is a more general N -gram substitution cipher employing linear transformations.
- d) **Polyalphabetic**: This type of substitution cipher is made up of multiple simple substitution ciphers involving the use of different keys. The development of polyalphabetic ciphers began with Leon Alberti in 1568 who invented a cipher disk that defined multiple substitutions.

1.4.2 Transposition ciphers

Transposition ciphers are the ones that change the positions of characters of the plain-text to produce the cipher-text [10]. In other words, the characters of the plain-text remain the same, but the order is shuffled around. Thus, transposition ciphers consist of rearrangements of the plain-text characters. The oldest known transposition cipher is the Scytale cipher used by ancient Greeks as early as 400 B.C.

1.4.3 Product ciphers

Substitution and transposition ciphers are not always enough to conceal a message. They may be broken very easily using statistical analysis. To enhance the level of security these ciphers are sometimes combined to form a product cipher [10]. A product cipher is a combination of two or more ciphers in a cascaded manner such that the cipher-text of one cipher becomes the plain-text for the next one. The combination is done to make the final product superior to that of any one of its components. A product cipher involves the steps of both substitution and transposition. An early product cipher is the German ADFGVX cipher used in World War I [10]. It is a transposition cipher combined with a simple substitution. It was a very complex algorithm for its day but was broken by Georges Painvin, a French cryptanalyst.

1.5 Few popular algorithms

In this section, a few popular algorithms are discussed from different perspectives along with their merits and demerits, clearly indicating the evolution in minimizing the chance of breaking ciphers. After all, these algorithms laid the foundation for today's efficient ciphers. Most of these algorithms have been used before the advent of computers. Apart from the ones discussed here, there are so many other ciphers worth mentioning here but not to be discussed since it is beyond the scope of this thesis. Triple DES, AES, FEAL, IDEA, Blowfish, RC5, CAST-128, and so on, are very few names taken from a huge repository of such algorithms. Only few algorithms have been taken for discussion to get an idea of some problems associated with them.

1.5.1 Caesar cipher

The Caesar cipher is one of the most widely known ciphers. It is a fixed-key monoalphabetic substitution cipher with the i^{th} letter of the plain-text replaced by the $(i + k)^{\text{th}}$ letter modulo 26, since the number of English alphabets is 26 [10]. Julius Caesar used this cipher with $k=3$. The following example illustrates Caesar's encryption.

Plain-text : EVERY DOG HAS A DAY
 Key : Use $k=3$ over English alphabet
 Cipher-text : HYHUB GRJ KDV D GDB

This cipher is very easy to break since the key-space is very small. Trying all the 26 possible keys the cipher may be easily broken. This type of cryptanalysis is known as *brute-force* attack. During decryption, the i^{th} letter of the cipher-text replaced by the $(26 + i - k)^{\text{th}}$ letter modulo 26. This type of cipher can also be broken by statistical frequency analysis.

1.5.2 Rail Fence cipher

This is one of the simplest forms of transposition cipher. The cipher-text is composed by writing the plain-text in two rows, proceeding down, then across, and reading it across, then down [6].

For example, the plain-text 'BLOCK CIPHERS' would be written as:

BOKIHR
 LCCPES

As a result, the cipher-text is:

BOKIHLCCPES

Mathematically, the key to transposition cipher is the permutation function. Since the permutation does not alter the frequency of plain-text characters, a transposition cipher can be detected comparing character frequencies with a model of the language.

1.5.3 Vigenère cipher

Vigenère cipher is a polyalphabetic substitution cipher that chooses a sequence of keys, represented by a string. The key letters are applied to successive plain-text characters, and when the end of the key is reached, the key starts over. Table 1.1 shows the Vigenère tableau to implement this cipher [6].

Table 1.1: Vigenère Tableau

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>	<i>j</i>	<i>k</i>	<i>l</i>	<i>m</i>	<i>n</i>	<i>o</i>	<i>p</i>	<i>q</i>	<i>r</i>	<i>s</i>	<i>t</i>	<i>u</i>	<i>v</i>	<i>w</i>	<i>x</i>	<i>y</i>	<i>z</i>
<i>a</i>	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
<i>b</i>	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
<i>c</i>	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
<i>d</i>	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
<i>e</i>	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
<i>f</i>	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
<i>g</i>	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
<i>h</i>	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
<i>i</i>	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
<i>j</i>	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
<i>k</i>	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
<i>l</i>	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
<i>m</i>	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
<i>n</i>	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
<i>o</i>	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
<i>p</i>	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
<i>q</i>	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
<i>r</i>	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
<i>s</i>	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
<i>t</i>	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
<i>u</i>	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
<i>v</i>	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
<i>w</i>	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
<i>x</i>	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
<i>y</i>	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
<i>z</i>	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

For example, let the message be 'THE BOY HAS THE BAG' and let the key be the word 'VIG'. Then the message will be enciphered as follows.

Key	V I G V I G V I G V I G V I G
Plain-text	T H E B O Y H A S T H E B A G
Key	O P K W W E C I Y O P K W I M

The process of encryption is very simple: Given a key letter x and a plaintext letter y , the cipher-text letter is at the intersection of the row labelled x and the column labelled y ; in this case the cipher-text letter is V . Decryption also is usually simple. The key letter identifies the row. The position of the cipher-text letter in that row determines the column, and the plaintext letter is at the top of that column.

The strength of this cipher is that there are multiple cipher-text letters for each plaintext letter, one for each unique letter of the keyword. Thus the letter frequency information is obscured. For many years, the Vigenère cipher was considered unbreakable. However, even this scheme was vulnerable to cryptanalysis. A Prussian cavalry officer named Kasiski noticed that repetitions occur in the cipher-text when characters of the key appear over the same characters in the plain-text. In the above example, the string 'OPK' appears twice. Both the occurrences are caused by the key sequence 'VIG' enciphering the same set of plain-text characters 'THE'.

Statistical techniques can also be applied for cryptanalysis of this cipher since the key and the plaintext share the same frequency distribution of letters.

1.5.4 One-Time Pad

This cipher is an enhancement to the Vigenère cipher in which the key is a random sequence of characters and is not repeated. One-time pad is unbreakable because there is not enough information whatsoever in the cipher-text to determine the plain-text or the key uniquely [1,2,6,7]. It produces random output that bears no statistical relationship to the plaintext.

The only problem associated with the one-time pad technique is the length and the randomness of the key used. The key required in this cipher grows linearly with the plain-text length, which limits its use for practical purposes. Any heavily used system might require millions of random characters on a regular basis. Supplying truly random characters in this volume is a significant task. The problem of key distribution and protection is more daunting. For any message to be transmitted, a key of the equal length is needed by both sender and receiver. Thus a mammoth key distribution problem exists. Due to these difficulties, despite its effectiveness of the

highest level, it is of limited utility and is useful primarily for low-bandwidth channels requiring very high security.

The one-time pad was first used in the **Vernam Cipher**, designed by Gilbert Vernam in 1917, in which the key bits were added modulo 2 to the plain-text bits on a bit-by-bit basis. If $X = x_1x_2x_3\dots$ denotes the bit-stream of some plain-text and the bit-stream $K = k_1k_2k_3\dots$ is the key, then the Vernam cipher produces a cipher-text bit-stream $Y = y_1y_2y_3\dots$, where $y_i = (x_i + k_i) \bmod 2$ for $i = 1, 2, 3, \dots$. The Vernam cipher can be efficiently implemented by introducing XOR of each pair of plain-text and key bits such that $y_i = (x_i \oplus k_i)$.

1.5.5 Rotor machines

The rotor ciphers were the most important cryptographic devices use in World War II, and remained dominant at least until the late 1950s [1]. The American **Sigaba**, the British **Typex**, the German **Enigma**, and the Japanese **Purple**, were all rotor machines.

A rotor machine has a keyboard and series of rotors and implements a version of the Vigenère cipher. Each rotor is an arbitrary permutation of the alphabet, has 26 positions and performs a simple substitution. In some implementations, the plain-text characters are also permuted before or after substitution, thus making a product cipher. The output pins of one rotor are connected to the input pins of the next. It is the combination of several rotors and the gears moving them that make the machine secure.

1.5.6 Data Encryption Standard (DES)

The Data Encryption Standard (DES) is a bit oriented product cipher and was designed to encipher sensitive but non-classified data [1,3,6]. Its input, output, and key are each 64 bits long. The sets of 64 bits are referred to as blocks. The cipher consists of 16 rounds, or iterations. Each round uses a separate key of 48 bits which is generated from the key block by dropping the parity bits, permuting the bits, and

extracting 48 bits. During decryption the order in which the round keys are used is reversed.

The rounds are executed sequentially, the input to it being the output of the previous round. The right half of the input and the round key are run through a function f that produces 32 bits of output. This output is XORed with the left half, and the resulting left and right halves are swapped.

The function f provides the strength of the DES. The right half (32 bits) of the input is expanded to 48 bits and XORed with the round key. The resulting bits are divided into 8 sets of 6 bits each, and each set is put through a substitution table called the *S-box*. Each *S-box* produces a 4-bit output. The outputs from these boxes are concatenated, and then permuted, to form a single 32-bit output of the function f .

There are so many aspects of the DES algorithm to be discussed, but it is beyond the scope of this thesis. Although it has suffered and endured, to some extent, cryptanalysis throughout the world, it is one of the most important classical ciphers in the history of cryptography. It provided the impetus for many advances in the field and laid the theoretical and practical groundwork for many other ciphers. The concepts of *linear* and *differential cryptanalysis* were developed by researchers while analyzing the DES.

Triple DES, a variation of DES, has been used as a benchmark in this thesis.

1.5.7 RSA

RSA, named after its inventors Rivest, Shamir and Adleman, was proposed in 1977 shortly after the emergence of public-key cryptography [3,6]. It is an exponentiation cipher. The public key consists of a pair of integers (n, e) where n , the RSA modulus, is a product of two large randomly selected (and secret) prime numbers p and q of the same bit-length. The encryption exponent, e , is an integer satisfying $1 < e < \phi$ and $\gcd(e, \phi) = 1$ where $\phi = (p - 1)(q - 1)$. The private key d , also called decryption exponent, is the integer satisfying $1 < d < \phi$ and $ed \bmod \phi = 1$. It has

been proven that the problem of determining the private key d from the public key (n, e) is computationally equivalent to the problem of determining the factors p and q of n . If m is the plain-text and c is the cipher-text, then $c = m^e \bmod n$ and $m = c^d \bmod n$.

The larger the number of bits in e and d , the more secure the algorithm. However, since the calculations involved, both in key generation and in encryption/decryption, are complex, the larger size of the key will make the system run slow. Most discussions of the cryptanalysis of RSA have focussed on the task of factoring n into its two prime factors. For a large n with large prime factors, factoring is a hard problem. Currently, a 1024-bit key size (about 300 decimal digits) is considered strong enough for virtually all applications.

1.6 Confusion and diffusion

Claude Shannon introduced the concepts of confusion and diffusion, which are significant from the perspective of the computer-based cryptographic techniques. Confusion and diffusion are two very important security principles for block ciphers. More than fifty years after these principles were first written, they remain the cornerstone of good block cipher designs [5].

Confusion serves to hide any relationship between the plain-text, the cipher-text and the key [1]. It is a technique of ensuring that a cipher-text gives no clue about the original plain-text. This is to try and frustrate the attempts of a cryptanalyst to look for patterns in the cipher-text, so as to deduce the corresponding plain-text. If a cipher does not satisfy confusion, it can be broken using statistical properties of the plaintext blocks, like frequency analysis. Confusion is achieved by means of substitution techniques. Even bit-level transpositions (or permutations) on a large block of bits can create some confusion on the character level. Attacks like linear and differential cryptanalyses can exploit even a slight relationship between the plain-text, cipher-text and the key. Good confusion makes the relationship statistics very complicated.

Diffusion increases the redundancy of the plain-text by spreading it across the cipher-text [1]. It spreads the influence of individual plain-text and key bits over as

much cipher-text as possible. With good diffusion, minor changes in the plain-text or the key should result in major and random looking changes to the cipher-text. Diffusion can be accomplished by permutation (transposition) techniques.

Stream cipher relies only on confusion whereas block cipher uses both confusion and diffusion. Although confusion alone may be enough for security, but it will require lots of memory to implement. A good practice is to repeatedly mix smaller confusion, requiring lesser memory, and diffusion in a single cipher in different combinations. For example, in the function f of the DES algorithm, the expansion permutation and P-box perform diffusion, and the S-boxes perform confusion.

1.7 Block cipher modes of operation

Reasonable amount of plain-text in general and computer communication data in particular will always yield repeated sequences or patterns. If a data stream is encrypted block by block, patterns in the plain-text may produce statistically significant patterns in the cipher-text. These patterns can give an attacker the entering wedge needed for an attack. Another possibly worse problem in commercial applications is that an attacker with some knowledge of message contents can substitute encrypted blocks from one message for those of another one. This type of attack is also called a '*cut-and-paste*' attack [8].

The term *cipher mode* refers to a set of techniques used to apply a block cipher to a data stream. Several modes of operation have been developed to disguise repeated plain-text blocks and otherwise improve the security offered by block ciphers [8]. A mode of operation may include a combination of a series of basic algorithm steps on a block cipher, and some kind of feedback from the previous step. Theoretically, there could be countless different ways of combining and feeding the inputs and outputs of a cipher. However, in practice, four basic modes are used, viz. Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feed-back (CFB), and Output Feed-back (OFB). The first two modes operate on block cipher, whereas the latter two modes are block cipher modes that can be used as if they are working on stream cipher. These modes are described briefly in the following sub-sections.

1.7.1 Electronic Code Book (ECB) mode

Electronic Code Book (ECB) is the simplest and most trivial mode of operation [5]. The operation is illustrated in figure 1.2. In this mode, the cipher is simply applied to the plain-text block by block. It is the fastest mode of operation and can be sped up by using parallel hardware. It does not require extra bits for seeding a feedback loop. However, some padding bits may be needed to guarantee that full blocks are provided for encryption and decryption.

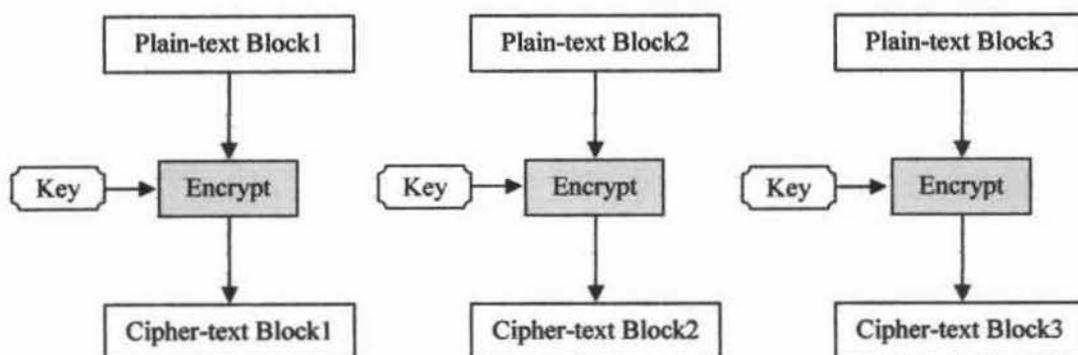


Figure 1.2: The ECB mode of operation

ECB has security problems that limit its usability in practice. Repeated patterns in the plain-text can yield repeated patterns in the cipher-text. It is also easy to modify a cipher-text message by adding, removing, or switching encrypted blocks. Therefore, ECB is suitable only for encrypting small messages, where the scope for repeating the same plain-text blocks is quite less.

1.7.2 Cipher Block Chaining (CBC) mode

To overcome the problem of the ECB mode, the Cipher Block Chaining (CBC) mode ensures that even if a block of plain-text repeats in the input, the identical plain-text blocks yield totally different cipher-text blocks in the output [5]. The operation is depicted in figure 1.3. This mode of operation hides patterns in the plain-text by systematically combining (XOR operation) each plain-text block with the previous cipher-text block before actually encrypting it. The mathematical formula for CBC encryption is:

212074

02 FEB 2009



$$C_0 = IV \text{ and } C_i = E_K(P_i \oplus C_{i-1}) \text{ for } i = 1, 2, 3, \dots, n$$

while the formula for CBC decryption is:

$$P_i = D_K(C_i) \oplus C_{i-1} \text{ for } i = 1, 2, 3, \dots, n \text{ where } C_0 = IV$$

Chaining adds a feedback mechanism to the block cipher. In order to guarantee that there is always some random-looking cipher-text to combine with the actual plain-text, the process is started with a block of random bits called the *initialization vector* (IV). Two messages with identical plain-texts will never yield the same cipher-text as long as the IV is different for each message. Since each cipher-text block acts as an IV for the next plain-text block, and all the cipher-text blocks are sent to the receiver, there is no special reason why the IV for the first block should be kept secret. However, in practice, for maximum security, both the key and the IV are kept secret.

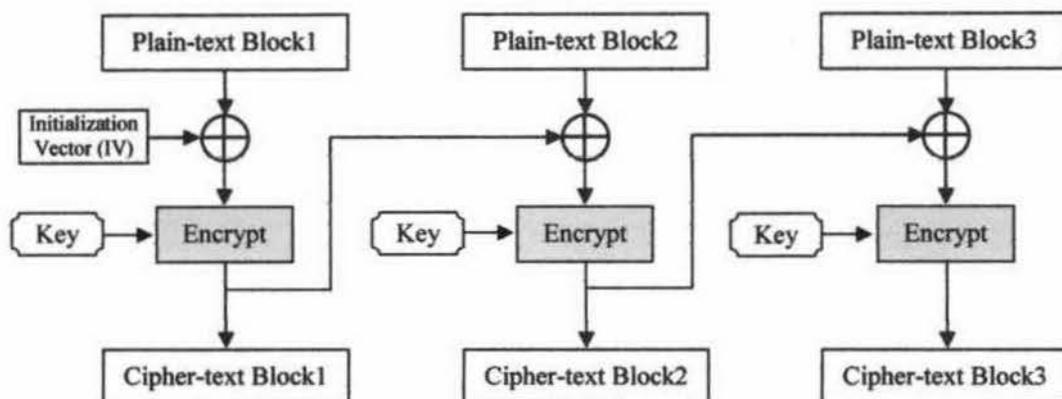


Figure 1.3: The CBC mode of operation

CBC has been the most commonly used mode of operation. Its main drawbacks are that encryption is sequential (i.e., it cannot be parallelized), and that the message must be padded to a multiple of the cipher block size. Hence the cipher-text may be as many as two blocks longer than the plain-text. One block is added to transmit the IV, which will be used again for decryption. The other block contains padding data so that a full block needed by the cipher is encrypted and decrypted.

CBC is a popular mode since it has reasonably good security properties. Patterns in the plain-text are hidden as intended. Furthermore, the interrelationship between data blocks caused by the chaining can make it difficult to modify structured messages in a useful fashion. However, there are risks of 'cut-and-paste' attacks. For

example, the attacker can take a sequence of blocks from a different message encrypted with the same key and insert them into another message. The CBC process will generate a block of garbage text where the data was inserted and then decrypt the rest of the message correctly. A message will also decrypt properly if the attacker truncates it at the beginning or end.

A one-bit change in a plain-text affects all following cipher-text blocks, and a plain-text can be recovered from just two adjacent blocks of cipher-text. As a consequence, decryption can be parallelized, and a one-bit change to the ciphertext causes complete corruption of the corresponding block of plain-text, and inverts the corresponding bit in the following block of plain-text.

1.7.3 Cipher Feed-back (CFB) mode

The cipher feed-back (CFB) mode illustrated by figure 1.4, very much like CBC, makes a block cipher into a self-synchronizing stream cipher [1,5]. However, the plain-text is not encrypted directly in case of CFB. Instead, the block cipher is used to generate a 'constantly changing key' that encrypts the plain-text with a Vernam cipher. In other words, the temporary key is generated by encrypting the previous cipher-text block (or the IV for the first block) with the block cipher and then the plain-text block is encrypted by XORing it with the temporary key. Operation is quite similar to CBC, in particular, CFB decryption is almost identical to CBC decryption performed in reverse:

$$C_0 = IV \text{ and } C_i = E_K(C_{i-1}) \oplus P_i \text{ for } i = 1, 2, 3, \dots, n$$

$$P_i = D_K(C_{i-1}) \oplus C_i \text{ for } i = 1, 2, 3, \dots, n \text{ where } C_0 = IV$$

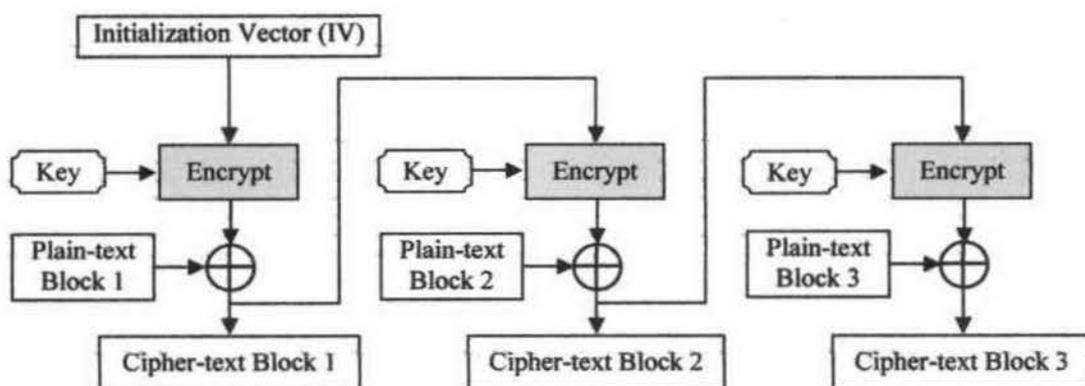


Figure 1.4: The CFB mode of operation

Like CBC mode, changes in the plain-text propagate forever in the ciphertext, and encryption cannot be parallelized, whereas, decryption can be parallelized. When decrypting, a one-bit change in the cipher-text affects two plain-text blocks: a one-bit change in the corresponding plain-text block, and complete corruption of the following plain-text block. Later plain-text blocks are decrypted normally. A form of pipelining is possible since the only encryption step which requires the plain-text is the final XOR.

A particular benefit of CFB is that it does not require padding since it is not limited to the cipher's block size. Hence the mode can be adapted to work with smaller blocks down to individual bits. The resulting cipher-text has comparable security with CBC. In other words, CFB is also vulnerable to 'cut-and-paste' attacks.

1.7.4 Output Feed-back (OFB) mode

The output feed-back (OFB) mode is similar to CFB, but slightly simpler. It makes a block cipher into a synchronous stream cipher: it generates keystream blocks, which are then XORed with the plain-text blocks to get the cipher-text [1,5]. The block cipher is used only to generate a keystream, starting from the IV. A keystream block is obtained by encrypting the previous key block. It does not depend on the data stream at all. Neither the plain-text nor the cipher-text is fed back to affect the encryption process. Because of the symmetry of the XOR operation, encryption and decryption are exactly the same:

$$C_i = P_i \oplus K_i \text{ for } i = 1, 2, 3, \dots, n$$

$$P_i = C_i \oplus K_i \text{ for } i = 1, 2, 3, \dots, n$$

$$K_i = E_K(O_{i-1}) \text{ for } i = 1, 2, 3, \dots, n$$

$$K_0 = IV$$

Each output feedback block cipher operation depends on all previous ones, and so cannot be performed in parallel. However, because the plaintext or ciphertext is only used for the final XOR, the block cipher operations may be performed in advance, allowing the final step to be performed in parallel once the plain-text or the cipher-text is available. The OFB mode of operation has been pictorially demonstrated in figure 1.5.

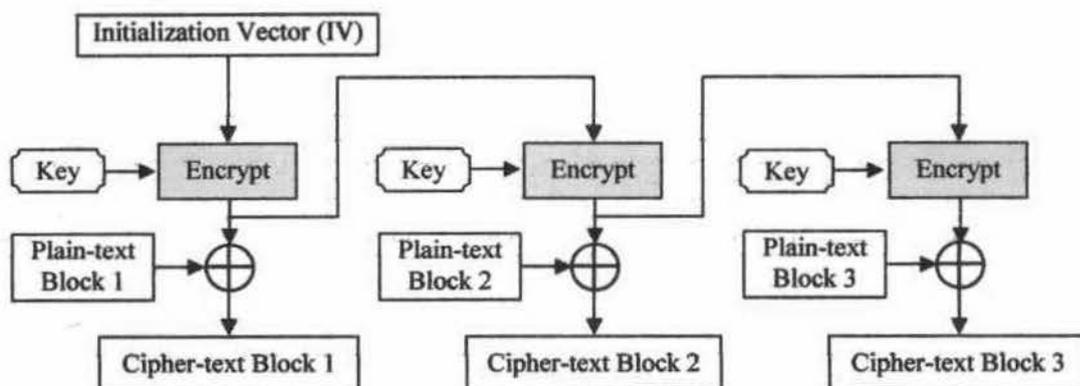


Figure 1.5 The OFB mode of operation

While OFB offers better security properties than ECB, it has shortcomings compared with the other two modes. In particular, there is a direct relationship between the plain-text bits and the cipher-text bits. Hence it is vulnerable to attacks.

1.8 Proposal of the thesis

This research work has been carried out to propose some algorithms to design and implement Intel 8085 microprocessor-based cryptosystems to enhance security in transmission as well as software implementation using C language for analyses of the algorithms developed. The low level implementation may be fabricated in chips and applied in embedded systems, which are being widely used to build small equipment like cell-phones.

Numerous cryptosystems have been developed and implemented over years. Each has its advantages and disadvantages. Generation of crypto models on cryptosystem is a continuous and growing need against the threat of privacy violations. Although there are highly secure algorithms being used today, they cannot be implemented in very small systems having very little memory space and low processing power. There are so many communication devices that use very small microprocessors. Although these devices are the main targets of this research work, the proposed algorithms may well fit in other environments too. Complicated computations have been avoided as far as possible, without compromising too much with the security offered by the algorithms. In case such computations are needed,

alternative techniques have been proposed to get the same results, especially keeping in mind the low-level implementations.

Since most of the devices which have been aimed at by this research work are available in the form of embedded systems, it will be worth presenting here an outline of these systems.

1.8.1 Embedded systems

Most people might think that they have one or two microprocessors at home depending on the number of computers they own. A careful observation may reveal that they may actually have a lot more than that, probably even more than ten or twenty! Today, microprocessors are embedded in almost every electronic appliance one can think of. They have become omnipresent in almost every aspect of our lives.

Embedded systems are electronic devices that incorporate microprocessors within their implementations. The main purposes of the microprocessor are to simplify system design and provide flexibility [18,19]. Having a microprocessor in the device means that removing bugs, making modifications, or adding new features are only matters of rewriting the software that controls the device. However, unlike PCs, embedded systems may not have a disk drive, large memory space or even the operating system. Softwares are mainly stored in ROM chips, which means that modifying the software requires either replacing or reprogramming the ROM.

Embedded systems are found in a wide range of application areas. Table 1.2 illustrates how wide the range of application may be [19]. Originally, embedded systems were used only for expensive industrial-control applications, but as the advancements in technology brought down the cost of dedicated processors, they began to appear in moderately expensive applications such as automobiles, communications and office equipment, and televisions. The latest embedded systems are so inexpensive that they are found in almost every electronic product people use.

In many cases people are not even aware that a microprocessor is present and hence don't realize just how pervasive they have become. Although the typical family

may own only one or two personal computers, the number of embedded computers found within their home, cars, and other personal belongings is much greater. It is quite surprising that embedded processors account for virtually 100% of worldwide microprocessor production [19]. For every microprocessor produced for use in a desktop computer, more than hundred are produced for use in embedded systems.

Table 1.2: Examples of embedded systems

Application area	Examples
Aerospace	Navigation systems, automatic landing systems, flight altitude controls, engine controls, space explorations (e.g. Mars Pathfinder), space telescopes and other astronomical instruments.
Automotive	Fuel injection control, passenger environmental controls, antilock braking systems, air bag controls, GPS mapping.
Children's Toys	Nintendo's - 'Game Boy', Mattel's - 'My Interactive Pooh', Tiger Electronics' - 'Furby'.
Communication	Satellites, network routers, switches, hubs, telecommunication equipment.
Computer Peripherals	Printers, scanners, displays, modems, hard disk drives, CD/DVD drives, plotters.
Home	Dishwashers, washing machines, microwave ovens, VCRs, CD-DVD players, other music systems, televisions, fire/security alarms, lawn sprinkler controls, thermostats, refrigerators, still and video cameras, clock radios, answering machines and many more.
Industrial	Elevator controls, surveillance systems, robots, temperature controllers, IC fabrication systems and hundreds of other industrial equipments.
Instrumentation	Data collection, oscilloscopes, signal generators, signal analyzers, power supplies.
Medical	Imaging systems (e.g., X-Ray, MRI, ultrasound), patient monitors, heart pacers, other equipments used in ICUs, OTs etc.
Office Automation	FAX machines, copiers, telephones, EPBAX, cash registers, card readers (for credit cards etc.).
Personal	Personal Digital Assistants (PDAs), pagers, cell phones, wrist watches, video games, portable MP3 players, GPS.

The process of designing and implementing software for embedded systems is quite different from writing application programs for PCs. Writing embedded application programs presents new challenges regarding reliability, performance, and cost.

Reliability expectations place greater responsibility on programmers to eliminate bugs and to design the software to tolerate errors and unexpected situations. Many embedded systems have to run 24 hours a day, and 365 days a year. One cannot just 'reboot' when something goes wrong. For this reason, good coding practices and thorough testing take on a new level of importance in the realm of embedded systems.

Performance goals force the programmers to have a better understanding of how alternative methods for performing input and output provide opportunities to trade speed, complexity, and cost. Although high-level languages are generally used for better productivity, the programmer may have to drop to the low level and program directly in assembly language. Further, the programmer has to lie within the limitations regarding the range and resolution of numbers, memory space and the like.

The cost factor has actually steered the way how embedded systems are designed and produced. Unlike processors embedded in large, expensive systems such as medical equipment, consumer products are designed to be mass produced with a minimal cost.

In short, programming an embedded system is somewhat like PC programming 20 years ago. The hardware for the system is usually chosen to make the device as cheap as possible. This means that the programmer has to compromise with slow processors and low memory, while at the same time battling a need for efficiency not seen in normal PC applications. Apart from processor and memory, other resources programmers expect may not even exist in embedded systems. For example, most embedded processors do not have floating point hardware units. These resources either need to be emulated in software, or avoided altogether.

1.8.2 Proposed algorithms

As per the scenario discussed in the preceding section, special considerations have been made for the proposed algorithms to be as simple as possible so as to run efficiently with slow processors and low memory. At the same time, attention has been given to design robust algorithms to face the challenges of various types of cryptanalytic attacks.

During this research work, seven different techniques/algorithms have been developed and implemented through an Intel 8085 microprocessor-based system, which are listed below.

- [1] **Prime Position Orientations (PPO)**
- [2] **Block Exchange Technique (BET)**
- [3] **Selective Positional Orientation of Bits (SPOB)**
- [4] **Modulo-Arithmetic Technique (MAT)**
- [5] **Overlapped Modulo-Arithmetic Technique (OMAT)**
- [6] **Modified Modulo-Arithmetic Technique (MMAT)**
- [7] **Bit-pair Operation and Separation (BOS)**
- [8] **Decimal Equivalent Positional Substitutions (DEPS)**

The PPO, BET and SPOB are transposition ciphers, whereas the rest are substitution ciphers. The BOS technique also involves some transposition operations in addition to substitutions.

All the proposed algorithms are block ciphers and have been implemented for bit streams of 512 bits. The algorithms can be easily enhanced to work with a higher stream size for better security. All these ciphers have several rounds, each round working on a particular block size. The encryption/decryption is started with a small block-size, say 8bits, and doubling it in each round thereafter reaching the last round with block-size 256 or 512. In some cases, the decryption starts with the maximum block-size and moves down to the minimum block size. All the algorithms have been put through several statistical tests for analysing their weaknesses, and hence the strength. A good degree of confusion and diffusion within the bit stream has been noticed in all of these ciphers. The following process was adopted for each proposed block cipher.

- a) Writing a C language program for the purpose of analysis. [33,34]
- b) Writing an 8085 assembly language program for implementation. [20-29]
- c) Analyzing the strength of the cipher by running several test algorithms.

The C language programs were needed for encrypting a set of chosen files and analyzing the blocks ciphers through various perspectives. The methods used for testing the algorithms are discussed at the end of this chapter. Outlines of the proposed block cipher techniques are presented in the following sub-sections.

1.8.2.1 Prime Position Orientations (PPO)

In this technique, the bit stream is first divided into blocks of 8 bits each. The positions of the bits within a block are numbered 1 to 8 starting from the MSB. The bits whose positions are found to be prime numbers (2, 3, 5, and 7 in this case) are picked up for transposition. Three different techniques have been developed depending on whether the prime positional bits are pushed to the front, or the rear, or divided among the front and the rear portions of the block being considered. The three options can be used in a cascaded manner, or one of them can be chosen for each block depending on the key.

The process is repeated for several rounds, each time doubling the block size. It was also noticed that the original block of a particular round would be regenerated if that round was reiterated several number of times. Each round was given several iterations and the number of iterations formed a part of the key. During decryption, the key was used to give the remaining iterations to get back the original bit stream.

1.8.2.2 Block Exchange Technique (BET)

In this technique, the encryption is performed through a multi-round cascaded system. In the first round, the message is taken as blocks of 1 bit each, such as (a), (b), (c), (d), (e), (f), (g), (h) and so on, the letters a, b, c, d etc. denoting either 0 or 1. These blocks are divided among several sets, each set containing 4 contiguous blocks. The second and the third elements (blocks) in each set are then exchanged. So the message is converted to (a), (c), (b), (d), (e), (g), (f), (h) and so on. In the second round, the block size is doubled making blocks as {(a, c), (b, d), (e, g), (f, h)}. The same exchange technique is applied once again to this intermediate stream converting the message to (a, c), (e, g), (b, d), (f, h). The process is repeated, each time doubling

the block size, up to 512-bit block size or more. The same process is applied for decryption.

1.8.2.3 Selective Positional Orientation of Bits (SPOB)

In this technique, the stream of bits are divided into a number of blocks each containing n bits, where n is one of 8, 16, 32, 64, 128, 256, or 512, depending on the round. Within each block, a pair of bits is selected using the rules of proposed technique, and swapping is performed among the bits in each pair. The selection of the bits in each pair is altered in every pass. Several passes will constitute a round.

If the whole process is performed repeatedly for a particular block size, the original block is regenerated after a finite number of iterations. One of these iterations is selected to generate the encrypted block and hence the corresponding encryption key. The same operation is performed for decryption.

1.8.2.4 Modulo-Arithmetic Technique (MAT)

In this case, the original message is considered as a stream of bits, which is then divided into a number of blocks, each containing n bits, where n is any one of 8, 16, 32, 64, 128, 256. If $B_1, B_2, B_3, B_4, \dots$ are the blocks in the stream, then they are paired as $(B_1, B_2), (B_3, B_4)$, and so on. The two adjacent blocks in each pair are then added where the modulus of addition is 2^n . The result replaces the second block, first block remaining unchanged. The modulo-addition has been implemented in a very simple manner where the carry out of the MSB is discarded to get the desired result. The technique is applied in a cascaded manner by varying the block size from 8 to 256. The whole technique has been implemented by using a modulo-subtraction technique for decryption.

1.8.2.5 Overlapped Modulo-Arithmetic Technique (OMAT)

This is just an enhanced form of the simple MAT algorithm. In this case, the way the blocks are paired is different. The block-pairs overlap with each other and hence the name of the technique. Operations are carried out in block-pairs

(\bar{B}_1, \bar{B}_2) , (\bar{B}_2, \bar{B}_3) , (\bar{B}_3, \bar{B}_4) , and so on. Except for first and the last block, each block is common to two adjacent pairs of blocks.

1.8.2.6 Modified Modulo-Arithmetic Technique (MMAT)

This is also a modification of the MAT algorithm where the modulo-addition is carried out twice for each pair of blocks. Unlike the MAT algorithm, both the blocks are replaced with the result of the addition. The result of the first addition replaces one block while the other block is replaced by the result of the second addition. The degree of confusion and diffusion is higher than simple MAT.

1.8.2.7 Bit-pair Operation and Separation (BOS)

In this technique also, the original stream of bits is divided into blocks of $n=2^k$ bits each, where k is 3, 4, 5, 6, 7, 8, 9, and so on, for each round. 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 in the front and that of the second operation in the rear. The encryption is started with block size of 8 bits and repeated for several times and the number of iterations also forms a part of the key. The technique is applied in a cascaded manner doubling the block size each time. The same process is used for decryption.

1.8.2.8 Decimal Equivalent Positional Substitutions (DEPS)

In this case also, the original stream of 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. The decimal equivalent of the block under consideration is evaluated and checked whether the integer value is even or odd. The position of that integer in the series of natural even or odd numbers is evaluated. A '0' or '1' is pushed to the output stream depending on whether the integer is even or odd, respectively. The process is carried out recursively with the positional values for a finite number of times, equal to the length of the source block. During decryption, bits in the target block are to be considered along LSB-to-MSB direction after which we get an integer value, the binary equivalent of which is the source block. The sweetness of the technique lies in its

microprocessor-based implementation where no conversion to decimal and no calculations to ascertain whether the decimal value is even or odd and to find its position in the series of odd or even numbers are needed, no matter how long the block is.

1.8.3 Methods of evaluation

Although no standard methods are available for testing the strength of a symmetric cipher, several tests suggested in popular journals/websites [35,36,37] have been used to examine the vulnerability of the encryption algorithms proposed in this thesis. These methods are explained briefly in sections 1.8.3.1 through 1.8.3.4. The results of these tests for the proposed ciphers have been compared with those obtained for Triple DES algorithm, which has been used as a benchmark.

1.8.3.1 Character frequency distribution

One way to judge a symmetric cipher is to examine the frequency distributions of all the 256 ASCII characters in the source and the encrypted files. For this purpose, a number of files in each of the categories like .txt, .exe, .dll, .jpg etc. have been chosen and are subjected to encryption using the proposed algorithms. The frequencies of characters in the source and the encrypted files are computed and then analysed to check whether the characters in the encrypted file are more or less equally distributed in the 0 to 255 range compared to that of the source file. The standard deviation of the distribution may be a good measure of evenness.

1.8.3.2 Heterogeneity of the source and the encrypted files

This test has been applied to test whether the source and the encrypted files are heterogeneous or not. There many ways for comparing any two files. Merely comparing the two files will not be enough. The degree of difference between the files being compared is the actual measure of heterogeneity.

The best way to do this is to compare the frequencies of each character in the source and the encrypted files to ensure there is a good amount of difference. In other

words, the frequencies obtained in the previous test may be used to apply the most popular χ^2 -test (Chi-Square test). A high χ^2 value with high Degree of Freedom (DF) for such a pair of files will indicate heterogeneity between those files [30,31,32].

1.8.3.3 Avalanche test

In this test, a binary string is encrypted several times, each time with a small modification. At first, the original string is encrypted without any modification. In the subsequent steps, the binary string is encrypted for a number of times, equal to the length of the string, each time complementing one bit. The encrypted strings are examined to ensure that the change in one bit of the source string, more or less, affects the whole encrypted string. Simply speaking, this test checks the diffusion property of the encryption algorithm [35].

1.8.3.4 Runs test

It must be ensured that the attempts of a cryptanalyst to look for patterns in the cipher-text, so as to deduce the corresponding plain-text, are frustrated. This is achieved by creating a good amount of confusion as discussed in section 1.6. This test uses the strings obtained in the avalanche test to compare the number of *runs* [35]. A *run* of length i in a binary n -tuple is an i -tuple of consecutive bits (1's or 0's) not preceded or succeeded by the same bit. The number of runs in an n -tuple ranges from 1 to n .

1.8.4. Microprocessor-based implementation

The proposed algorithms have been implemented in an Intel 8085 microprocessor-based system. The schematic view of the implementation is given by the block diagram in figure 1.6.

The incoming bit stream from any source may be captured and stored in the input buffer. The microprocessor-based system takes the input from the buffer, generates cipher-text following the particular algorithm being used and sends to output buffer. The cipher-text may be sent for the destination through a transmission line.

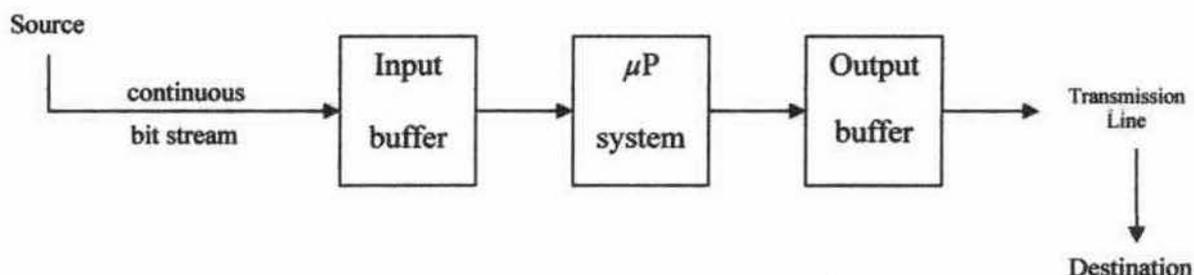


Figure 1.6: A typical microprocessor- based implementation

1.9 Structure of the thesis

The historical aspect of cryptology, the background of the research work, and an overview of the proposals made in the thesis were deliberated in this chapter. The proposed algorithms, namely PPO, BET, SPOB, MAT, OMAT, MMAT, BOS and DEPS, their microprocessor-based implementations, and the results of the various tests conducted along with comparisons are presented from chapter 2 through chapter 9, respectively. The Triple DES algorithm has been used as a benchmark for comparison in these chapters. Chapter 10 discusses the nature of some cascaded combinations of the proposed algorithms. Two such combinations, viz. OMAT+BET and MMAT+DSPB, are taken up as examples in this chapter. The possible structures of the encryption/decryption keys for the various proposed algorithms are presented in chapter 11. Several conclusions are drawn in chapter 12 after elaborate comparisons among the proposed algorithms.

The C programs of the proposed algorithms along with the programs of the various test algorithms are given in Appendix A. The 8085 assembly language programs for the 'microprocessor-based implementations', the main objective of the thesis, are listed in Appendix B. An index of the figures and tables in this thesis are given in Appendix C. References and a separate list of publications by the author are given at the end of the thesis.

Prime Position Orientations (PPO)

2.1 Introduction

In this chapter, a private-key encryption technique has been proposed, where a 512 bit long string of bits (0 and 1) is considered as the input, which represents the plain-text. The stream is divided into a finite number of blocks of equal size, each containing n bits, where n may be any number between 8 and 512 and is an exponent of 2. The bits in each block are numbered, starting at 1 from the left hand side, i.e., the MSB. The bits, each of whose position within a block is a prime number (henceforth called a prime bit), are marked for transposition. The prime bits are then moved towards MSB or LSB, or distributed on the two ends depending on the scheme chosen. Hence, an intermediate block is generated. The operation is repeated for various block-sizes starting from 8 up to 512, each time doubling the block-size.

If the string size is not a multiple of the block-size for various rounds, the remaining bits at the end is taken as a single block and same operations are performed on it. Padding may also be used as an alternative, but it will generate a very little space overhead. The scheme itself does not have any storage overhead. The same procedure is reiterated several times to decrypt the stream.

2.2 The PPO scheme

In *Round 1*, the input string of 512 bit is divided into a finite number of blocks of equal size, each containing 8 bits. Now, in each block, one of the following operations may be performed:

(i) All the prime bits are pushed towards the right hand side, i.e. the LSB, and the rest of the bits (non-prime bits) towards left, to generate an intermediate block. The operation will be, henceforth, called **Right Shift Prime Bits (RSPB)**. The

procedure is iterated for a finite number of times and the number of iterations will form a part of the encryption key, which is discussed in chapter 11.

(ii) All the prime bits are pushed towards the left hand side, i.e. the MSB, and the rest of the bits (non-prime bits) towards right, to generate an intermediate block. The operation will be, henceforth, called **Left Shift Prime Bits (LSPB)**. The procedure is iterated for a finite number of times and the number of iterations will form a part of the encryption key.

(iii) The prime bits are divided into two groups, one being pushed towards the right hand side, i.e. the LSB, and the other towards left, i.e. the MSB. The rest of the bits (non-prime bits) are squeezed in the middle. If the number of prime bits is even, then equal number of prime bits will fall at either end. In case there are odd number of prime bits (say k), then $\text{int}(k/2)$ bits are sent towards left and $\text{int}(k/2)+1$ bits are sent towards right. The operation will be, henceforth, called **Divide Shift Prime Bits (DSPB)**. The procedure is iterated for a finite number of times and the number of iterations will form a part of the encryption key.

(iv) The fourth operation is a mixture of all the above three operations applied in a particular sequence. The procedure is iterated for a finite number of times. The sequence, in which these schemes are applied, and the number of iterations will form a part of the encryption key. Henceforth, **Combined Shift Prime Bits (CSPB)** will be the name given to the operation.

The same procedure is performed in **Round 2**, **Round 3**, and so on, on the intermediate blocks generated in the previous round, each time doubling the block-size till it is 512 in **Round 7**.

For encrypting a particular file, chunks of 512 bits are read and encrypted. The last chunk from the source file may be equal to or less than 512 bits. In the later case, the last block in a particular round may not be equal to the block-size of that round. This situation is handled by taking the remaining bits at the end as a single block and performing the same operations. As an alternative, padding bits may be added to

reach 512 bits, but this will generate some space overhead. The example presented in Section 2.3 will illustrate the above operations for an 8-bit block.

2.3 Example

An 8-bit block, say $S = 10110101$, is taken as the source block. The effects of LSPB, RSPB, DSPB and CSPB on this source block are illustrated in figure 2.1.

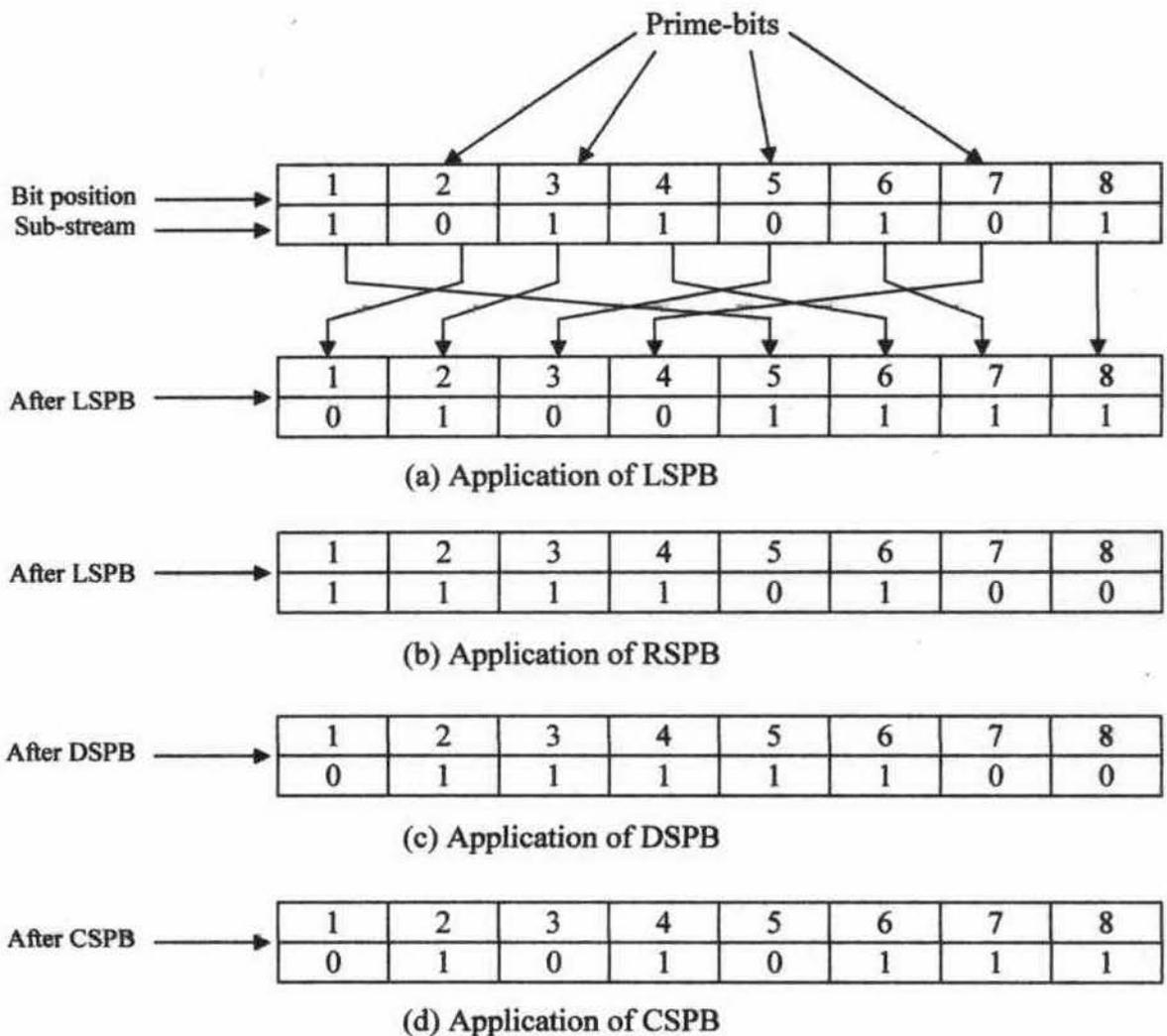


Figure 2.1: Trace of bit movements in PPO

Part (a) of the figure shows the movement of the bits when LSPB is performed. The effect of RSPB is illustrated by part (b) of the figure. Similarly, part (c) and part (d) portray the effects of DSPB and CSPB.

2.4 Discussion

In figure 2.1(a), one can visualise the transpositions of all the bits due to LSPB. For example, bit number 7 moves to position 4, bit number 4 moves to position 6, and bit number 6 moves to position 7. This means, if LSPB is applied again and again, then bit number 7 will come to its original position after 3 iterations. The mapping $7 \Rightarrow 4 \Rightarrow 6 \Rightarrow 7$ will illustrate the transpositions of bit number 7 when LSPB is performed repeatedly. Similarly, the movement of bit number 1 can be depicted by the mapping $1 \Rightarrow 5 \Rightarrow 3 \Rightarrow 2 \Rightarrow 1$ and it will come to its original position in 4 iterations. This shows that the original bit stream is obtained again after some iterative application of LSPB. The mapping in figure 2.2 shows the transpositions of all the 8 bits and the effect of iterative use of the operation.

1	⇒	5	⇒	3	⇒	2	⇒	1	⇒	5	⇒	3	⇒	2	⇒	1	⇒	5	⇒	3	⇒	2	⇒	1
2	⇒	1	⇒	5	⇒	3	⇒	2	⇒	1	⇒	5	⇒	3	⇒	2	⇒	1	⇒	5	⇒	3	⇒	2
3	⇒	2	⇒	1	⇒	5	⇒	3	⇒	2	⇒	1	⇒	5	⇒	3	⇒	2	⇒	1	⇒	5	⇒	3
4	⇒	6	⇒	7	⇒	4	⇒	6	⇒	7	⇒	4	⇒	6	⇒	7	⇒	4	⇒	6	⇒	7	⇒	4
5	⇒	3	⇒	2	⇒	1	⇒	5	⇒	3	⇒	2	⇒	1	⇒	5	⇒	3	⇒	2	⇒	1	⇒	5
6	⇒	7	⇒	4	⇒	6	⇒	7	⇒	4	⇒	6	⇒	7	⇒	4	⇒	6	⇒	7	⇒	4	⇒	6
7	⇒	4	⇒	6	⇒	7	⇒	4	⇒	6	⇒	7	⇒	4	⇒	6	⇒	7	⇒	4	⇒	6	⇒	7
8	⇒	8	⇒	8	⇒	8	⇒	8	⇒	8	⇒	8	⇒	8	⇒	8	⇒	8	⇒	8	⇒	8	⇒	8

Figure 2.2: LSPB iterations on 8-bit block

It is evident from the figure that although bit number 7 and bit number 4 require 3 and 4 iterations, respectively, to come to their original positions, the block of 8 bits is regenerated after 12 iterations. A closer observation reveals that bit number 8 does not change its position at all. It is to be noted that this will not affect the level of security since its position will change in the subsequent rounds when the block size is different.

Now, that the original block is regenerated after 12 iterations, it is not necessary to use another algorithm for decryption. Any intermediate iteration may be selected to produce the cipher-text and decryption can be done by applying the remaining iterations. Hence, the same procedure may be applied for encryption as well as for decryption.

Table 2.1 shows the complete encryption and decryption process of an 8-bit block using LSPB only. Here, the intermediate block S_6 is considered as the encrypted block, although any one of S_1 through S_{11} can be chosen for the purpose. A cycle is formed in 12 consecutive iterations, and as a result of the iterations, the original block is regenerated.

Table 2.1: Encryption/Decryption process of 8-bit block using LSPB

Bit positions	1	2	3	4	5	6	7	8
S (plain-text)	1	1	1	0	0	1	0	1
S_1	1	1	0	0	1	0	1	1
S_2	1	0	1	1	1	0	0	1
S_3	0	1	1	0	1	1	0	1
S_4	1	1	1	0	0	0	1	1
S_5	1	1	0	1	1	0	0	1
S_6 (cipher-text)	1	0	1	0	1	1	0	1
S_7	0	1	1	0	1	0	1	1
S_8	1	1	1	1	0	0	0	1
S_9	1	1	0	0	1	1	0	1
S_{10}	1	0	1	0	1	0	1	1
S_{11}	0	1	1	1	1	0	0	1
S_{12} (plain-text)	1	1	1	0	0	1	0	1

Similarly, the transpositions of all the bits for an 8-bit block due to RSPB can be visualised in figure 2.3.

$1 \Rightarrow 1 \Rightarrow 1$
 $2 \Rightarrow 5 \Rightarrow 7 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 5 \Rightarrow 7 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2$
 $3 \Rightarrow 6 \Rightarrow 3 \Rightarrow 6 \Rightarrow 3 \Rightarrow 6 \Rightarrow 3 \Rightarrow 6 \Rightarrow 3 \Rightarrow 6 \Rightarrow 3$
 $4 \Rightarrow 2 \Rightarrow 5 \Rightarrow 7 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 5 \Rightarrow 7 \Rightarrow 8 \Rightarrow 4$
 $5 \Rightarrow 7 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 5 \Rightarrow 7 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 5$
 $6 \Rightarrow 3 \Rightarrow 6 \Rightarrow 3 \Rightarrow 6 \Rightarrow 3 \Rightarrow 6 \Rightarrow 3 \Rightarrow 6 \Rightarrow 3 \Rightarrow 6$
 $7 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 5 \Rightarrow 7 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 5 \Rightarrow 7$
 $8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 5 \Rightarrow 7 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 5 \Rightarrow 7 \Rightarrow 8$

Figure 2.3: RSPB iterations on 8-bit block

In this case, bit number 1 does not change its position, but as explained earlier, it will not remain the same in subsequent rounds. It is evident that for an 8-bit block, only 10 iterations are required for RSPB to regenerate the original block. Table 2.2

shows the complete encryption and decryption process of an 8-bit block using RSPB only.

Table 2.2: Encryption/Decryption process of 8-bit block using RSPB

Bit positions	1	2	3	4	5	6	7	8
S (plain-text)	1	1	0	1	0	1	0	1
S ₁	1	1	1	1	1	0	0	0
S ₂	1	1	0	0	1	1	1	0
S ₃	1	0	1	0	1	0	1	1
S ₄	1	0	0	1	0	1	1	1
S ₅ (cipher-text)	1	1	1	1	0	0	0	1
S ₆	1	1	0	1	1	1	0	0
S ₇	1	1	1	0	1	0	1	0
S ₈	1	0	0	0	1	1	1	1
S ₉	1	0	1	1	0	0	1	1
S ₁₀ (plain-text)	1	1	0	1	0	1	0	1

Since the number of iterations to regenerate the original block in this case is 10, any one of S₁ through S₉ may be chosen as the cipher-text.

For DSPB, as in the case of LSPB, the number of iterations to come back to the original block is 12 as is evident from the mapping depicted in figure 2.4.

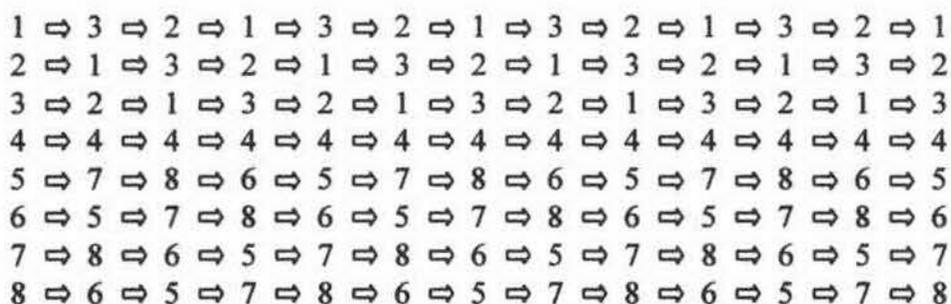


Figure 2.4: DSPB iterations on 8-bit block

In case of DSPB, the position of bit number 4 remains unchanged, and as already explained, it will alter in the subsequent rounds. All the 12 iterations for a block of 8 bits due to DSPB are shown in table 2.3. As in LSPB, any one of S₁ through S₁₁ may be selected as the cipher-text since the number of iterations to regenerate the original 8-bit block is 12.

Table 2.3: Encryption/Decryption process of 8-bit block using DSPB

Bit positions	1	2	3	4	5	6	7	8
S (plain-text)	1	1	0	1	0	1	0	1
S ₁	1	0	1	1	1	1	0	0
S ₂	0	1	1	1	1	0	1	0
S ₃	1	1	0	1	0	0	1	1
S ₄	1	0	1	1	0	1	0	1
S ₅	0	1	1	1	1	1	0	0
S ₆ (cipher-text)	1	1	0	1	1	0	1	0
S ₇	1	0	1	1	0	0	1	1
S ₈	0	1	1	1	0	1	0	1
S ₉	1	1	0	1	1	1	0	0
S ₁₀	1	0	1	1	1	0	1	0
S ₁₁	0	1	1	1	0	0	1	1
S ₁₂ (plain-text)	1	1	0	1	0	1	0	1

In CSPB, all of LSPB, RSPB, and DSPB are performed in a cascaded manner. The original block is subjected to LSPB. RSPB is performed on the intermediate block generated by LSPB. Finally the block is subjected to DSPB. This combined sequence of operations completes just one cycle. The whole process is repeated in the subsequent iterations. The sequence in which the individual operations are performed may be altered to get a different result. One may experiment other possibilities by applying only two of the operations instead of all the three. For the purpose of this thesis, CSPB means combination in the order LSPB, then RSPB, and then DSPB. Figure 2.5 illustrates the whole lot of iterations required to regenerate the original block of 8 bits.

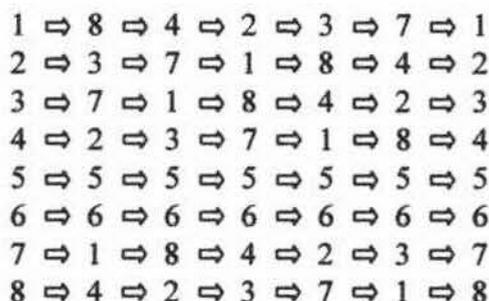


Figure 2.5: CSPB iterations on 8-bit block

The number of iterations for CSPB to regenerate the original block of 8 bits is just 6. The positions of bits 5 and 6 do not change but will not stay the same in the subsequent rounds when the block size changes. An example of the encryption (and

decryption) process using CSPB for and 8-bit block is given in table 2.4. Any intermediate block among S_1 through S_5 may be selected as the cipher-text.

Table 2.4: Encryption/Decryption process of 8-bit block using CSPB

Bit positions	1	2	3	4	5	6	7	8
S (plain-text)	1	1	0	1	0	1	0	1
S_1	0	1	1	1	0	1	0	1
S_2	0	1	1	1	0	1	1	0
S_3 (cipher-text)	1	1	1	0	0	1	1	0
S_4	1	0	1	0	0	1	1	1
S_5	0	0	0	1	0	1	1	1
S_6 (plain-text)	1	1	0	1	0	1	0	1

The number of iterations to regenerate the original block with these algorithms may be computed in a similar manner for block-sizes of 16 bits and higher. However, it will become more cumbersome as the block-size increases. A simpler method has been adopted for this purpose. A string of 512 bits is taken and encrypted repeatedly with a particular block-size. The string generated after each encryption is compared with the original string. If they are different, the encryption is iterated again, and if they are same, the number of iterations for that block-size is noted. Table 2.5 shows the number of iterations required for different block-sizes to complete a cycle, i.e. to regenerate the original block, for RSPB, LSPB, DSPB, and CSPB.

Table 2.5: Number of iterations for a complete cycle

Block-size	Total no. of iterations required to complete a cycle			
	LSPB	RSPB	DSPB	CSPB
8	12	10	12	6
16	13	15	40	12
32	31	16	180	30
64	450	330	14790	1020
128	117180	32760	15760	17784
256	251	33550	261744	40680
512	9780	35870	374144	68532

It is evident from table 2.5 that if we increase the block size, the number of iterations to complete a cycle also increases, of course with exceptions for LSPB with block-sizes 256 and 512. No direct relation between the block-size and the number of iterations could be established for any of the proposed algorithms.

In actual implementation, chunks of 512 bits are read from the file and encrypted. The process of encryption is started with block-size of 8 bits and a number of iterations are applied. As explained earlier, the number iterations, which forms a part of the key, should be less than the total number of iterations required to regenerate the original block for that particular block-size. The remaining iterations are applied during decryption. The same process is repeated with block-sizes 16, 32, 64, 128, 256, and 512. The order in which the block-sizes are chosen may be altered to enhance the security provided by the algorithm. Choosing a random order of block-sizes will also increase the length of the key, which is discussed in chapter 11.

2.5 Microprocessor-based implementations

Various routines for implementing the proposed algorithms in a microprocessor-based system are listed in this section. The assembly language versions were written and tested for an Intel 8085 system using a simulator as well an SDA-85 kit. The routines of LSPB, RSPB and DSPB are listed here. CSPB, being a combination of these three in a sequence, has not been included.

2.5.1 Routines for 8-bit block-size

For the routines implementing 8-bit round of the proposed algorithms, 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

2.5.1.1 Routines for 8-bit RSPB

Main program:

- Step 1 : Initialize SP with the address of the highest memory location available
- Step 2 : Clear the result area
- Step 3 : Load F900H to HL pair, FA00H to DE pair, initialize some memory location as a counter
- Step 4 : Move the content of M to A
- Step 5 : Push HL and DE pair into the stack
- Step 6 : Call subroutine 'MAP'
- Step 7 : Pop HL and DE pair from the stack
- Step 8 : Move the content of M to A
- Step 9 : AND 80H with A
- Step 10 : ADD B to A
- Step 11 : Store the content of A in the memory location pointed to by DE pair
- Step 12 : Increment HL and DE pair
- Step 13 : Decrement the counter variable in memory
- Step 14 : Repeat from step 4 till the counter is zero
- Step 15 : End

Subroutine MAP:

This subroutine takes the content of A as a parameter. It consults the tables and sets the corresponding bits in the result area.

- Step 1 : Load HL pair with FB00H and DE pair with FD00H
- Step 2 : Clear B and initialize some memory location with 07H as a counter
- Step 3 : Move the content of M to C
- Step 4 : Increment H so that HL pair points location FC00H ($FB+1=FC$)
- Step 5 : Move the content of M to A
- Step 6 : Rotate A once to the right
- Step 7 : If carry flag is set then perform RRC else perform RLC
- Step 8 : Exchange the content of DE pair with the content of HL pair
- Step 9 : Rotate A once either to left or to right depending on the step 7
- Step 10 : Decrement C
- Step 11 : Repeat from step 7 till C is zero
- Step 12 : Add B to A
- Step 13 : Move the content of A to B
- Step 14 : Decrement H so that HL pair points to location FB00H ($FC-1=FB$)
- Step 15 : Increment HL and DE pair
- Step 16 : Decrement the counter variable in memory
- Step 17 : Repeat from step 3 till the counter is zero
- Step 18 : Return

Look-up Tables

Table 1 (FB00H onwards): 04H, 01H, 01H, 02H, 02H, 01H, 03H

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

Table 3 (FD00H onwards): 01H, 02H, 04H, 08H, 10H, 20H, 40H

2.5.1.2 Routine for 8-bit LSPB

The main routine of LSPB is almost the same as that of RSPB except for an additional step (step 11). The same subroutine 'MAP', using the same look-up tables, is called from this routine and hence not listed here to avoid repetition.

- Step 1 : Initialize SP with the address of the highest memory location available
- Step 2 : Clear the result area
- Step 3 : Load F900H to HL pair, FA00H to DE pair, initialize some memory location as a counter
- Step 4 : Move the content of M to A
- Step 5 : Push HL and DE pair into the stack
- Step 6 : Call subroutine 'MAP'
- Step 7 : Pop HL and DE pair from the stack
- Step 8 : Move the content of M to A
- Step 9 : AND 80H with A
- Step 10 : ADD B to A
- Step 11 : Rotate the content of Accumulator 4 times left without carry
- Step 12 : Store the content of A in the memory location pointed to by DE pair
- Step 13 : Increment HL and DE pair
- Step 14 : Decrement the counter variable in memory
- Step 15 : Repeat from step 4 till the counter is zero
- Step 16 : End

2.5.1.3 Routine for 8-bit DSPB

The routine for DSPB also calls the same subroutine 'MAP', using the same look-up tables. There are quite a few modifications made in the main routine.

- Step 1 : Initialize SP with the address of the highest memory location available
- Step 2 : Clear the result area
- Step 3 : Load F900H to HL pair, FA00H to DE pair, initialize some memory location as a counter

- Step 4 : Move the content of M to A
- Step 5 : Push HL and DE pair into the stack
- Step 6 : Call subroutine 'MAP'
- Step 7 : Pop HL and DE pair from the stack
- Step 8 : Move the content of M to A
- Step 9 : AND 80H with A
- Step 10 : ADD B to A
- Step 11 : AND 0FH with A
- Step 12 : Rotate A two times without carry to the right
- Step 13 : Move the content of A to C
- Step 14 : Move the content of B to A
- Step 15 : AND 03H with A
- Step 16 : ADD C to A
- Step 17 : Move the content of A to C
- Step 18 : Move the content of B to A
- Step 19 : AND 0CH with A
- Step 20 : Rotate A four times without carry to the left
- Step 21 : ADD C to A
- Step 22 : Store the content of A in the memory location pointed to by DE pair
- Step 23 : Increment HL and DE pair
- Step 24 : Decrement the counter variable in memory
- Step 25 : Repeat from step 4 till the counter is zero
- Step 26 : End

2.5.2 Routines for 16-bit (and higher) block-sizes

The routines of the proposed algorithms 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 for 16-bit blocks are listed in this section. 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. As in the case of 8-bit block-size, only the routines

for RSPB, LSPB and DSPB are listed here. The fourth one, i.e. CSPB, is just a sequential combination of the other three, and hence, not included.

2.5.2.1 Routines for 16-bit (and higher) RSPB

Main program:

- Step 1 : Clear the result area
- Step 2 : Initialize register D with the number of iterations
(i.e. 512bits/16bits for 16-bit block-size)
- Step 3 : Initialize SP with the address of the highest memory location available
- Step 4 : Load BC pair with F900H
- Step 5 : Push HL pair, BC pair, DE pair and PSW into the stack
- Step 6 : Call subroutine 'OPPR'
- Step 7 : Load HL pair with FC00H and B with 02H
- Step 8 : Move the content of M to A
- Step 9 : ADD B to A
- Step 10 : Move the content of A to M
- Step 11 : Pop HL pair, BC pair, DE pair and PSW from stack
- Step 12 : Rotate the content of A two times without carry to the right
- Step 13 : Move the content of D to A
- Step 14 : Increment BC pair by the content of A
- Step 15 : Decrement D
- Step 16 : Repeat from step 3 till D is zero
- Step 17 : End

Subroutine OPPr:

This subroutine checks each and every individual bit of the input data and, depending on the bit-value, calls another subroutine 'BTST' for conversion from input data stream to output data stream.

- Step 1 : Clear L and initialize D with 02H (to handle 16-bit block-size)
- Step 2 : Load E with 08H and H with 01H
- Step 3 : Load A with the content of memory location pointed to by BC pair
- Step 4 : AND A with H
- Step 5 : If A is not equal 0 then call subroutine 'BTST'
- Step 6 : Increase the value of L register
- Step 7 : Move the content of H to A
- Step 8 : Rotate A without carry to the right
- Step 9 : Move the content of 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 3 till D is zero
- Step 15 : Return

Subroutine BTST:

This subroutine generates the result by consulting the look-up table and setting the corresponding bit in the result area.

- Step 1 : Push HL register pair, BC register pair, DE pair and PSW into stack
- Step 2 : Load A with the content of the memory location FC00H
- Step 3 : Move the content of A to B
- Step 4 : Load H with FBH
- Step 5 : Move the content of M to A
- Step 6 : Subtract 08H from A
- Step 7 : If carry=1 then 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 FAH
- Step 13 : Move the content of B to L
- Step 14 : Move the content of C to A
- Step 15 : Compare A with 00H
- Step 16 : If zero flag is set then jump to step 22
- Step 17 : Load A with 01H
- Step 18 : Rotate A without carry to the right
- Step 19 : Decrement C
- Step 20 : Repeat from step 19 till C is zero
- Step 21 : Repeat from step 24
- Step 22 : Load A with 01H
- Step 23 : OR A with M
- Step 24 : Move the content of A to M
- Step 25 : Pop HL pair, BC pair, DE pair and PSW from stack
- Step 26 : Return

Look-up Tables

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

Table 2 (FC00H onwards): 00H

2.5.2.2 Routine for 16-bit (and higher) LSPB

The routine for 16-bit (and higher) LSPB is same as that of RSPB and needs no modification. The only requirement for this purpose is to change the look-up tables.

Look-up Tables

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

Table 2 (FC00H onwards): 00H

2.5.2.3 Routine for 16-bit (and higher) DSPB

The routine for 16-bit (and higher) DSPB is also same as that of RSPB and LSPB. As in the case of LSPB, the only requirement for this purpose is to change the look-up tables.

Look-up Tables

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

Table 2 (FC00H onwards): 00H

2.6 Results and comparisons

The evaluation of the LSPB, RSPB, DSPB and CSPB has been done using the methods already discussed in section 1.8.3. Each algorithm has been tested in its weakest form, i.e., having just one pass (no iterations) in each round, so that the strength of the algorithm will increase during actual implementation. The results of the proposed algorithms have been compared with those of Triple DES.

For the purpose of testing, four categories of files, namely .dll, .exe, .jpg and .txt, have been considered and five different files of varying sizes have been included in each category and put through the encryption algorithms to be tested.

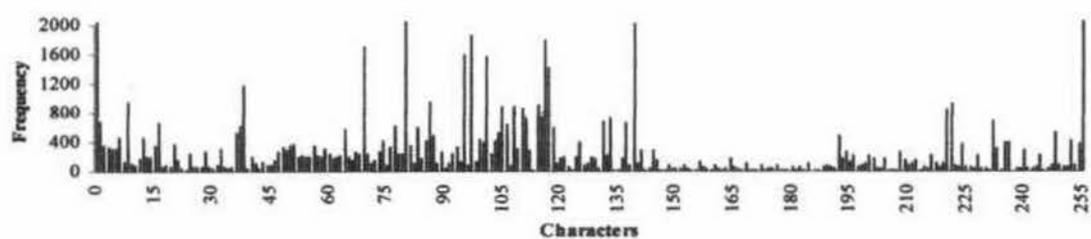
2.6.1 Character frequency

Although all the twenty files have been used to test the algorithms, the results of just one file in each category are shown here for the sake of brevity. Figure 2.6 shows the frequencies of all the 256 characters in the .dll source file and the ones obtained after encryption with LSPB, RSPB, DSPB, CSPB and Triple DES. Similarly, figure 2.7, figure 2.8, and figure 2.9 illustrate the comparative character-frequencies for .exe, .jpg, and .txt files, respectively.

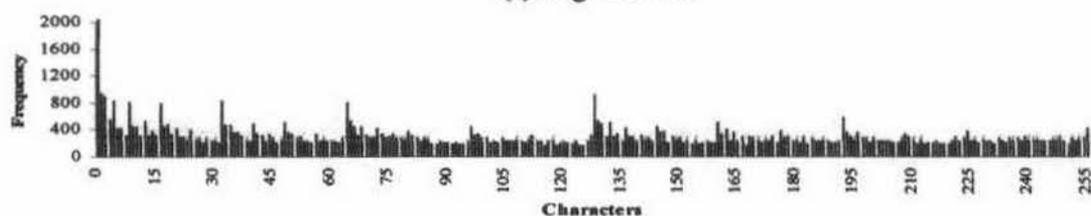
In all the graphs depicted in the figures, some very high values of character-frequency have been truncated to make the low values visible enough. If this is not done, then the low values will be visible as almost zero values.

In the original .dll file, the characters are clustered in some regions and almost negligible in some portions. Few characters have very high frequencies, which have been truncated in the graph. The same file, when encrypted with the all the proposed algorithms have more or less similar frequencies, which means that the characters are distributed evenly throughout the character space. The only exception is the character with ASCII value 0, which has quite a high frequency in all. These results have been compared to that of Triple DES. 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.

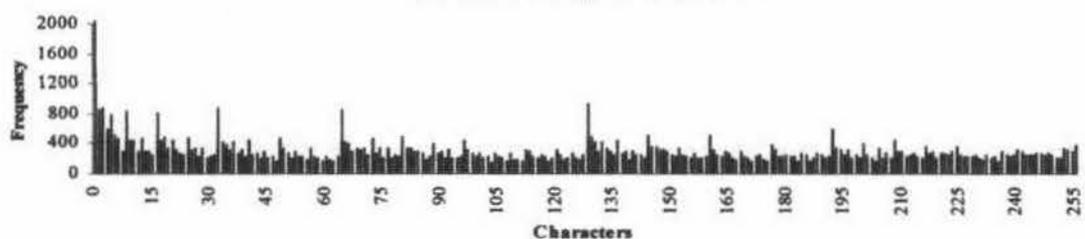
The results for .exe file are almost similar to that of the .dll file. The same explanations as in the case of .dll file hold true for .exe file also. It is clear from the frequency graphs that the performances of the proposed algorithms are quite good in case of the .jpg file and are comparable with that of Triple DES. In the case of .txt source file, the frequencies of almost half of the total characters are nil. The characters are fairly distributed over the character space after the files are encrypted. All the four proposed algorithms show similar results, which are quite comparable to that of Triple DES.



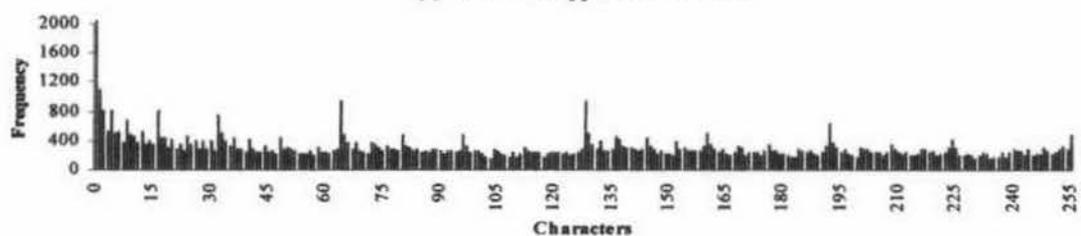
(a) Original .dll file



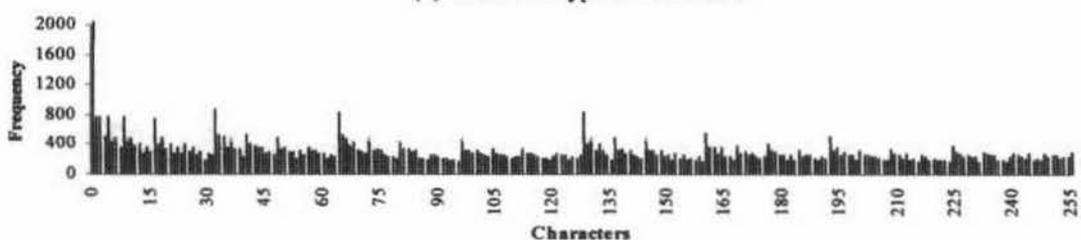
(b) .dll file encrypted with LSPB



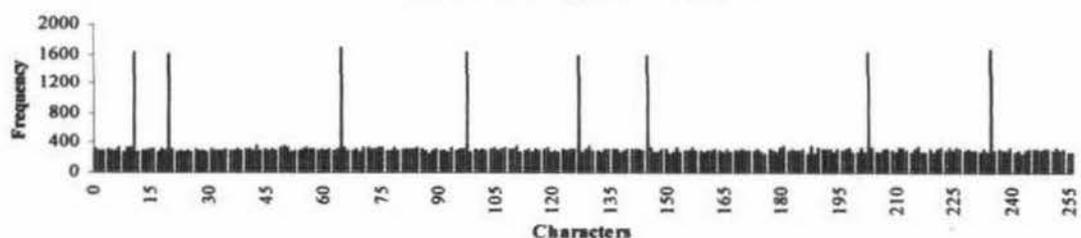
(c) .dll file encrypted with RSPB



(d) .dll file encrypted with DSPB

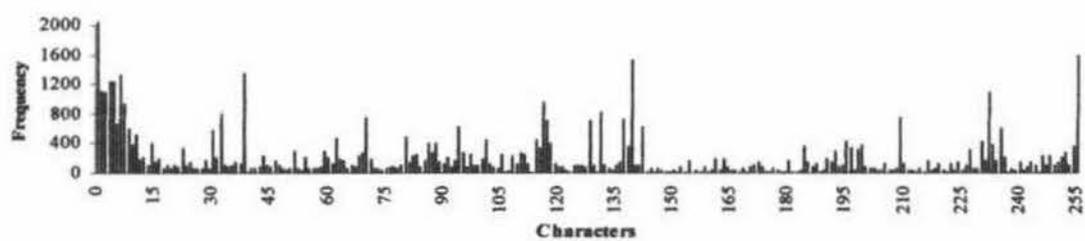


(e) .dll file encrypted with CSPB

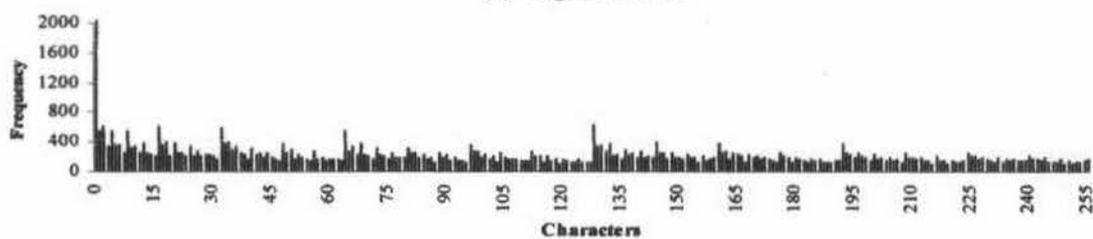


(f) .dll file encrypted with Triple DES

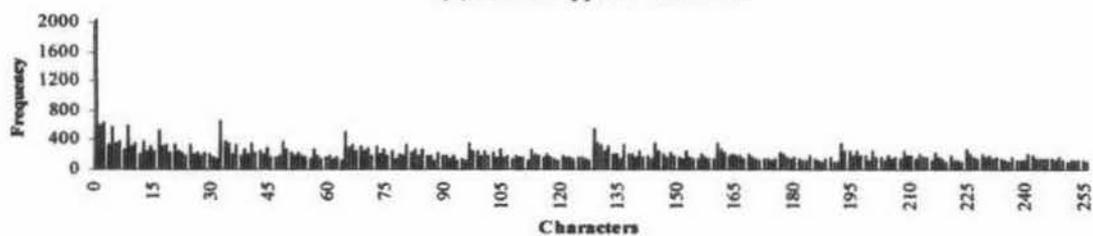
Figure 2.6: Comparison of character-frequencies in the source and encrypted .dll files



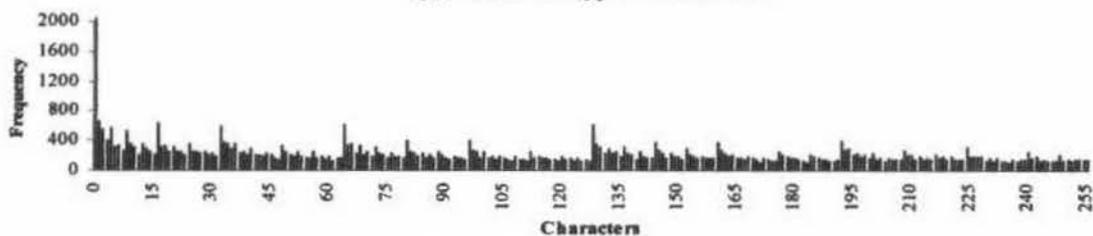
(a) Original .exe file



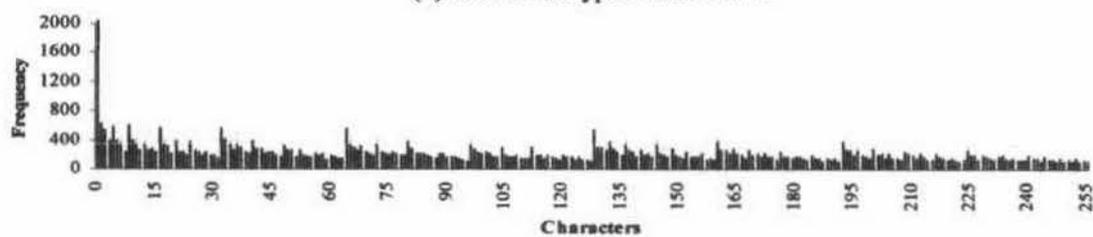
(b) .exe encrypted with LSPB



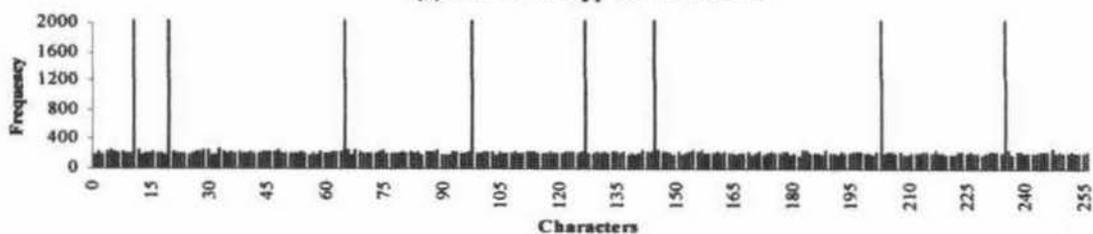
(c) .exe file encrypted with RSPB



(d) .exe file encrypted with DSPB



(e) .exe file encrypted with CSPB



(f) .exe file encrypted with Triple DES

Figure 2.7: Comparison of character-frequencies in the source and encrypted .exe files

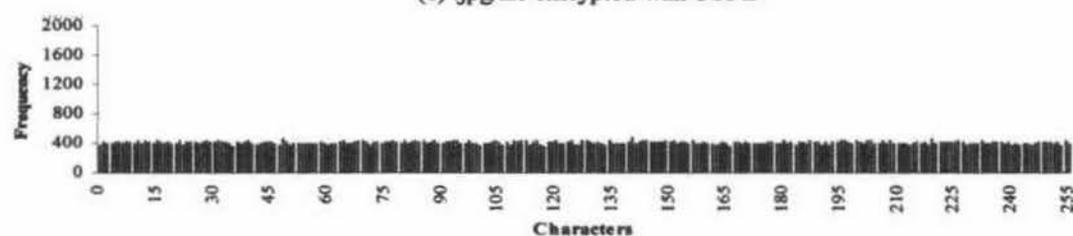
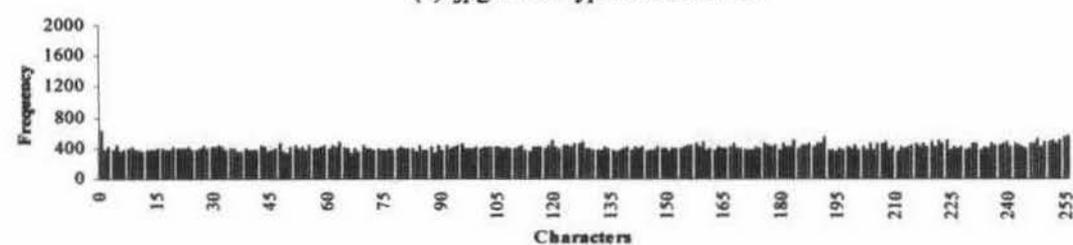
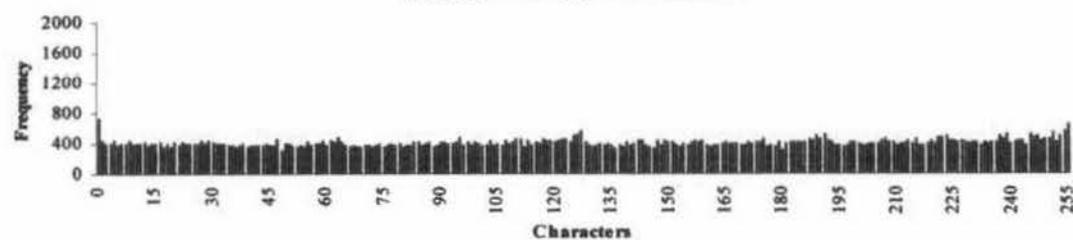
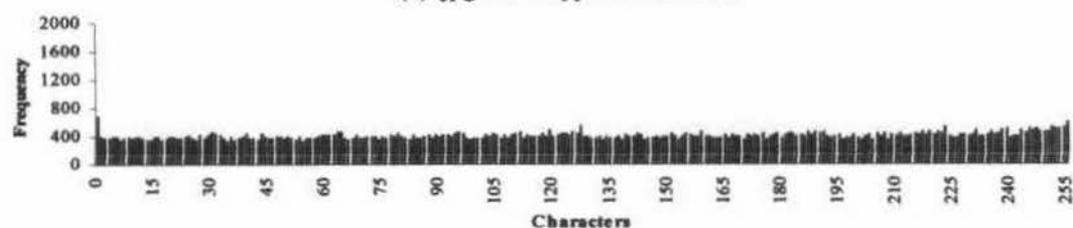
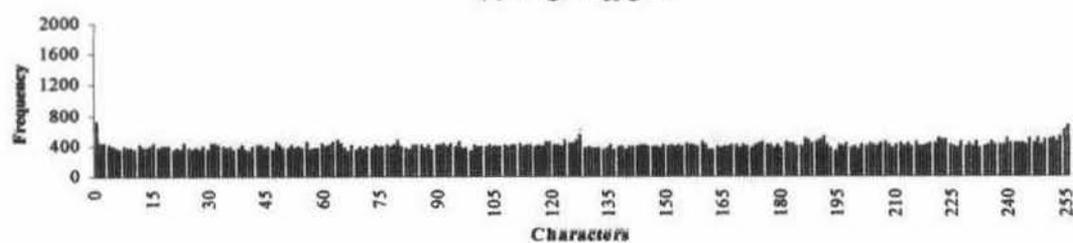
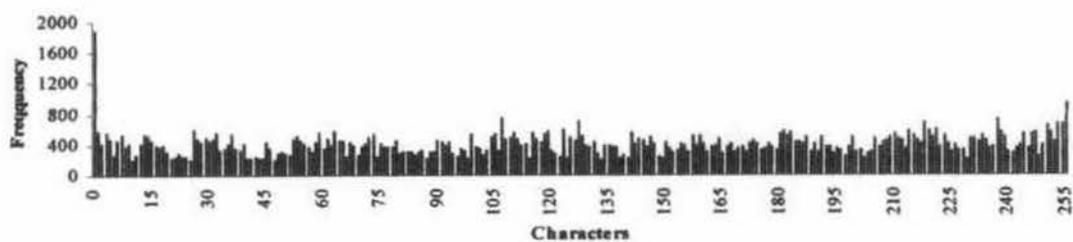


Figure 2.8: Comparison of character-frequencies in the source and encrypted .jpg files

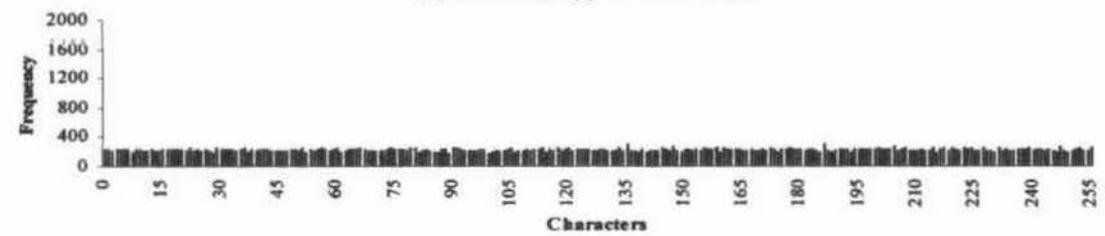
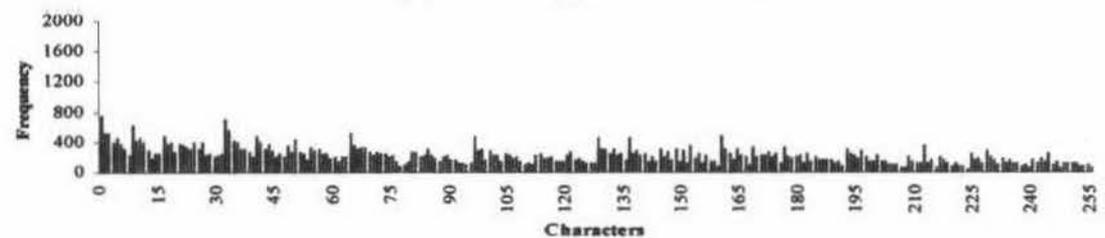
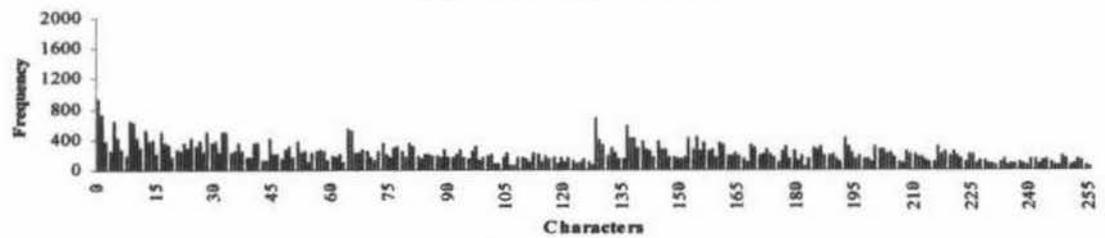
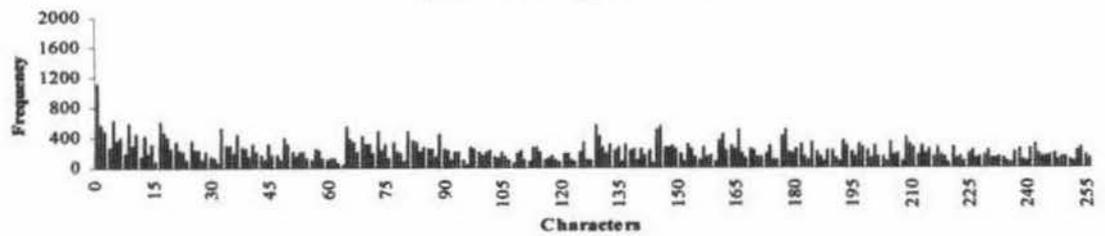
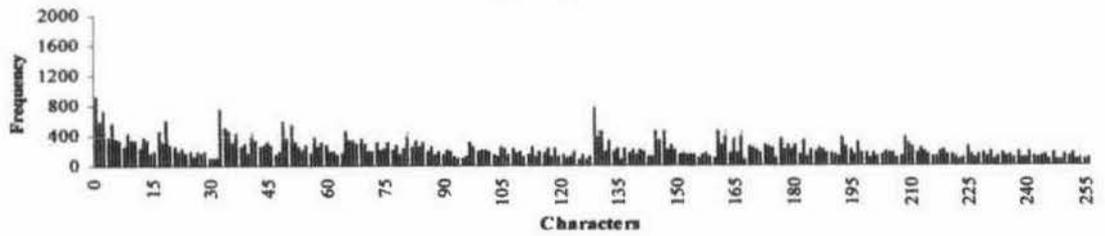
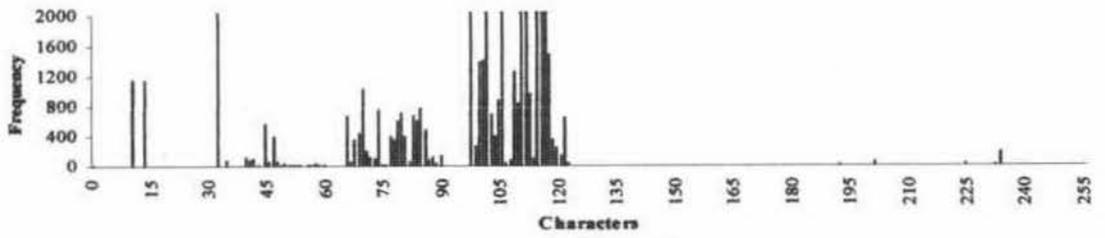


Figure 2.9: Comparison of character-frequencies in the source and encrypted .txt files

2.6.2 Chi-Square test and encryption time

The encrypted forms of all the twenty files are compared to their original ones to check whether and to what extent they differ with each other. The character-frequencies in each pair of original and encrypted files and χ^2 values were computed. The results of the χ^2 -test and encryption times were compared with those of Triple DES. Each category of files has been dealt with separately. The comparative χ^2 values and degrees of freedom (DF) for the proposed algorithms along with those for Triple DES are listed in table 2.6 and the comparative encryption times are listed in table 2.7. The comparisons in table 2.6 can be visualised in figure 2.10.

Table 2.6: χ^2 -test with .dll files

Sl. No.	Original file	File size (bytes)	LSPB		RSPB		DSPB		CSPB		Triple DES	
			χ^2	DF	χ^2	DF	χ^2	DF	χ^2	DF	χ^2	DF
1	1.dll	20480	4833	207	4816	205	4651	199	5136	203	29790	255
2	2.dll	53312	16449	255	16408	255	15799	255	16421	255	43835	255
3	3.dll	90176	40879	255	40957	255	40379	255	41168	255	66128	255
4	4.dll	118784	111298	255	111767	255	109164	255	116385	255	1211289	255
5	5.dll	204800	1014978	255	1047282	255	936791	255	1358258	255	2416524	255

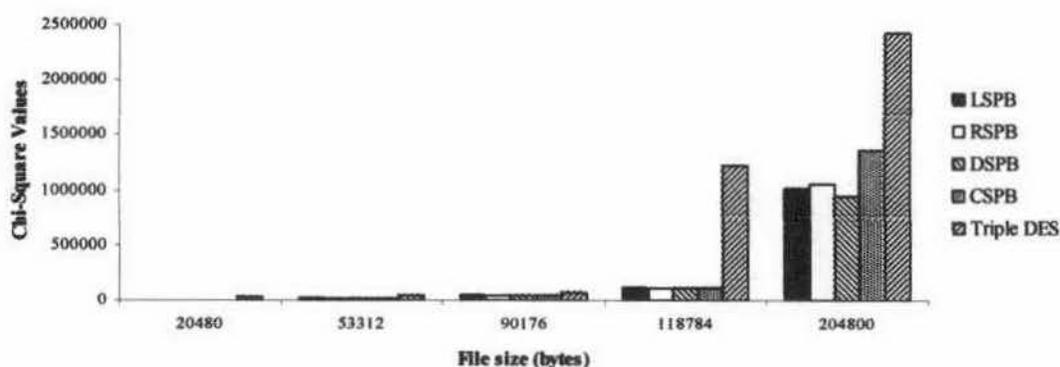


Figure 2.10: Proposed algorithms vs. Triple DES in χ^2 -test of .dll files

Table 2.7: Encryption time with .dll files

Sl. No.	Original file	File size (bytes)	Encryption time (secs)				
			LSPB	RSPB	DSPB	CSPB	Triple DES
1	1.dll	20480	0.054945	0.054945	0.054945	0.109890	06
2	2.dll	53312	0.109890	0.109890	0.164835	0.219780	16
3	3.dll	90176	0.219780	0.219780	0.274725	0.384615	26
4	4.dll	118784	0.329670	0.274725	0.439560	0.604396	34
5	5.dll	204800	0.549451	0.549451	0.714286	0.934066	69

The performances of the proposed algorithms in χ^2 -test with .dll are fair enough to be compared with that of Triple DES. The values for CSPB are fairly high compared to others. The degree of freedom (DF) in almost all the cases was 255 barring very few exceptions. Since Triple DES is quite complicated, it takes a long time to encrypt a file compared to any of the proposed algorithms. Very small encryption time and large χ^2 values indicate the strength of these algorithms.

Similar test for .exe files also show better results than in the case of .dll files. Table 2.8 and figure 2.10 illustrate the nature of χ^2 -test results for .exe files and table 2.9 compares the encryption times of the proposed algorithms with those of Triple DES.

Table 2.8: χ^2 -test with .exe files

Sl. No.	Original file	File size (bytes)	LSPB		RSPB		DSPB		CSPB		Triple DES	
			χ^2	DF	χ^2	DF	χ^2	DF	χ^2	DF	χ^2	DF
1	1.exe	23104	7009	255	6905	255	6763	255	7004	255	8772	255
2	2.exe	52736	17212	255	17387	255	16802	255	17018	255	43426	255
3	3.exe	131136	929886	255	930209	255	927133	255	937597	255	986693	255
4	4.exe	170496	154446	255	155177	255	147562	255	168235	255	475893	255
5	5.exe	200832	2274398	255	2280662	255	2196194	255	2342468	255	1847377	255

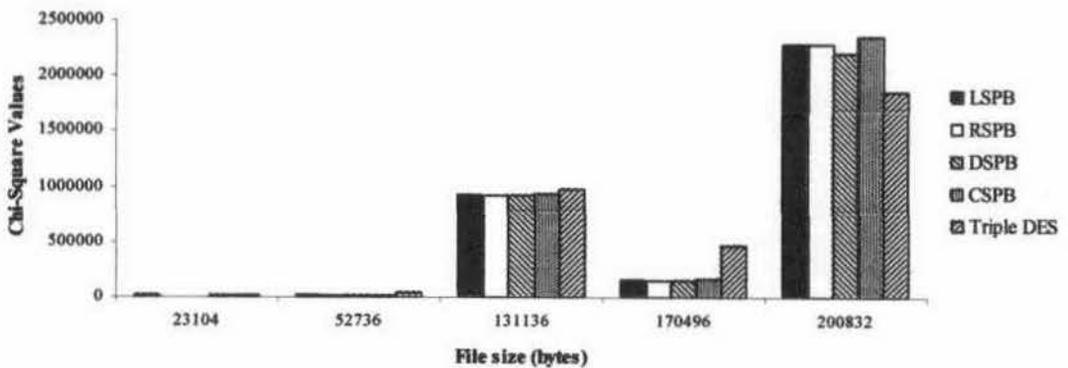


Figure 2.11: Proposed algorithms vs. Triple DES in χ^2 -test of .exe files

Table 2.9: Encryption time with .exe files

Sl. No.	Original file	File size (bytes)	Encryption time (secs)				
			LSPB	RSPB	DSPB	CSPB	Triple DES
1	1.exe	23104	0.054945	0.054945	0.054945	0.109890	12
2	2.exe	52736	0.109890	0.054945	0.164835	0.274725	15
3	3.exe	131136	0.329670	0.329670	0.439560	0.604396	29
4	4.exe	170496	0.439560	0.439560	0.604396	0.824176	49
5	5.exe	200832	0.549451	0.549451	0.659341	0.934066	58

It is evident from the tables and figures that the results for .exe files are better than those for .dll files and, for some files, even better than Triple DES. Moreover, the encryption times are much less than that of Triple DES. The test results in case of .jpg files are listed in table 2.10 and table 2.11 and also shown graphically in figure 2.12.

Table 2.10: χ^2 -test with .jpg files

Sl. No.	Original file	File size (bytes)	LSPB		RSPB		DSPB		CSPB		Triple DES	
			χ^2	DF	χ^2	DF	χ^2	DF	χ^2	DF	χ^2	DF
1	1.jpg	28544	3820	255	3842	255	3801	255	3996	255	4331	255
2	2.jpg	71232	2521	255	2450	255	2466	255	2421	255	2916	255
3	3.jpg	105600	4016	255	4006	255	4004	255	4191	255	5227	255
4	4.jpg	160704	8707	255	8510	255	8845	255	8423	255	22314	255
5	5.jpg	216576	11394	255	11132	255	11512	255	11325	255	29824	255

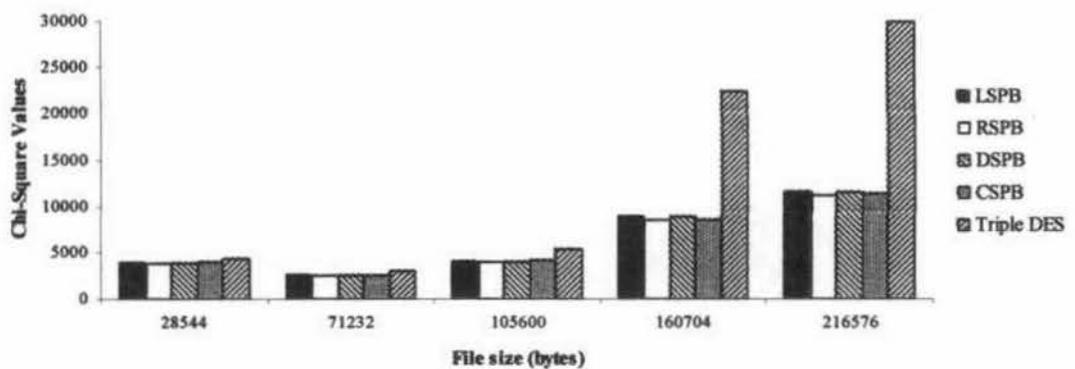
Figure 2.12: Proposed algorithms vs. Triple DES in χ^2 -test of .jpg files

Table 2.11: Encryption time with .jpg files

Sl. No.	Original file	File size (bytes)	Encryption time (secs)				
			LSPB	RSPB	DSPB	CSPB	Triple DES
1	1.jpg	28544	0.054945	0.054945	0.109890	0.164835	08
2	2.jpg	71232	0.219780	0.109890	0.274725	0.329670	21
3	3.jpg	105600	0.274725	0.329670	0.329670	0.494505	31
4	4.jpg	160704	0.439560	0.439560	0.604396	0.714286	47
5	5.jpg	216576	0.604396	0.604396	0.769231	0.989011	63

Some weaknesses are shown by the proposed algorithms in comparison to Triple DES in the case of .jpg files. The difference in χ^2 values is quite high in case of large files. Nevertheless, considering the encryption time together with the χ^2 values, the algorithms are still comparable with Triple DES, though they are not that good.

Finally, the test for .txt files also show better results like in the case of .exe files. Table 2.12 and figure 2.13 illustrate the nature of χ^2 -test results for .txt files and table 2.13 compares the encryption times of the proposed algorithms with those of Triple DES.

Table 2.12: χ^2 -test with .txt files

Sl. No.	Original file	File size (bytes)	LSPB		RSPB		DSPB		CSPB		Triple DES	
			χ^2	DF	χ^2	DF	χ^2	DF	χ^2	DF	χ^2	DF
1	t1.txt	6976	8014	85	8062	87	8049	90	7927	85	10629	183
2	t2.txt	23808	32154	255	33473	255	33345	255	32433	255	32638	255
3	t3.txt	58688	80779	255	83007	255	83061	255	81405	255	82101	255
4	t4.txt	118784	165273	255	170273	255	170692	255	166288	255	170557	255
5	t5.txt	190784	407160	255	427406	255	427033	255	409902	255	430338	255

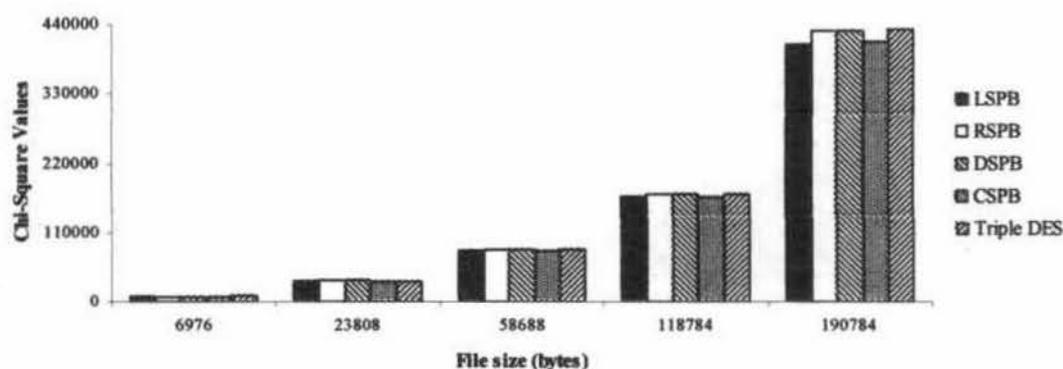
Figure 2.13: Proposed algorithms vs. Triple DES in χ^2 -test of .txt files

Table 2.13: Encryption time with .txt files

Sl. No.	Original file	File size (bytes)	Encryption time (secs)				
			LSPB	RSPB	DSPB	CSPB	Triple DES
1	t1.txt	6976	0.054945	0.054945	0.054945	0.054945	02
2	t2.txt	23808	0.109890	0.109890	0.109890	0.109890	07
3	t3.txt	58688	0.164835	0.164835	0.219780	0.274725	17
4	t4.txt	118784	0.329670	0.329670	0.439560	0.604396	35
5	t5.txt	190784	0.494505	0.494505	0.659341	0.934066	55

The χ^2 values obtained for .txt files are very large values which prove the heterogeneity between the source and encrypted files. The results are almost similar to those of Triple DES. The results are still better when the encryption times are considered together with the χ^2 values.

2.6.3 Avalanche and runs

To examine the effect of the plain-text bits on the cipher-text, i.e., to check whether a small change in the plain-text creates an avalanche in the cipher-text or not, a 32-bit binary string was repeatedly encrypted using the proposed algorithms, 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 noted. Tables 2.14 through 2.17 show the results of this test for LSPB, RSPB, DSPB, and CSPB, respectively.

Table 2.14: Avalanche and runs in LSPB

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	6D9A9E74	19	19	0
1 ST	C145D450	6D9A9E74	18	19	1
2 ND	0145D450	6D9A9E74	17	19	2
3 RD	6145D450	6D9A9E71	19	19	0
4 TH	5145D450	6D9A9E74	21	19	2
5 TH	4945D450	6D9A9E71	21	19	2
6 TH	4545D450	6D9A9E74	21	19	2
7 TH	4345D450	6D9A9E74	19	19	0
8 TH	4045D450	6D9A9E74	17	19	2
9 TH	41C5D450	6D9A9E74	17	19	2
10 TH	4105D450	6D9A9E74	17	19	2
11 TH	4165D450	6D9A9E74	19	19	0
12 TH	4155D450	6D9A9E74	21	19	2
13 TH	414DD450	6D9A9E74	19	19	0
14 TH	4141D450	6D9A9E74	17	19	2
15 TH	4147D450	6D9A9E74	17	19	2
16 TH	4144D450	4D9A9E74	19	19	0
17 TH	41455450	6D9A9E74	21	19	2
18 TH	41459450	6D9A1E74	19	17	2
19 TH	4145F450	6D9A9E74	17	19	2
20 TH	4145C450	6D9A9E74	17	19	2
21 ST	4145DE50	6D9A9E74	17	19	2
22 ND	4145D050	6D9A9A74	17	21	5
23 RD	4145D650	6D9A9E74	19	19	0
24 TH	4145D550	6D9A9E74	21	19	2
25 TH	4145D4D0	6D9A9E74	19	19	0
26 TH	4145D410	6D9A9E74	17	19	2
27 TH	4145D470	6D9A9E74	17	19	2
28 TH	4145D440	2D9A9E74	17	19	2
29 TH	4145D458	6D9A9E74	19	19	0
30 TH	4145D454	ED9A9E74	21	19	2
31 ST	4145D452	6D9A9E74	21	19	2
32 ND	4145D451	6D9A9E74	20	19	1

It is evident from the table that LSPB shows some amount of weakness in causing a sufficient amount of diffusion although there is a difference in runs between the plain-text and the cipher-text in most of the cases. The weakness is due absence of substitutions in the algorithm which involves only permutations. Nevertheless, due to its simplicity and other advantages, including the feasibility of intended target for implementation, it may be a good algorithm that can be used in a cascaded manner with a substitution cipher. This type of weakness is shown by almost all transposition ciphers since the number of 1's and 0's do not change.

Table 2.15: Avalanche and runs in RSPB

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	6B1092DC	19	19	0
1 ST	C145D450	EB1092DC	18	18	0
2 ND	0145D450	6B1092DC	17	19	2
3 RD	6145D450	6B10B2BC	19	19	0
4 TH	5145D450	6B1092DC	21	19	2
5 TH	4945D450	6B1092DC	21	19	2
6 TH	4545D450	6B1292DC	21	21	0
7 TH	4345D450	6B1092DC	19	19	0
8 TH	4045D450	6B1092DC	17	19	2
9 TH	41C5D450	6B1092DC	17	19	2
10 TH	4105D450	6B1092DC	17	19	2
11 TH	4165D450	6B1092DC	19	19	0
12 TH	4155D450	6B1092DC	21	19	2
13 TH	414DD450	6B1492BC	19	21	2
14 TH	4141D450	6B1092DC	17	19	2
15 TH	4147D450	6B1092DC	17	19	2
16 TH	4144D450	6B1092DC	19	19	0
17 TH	41455450	6B1092DC	21	19	2
18 TH	41459450	6B1092DC	19	19	0
19 TH	4145F450	6B1092DC	17	19	2
20 TH	4145C450	6B1092DC	17	19	2
21 ST	4145DE50	6B1092DC	17	19	2
22 ND	4145D050	6B1092DC	17	19	5
23 RD	4145D650	6B1092DC	19	19	0
24 TH	4145D550	6B1192DC	21	19	2
25 TH	4145D4D0	6B1092DC	19	19	0
26 TH	4145D410	6B1092DC	17	19	2
27 TH	4145D470	6B1092DC	17	19	2
28 TH	4145D440	6B1092DC	17	19	2
29 TH	4145D458	6B1092DC	19	19	0
30 TH	4145D454	6B1092DC	21	19	2
31 ST	4145D452	6B1092DC	21	19	2
32 ND	4145D451	6B1092DC	20	19	1

RSPB shows a similar type of results and the same explanations as in the case of LSPB hold true for RSPB as well. The difference in runs is also similar to LSPB.

Table 2.16: Avalanche and runs in DSPB

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	1160DE57	19	16	3
1 ST	C145D450	1160BE57	18	16	2
2 ND	0145D450	0160DE57	17	14	3
3 RD	6145D450	1168DE57	19	18	1
4 TH	5145D450	1160DE57	21	16	5
5 TH	4945D450	1140DE57	21	16	5
6 TH	4545D450	1170DE57	21	16	5
7 TH	4345D450	5160DE57	19	18	1
8 TH	4045D450	1160DE57	17	16	1
9 TH	41C5D450	1160DE57	17	16	1
10 TH	4105D450	1160DE57	17	16	1
11 TH	4165D450	1160DE57	19	16	3
12 TH	4155D450	9160DE57	21	17	4
13 TH	414DD450	1164DE57	19	18	1
14 TH	4141D450	1160DE57	17	16	1
15 TH	4147D450	1160D757	17	16	1
16 TH	4144D450	1160D757	19	16	3
17 TH	41455450	1140D757	21	16	5
18 TH	41459450	1120DE57	19	16	3
19 TH	4145F450	1160D757	17	16	1
20 TH	4145C450	1160D757	17	16	1
21 ST	4145DE50	1960D757	17	16	1
22 ND	4145D050	1160DE57	17	16	1
23 RD	4145D650	1160DE57	19	16	3
24 TH	4145D550	1160DE57	21	16	5
25 TH	4145D4D0	1160DE57	19	16	3
26 TH	4145D410	1160DE57	17	16	1
27 TH	4145D470	1160DE57	17	16	1
28 TH	4145D440	1160DE57	17	16	1
29 TH	4145D458	1162DE57	19	18	1
30 TH	4145D454	1160DE57	21	16	5
31 ST	4145D452	1160FE57	21	14	7
32 ND	4145D451	1160DE57	20	16	4

DSPB shows a better result compared to rest of its siblings, of course with some exceptions. The amount of diffusion caused by encryption and the difference in runs is much more in case of DSPB.

CSPB seems better than LSPB and RSPB, but not as much as DSPB. The amount of diffusion is very little, almost same as LSPB and RSPB. On the other hand, the average difference in runs is even better than DSPB.

Table 2.17: Avalanche and runs in CSPB

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	78C6391A	19	15	4
1 ST	C145D450	78C6391A	18	15	3
2 ND	0145D450	78C6391A	17	15	2
3 RD	6145D450	78C6391A	19	15	4
4 TH	5145D450	78C6391A	21	15	6
5 TH	4945D450	78C6391A	21	15	6
6 TH	4545D450	78C6391A	21	15	6
7 TH	4345D450	78C6391A	19	15	4
8 TH	4045D450	78C6391A	17	15	2
9 TH	41C5D450	78C6391A	17	15	2
10 TH	4105D450	78C6391A	17	15	2
11 TH	4165D450	78C6391A	19	15	4
12 TH	4155D450	78C6391A	21	15	6
13 TH	414DD450	78C6391A	19	15	4
14 TH	4141D450	78C6391A	17	15	2
15 TH	4147D450	78C6391A	17	15	2
16 TH	4144D450	78C6391A	19	15	4
17 TH	41455450	68C6391A	21	17	4
18 TH	41459450	78C6391A	19	15	4
19 TH	4145F450	78C6391A	17	15	2
20 TH	4145C450	78C6391A	17	15	2
21 ST	4145DE50	78C6391A	17	15	2
22 ND	4145D050	78C6391A	17	15	2
23 RD	4145D650	78C6391E	19	13	6
24 TH	4145D550	78C6393A	21	15	6
25 TH	4145D4D0	78C6391A	19	15	4
26 TH	4145D410	78C6391A	17	15	2
27 TH	4145D470	F8C6391A	17	14	3
28 TH	4145D440	78C6391A	17	15	2
29 TH	4145D458	78C6391A	19	15	4
30 TH	4145D454	78C6391A	21	15	6
31 ST	4145D452	78C6391A	21	15	6
32 ND	4145D451	78C6391A	20	15	5

2.7 Conclusion

In this chapter, four different microprocessor-based implementations of the technique of prime position orientations have been proposed. Various tests have also been done to analyse the strength of these algorithms. The strength of these algorithms may be enhanced by dividing the 512-bit block into blocks of unequal sizes and applying the requisite rounds. As already discussed in the preceding section, these transposition ciphers may be used in a cascaded manner with other substitution ciphers.

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 Block Exchange Technique (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, $S = S_1S_2S_3\dots S_m$, 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 $S = \text{abcdefghijklmnopqrstuvwxy}\phi\lambda\pi\theta\xi\psi$ 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	qrstuvwxy	$\phi\lambda\pi\theta\xi\psi$

Output:

B_1	B_2	B_3	B_4
abefcdgh	ijmknlop	qruvstwx	$yz\pi\theta\phi\lambda\xi\psi$

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

Input:

B_1	B_2
abefcdghijmknlop	qruvstwxvzπθφλξψ

Output:

B_1	B_2
abefijmncdghklop	qruvvyzπθstwxφλξψ

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

Input:

B_1
abefcdghijmknlopqruvstwxvzπθφλξψ

Output:

B_1
abefcdghqruvstwxijmknlopyzπθφλξψ

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. $S' = abefcdghqruvstwxijmknlopyzπθφλξψ$.

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

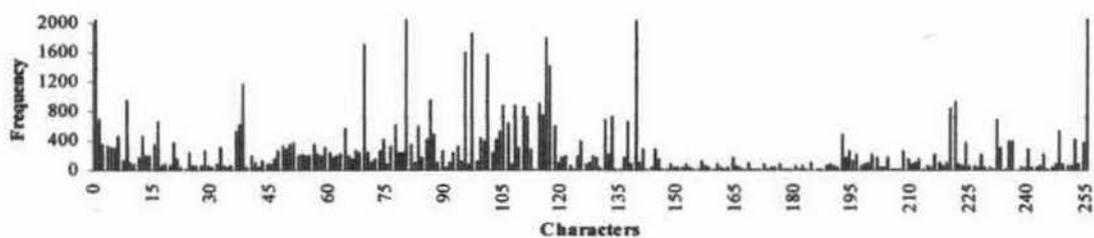
The routines for higher block-sizes, like 32, 64, 128 etc., are almost same with a slight modification in each case, and hence, not listed here.

3.5 Results and comparisons

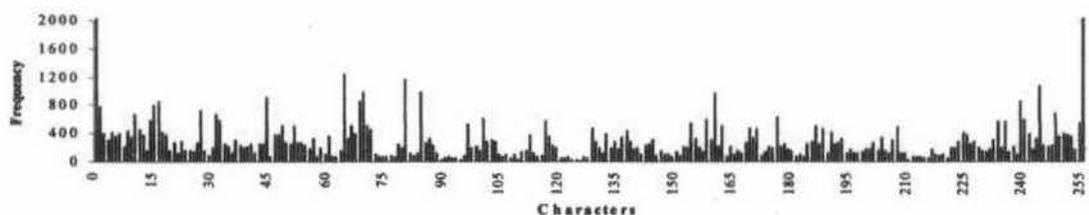
The methods already discussed in section 1.8.3 were used to test the strength of BET. Just one pass (no iterations) in each round was used to test it in its weakest form, so that its strength may increase during actual implementation. Triple DES has been used as a benchmark to compare the results of 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

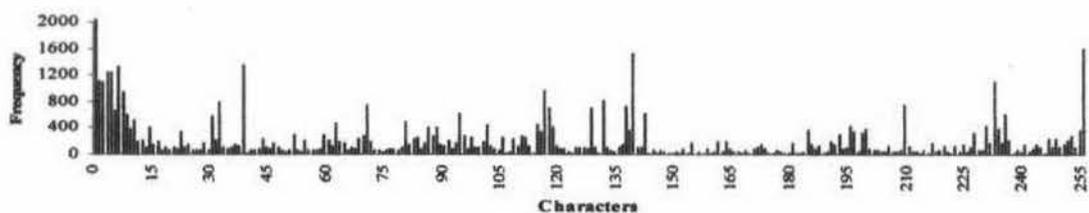


(b) .dll file encrypted with BET

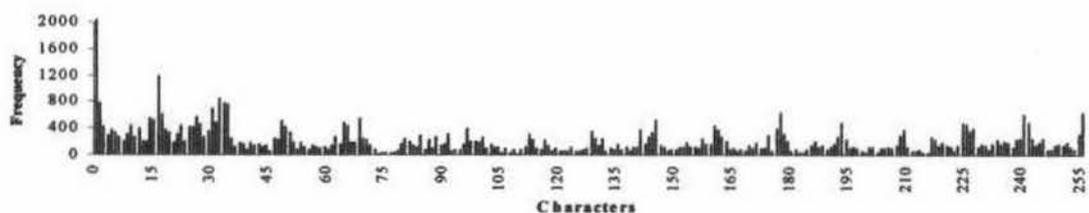


(c) .dll file encrypted with Triple DES

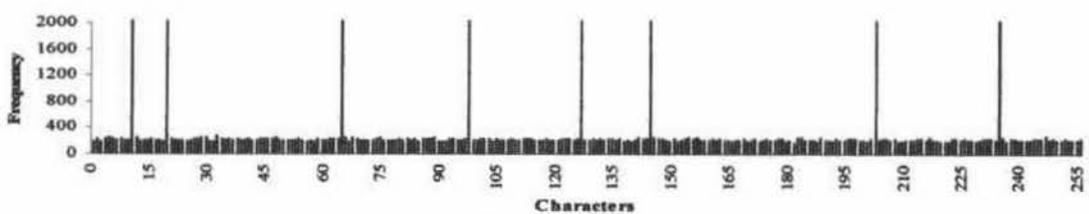
Figure 3.1: Character-frequencies in the source and encrypted .dll files



(a) Original .exe file



(b) .exe file encrypted with BET



(c) .exe file encrypted with Triple DES

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

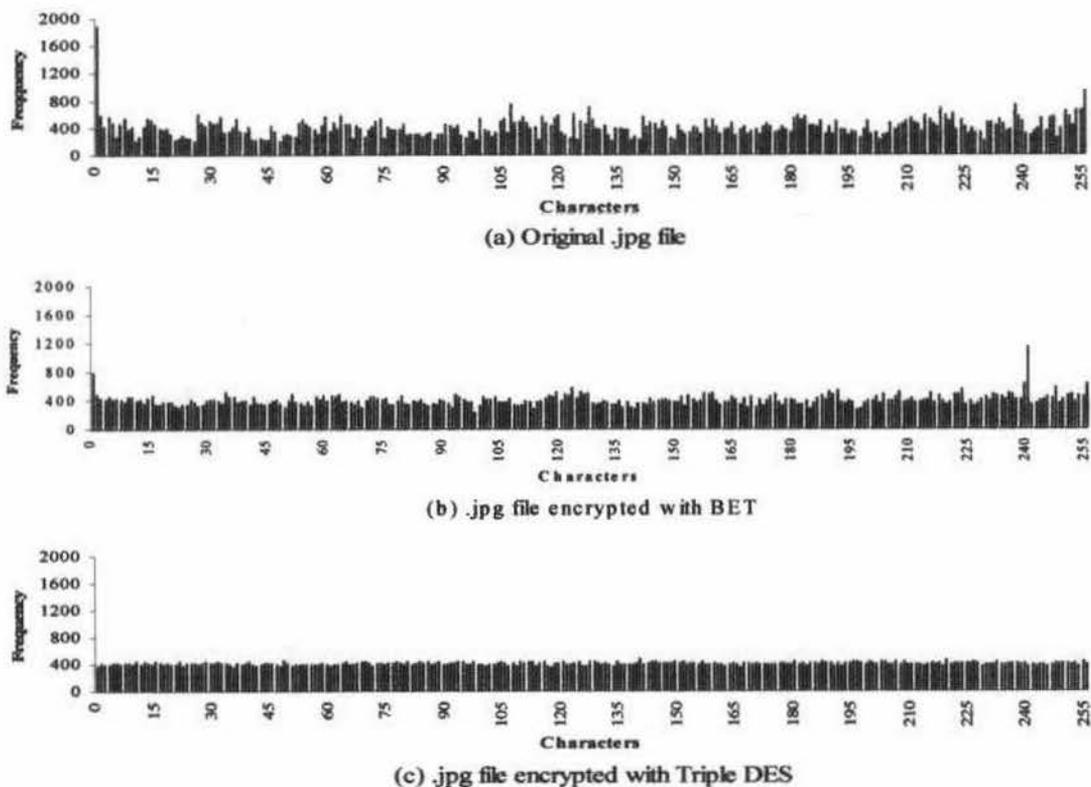


Figure 3.3: Character-frequencies in the source and encrypted .jpg files

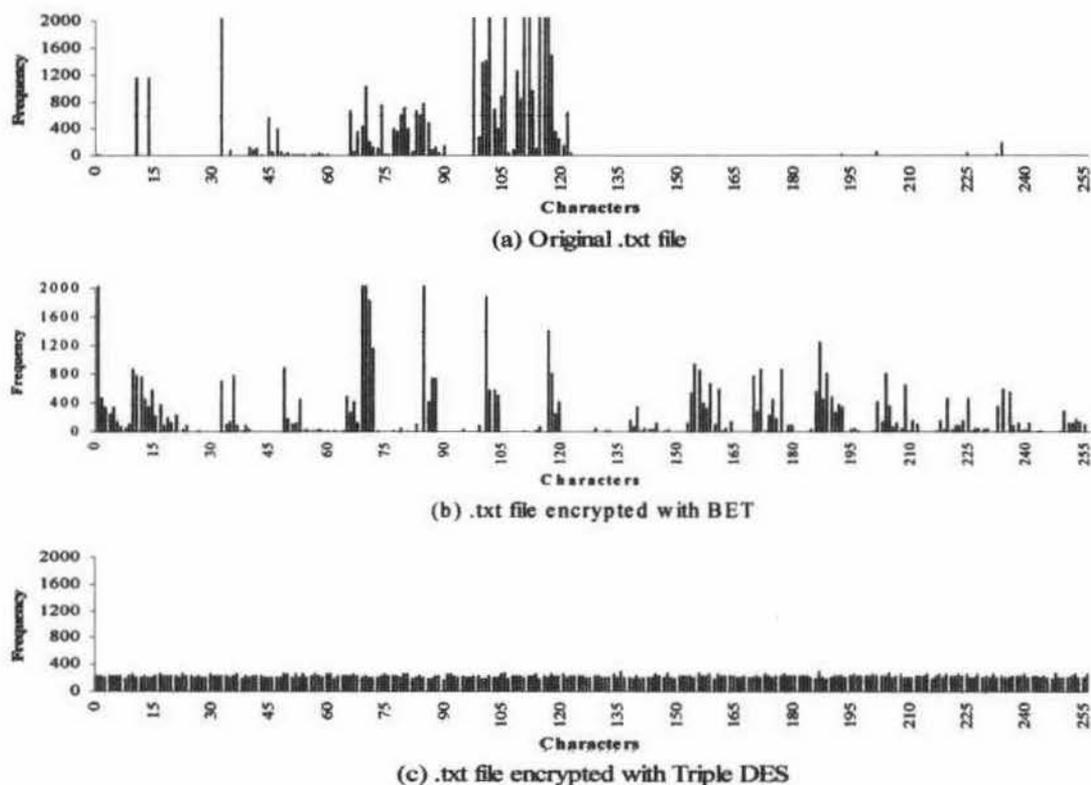


Figure 3.4: Character-frequencies in the source and encrypted .txt files

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.

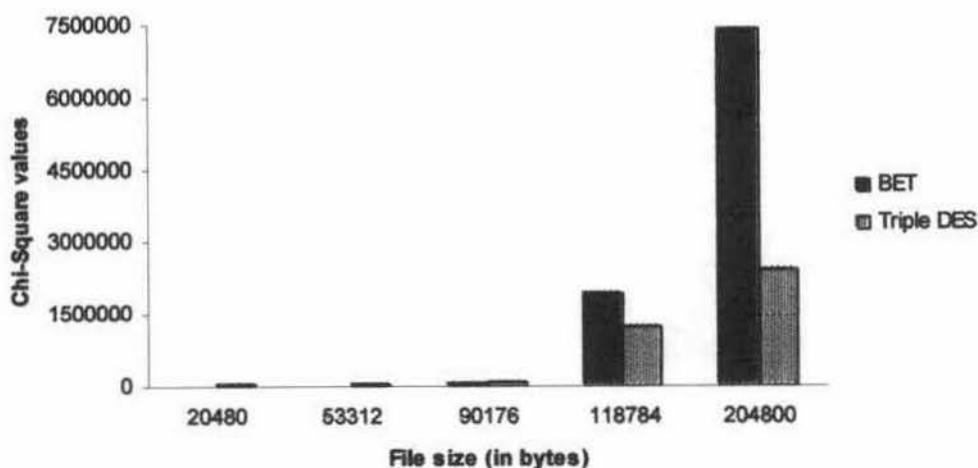
Since the non-printable characters are absent in the original .txt file, the frequencies of almost half of the total 256 characters are nil. The characters are fairly distributed over the character space after the files are encrypted by BET and the same is comparable with Triple DES.

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

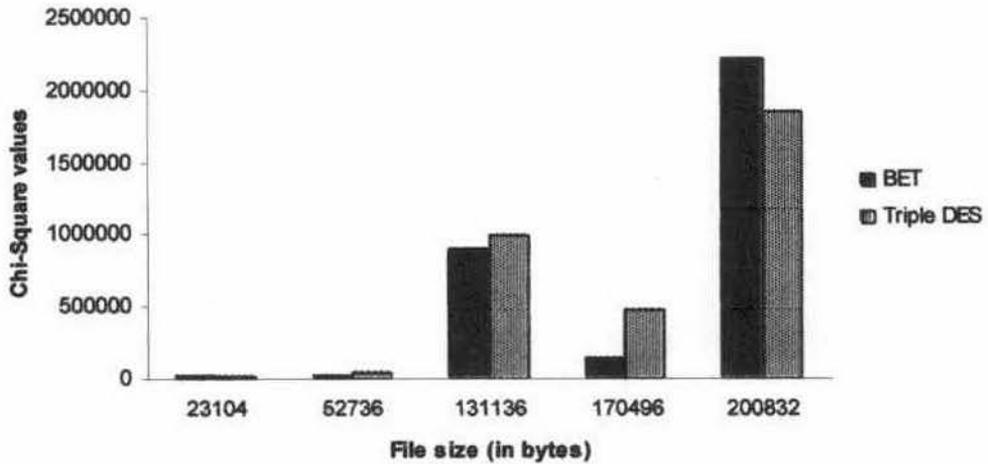


Figure 3.6: BET vs. Triple DES in χ^2 -test of .exe files

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 χ^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

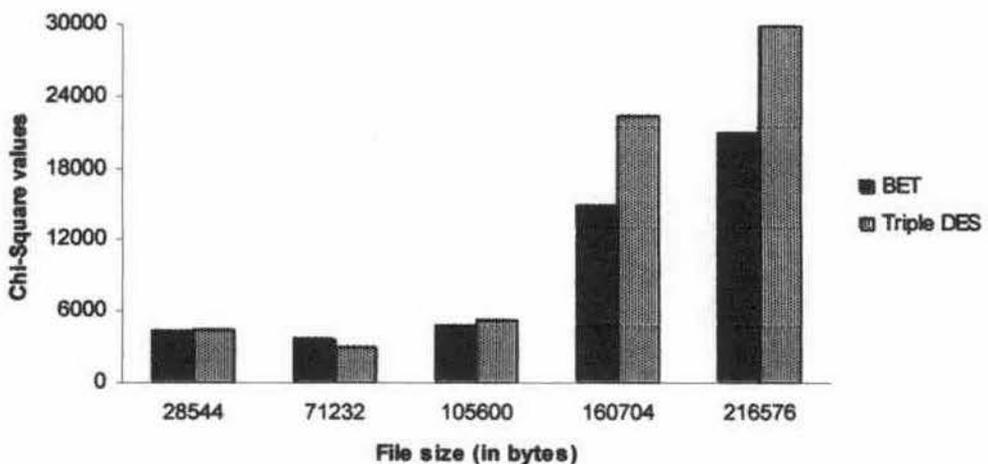


Figure 3.7: BET vs. Triple DES in χ^2 -test of .jpg files

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

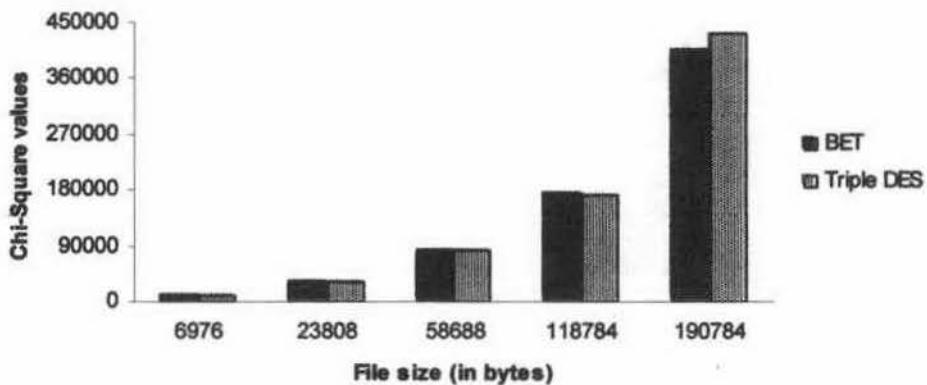


Figure 3.8: BET vs. Triple DES in χ^2 -test of .txt files

The nature shown by BET for .txt files is very much similar to that shown by Triple DES. The χ^2 values for BET are almost equal to those for Triple DES, although the degrees of freedom (DF) are bit less. High χ^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
1 ST	C145D450	D445C150	18	17	1
2 ND	0145D450	1445C150	17	16	1
3 RD	6145D450	5445C150	19	20	1
4 TH	5145D450	5445C151	21	19	2
5 TH	4945D450	5445C158	21	18	3
6 TH	4545D450	5445C154	21	20	1
7 TH	4345D450	7445C150	19	16	3
8 TH	4045D450	4445C150	17	16	1
9 TH	41C5D450	5445C1D0	17	16	1
10 TH	4105D450	5445C110	17	16	1
11 TH	4165D450	5645C150	19	18	1
12 TH	4155D450	5545C150	21	20	1
13 TH	414DD450	5C45C150	19	16	3
14 TH	4141D450	5045C150	17	16	1
15 TH	4147D450	5445C170	17	16	1
16 TH	4144D450	5445C140	19	16	3
17 TH	41455450	54454150	21	20	1
18 TH	41459450	54458150	19	18	1
19 TH	4145F450	5447C150	17	16	1
20 TH	4145C450	5444C150	17	18	1
21 ST	4145DE50	544DC150	17	18	1
22 ND	4145D050	5441C150	17	16	1
23 RD	4145D650	5445E150	19	18	1
24 TH	4145D550	5445D150	21	20	1
25 TH	4145D4D0	54C5C150	19	18	1
26 TH	4145D410	5405C150	17	16	1
27 TH	4145D470	5445C350	17	18	1
28 TH	4145D440	5445C050	17	16	1
29 TH	4145D458	5445C950	19	20	1
30 TH	4145D454	5495C550	21	20	1
31 ST	4145D452	5465C150	21	21	0
32 ND	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.

Selective Positional Orientation of Bits (SPOB)

4.1 Introduction

A novel but simple microprocessor-based transposition cipher, namely **Selective Positional Orientation of Bits (SPOB)**, has been proposed in this chapter. The input, as a string of binary bits, is divided into a number of blocks each containing n bits, where n is one of 8, 16, 32, 64, 128, 256, and 512. The bit positions are then altered using the proposed technique, where swapping is performed between pairs of selected bits within a block. If the whole process is performed repeatedly for a particular block size, the original block is regenerated after a finite number of iterations. One of these iterations is selected to generate the encrypted string and hence the corresponding encryption key. The same operation is performed for decryption.

4.2 The SPOB scheme

In SPOB, the source string of 512 bits is divided into a number of blocks, each containing a finite number of bits n ($1 < n \leq 512$). For each block $S = S_0S_1S_2\dots S_{n-1}$, a repetitive swap operation is performed between S_i and S_{i+k} where $i = 1, 2, \dots, n-2$ starting from MSB where $k = 2, 3, 4, 5, \dots, n-1$. Section 4.2.1 deals with the operations in detail.

4.2.1 The rounds in SPOB

Round 1: In *Round 1*, the 512-bit input string is broken into 64 blocks of 8 bits each. Within each block, the following operations are performed:

Step 1: Starting from the first bit (MSB), a pair of bits is selected so that there is exactly one bit between those two bits. The selected bits are swapped. This is repeated

till the last pair. So the swapping is done on the bit-pairs (1,3), (2,4), (3,5), (4,6), (5,7), and (6,8) within each block.

Step 2: In this step, similar operations are performed on the block of bits obtained from *Step 1*, where there are exactly two bits between the selected bits. So the swapping is done on the pairs (1,4), (2,5), (3,6), (4,7), and (5,8).

Step 3: In *Step 3*, there will be exactly three bits between the selected bits. So the swapping is performed on the bit-pairs (1,5), (2,6), (3,7), and (4,8).

Continuing in this fashion, one can go up to *Step 6* for 8-bit blocks, each time incrementing the number of bits between the selected pair of bits. Actually, there will be $n-2$ steps for an n -bit block wherein the $(n-2)^{\text{th}}$ step, only the first and the last bits (MSB and LSB) are swapped.

Round 2: After completing all the steps for 8-bit blocks, the block-size is doubled (i.e. block-size = 16) in *Round 2* and the whole process as in *Round 1* is repeated.

This way the entire process of encryption will complete in *Round 7*, where block size is 512.

If a round is performed repeatedly, the original block (input to the particular round) is regenerated after a finite number of iterations. An intermediate iteration in a particular round is chosen for the input to the next round. So the whole process is completed after iterating each round several times, less than the number of iterations required to complete a cycle in that particular round. The remaining iterations in each round are performed during decryption.

4.3 Example and discussion

The operations on an 8-bit block is shown here as an example. Instead of 0's and 1's, the bits are denoted by letters A, B, C etc. so that the movement of bits are clear to the readers. Considering a block of 8 bits, $X = ABCDEFGH$, only one round

will have to be performed. The operations in each step of the round are illustrated here.

Step 1: Swap bits 1 and 3, 2 and 4, and so on. X then changes to $X_1 = CDEFGHAB$. Roughly estimating, the bit A should be in place of C , but it has moved to the place of G . The reason is that the bit of a position is altered during each swap. Bit A moves to bit 3 after the first swap, but when bit 3 is again involved in a swap with bit 5, it moves further to bit 5. Eventually bit A moves to bit 7 and so is the fate with other bits also.

Step 2: Swap bits 1 and 4, 2 and 5, and so on. X_1 then changes to $X_2 = FGHABECD$.

Step 3: Swap bits 1 and 5, 2 and 6, and so on. X_2 then changes to $X_3 = BECDFGHA$.

Step 4: Swap bits 1 and 6, 2 and 7, and so on. X_3 then changes to $X_4 = GHADFBEC$.

Step 5: Swap bits 1 and 7, 2 and 8. X_4 then changes to $X_5 = ECADFBGH$.

Step 6: Swap bits 1 and 8. X_5 then changes to $X_6 = HCADFBGE$.

So, finally $X=ABCDEFGH$ changes to $X_6 = HCADFBGE$. All these steps taken together constitute the only round for an 8-bit block. Figure 4.1 shows the alterations of the bit positions after this round.

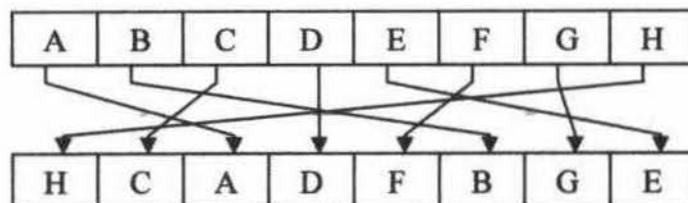


Figure 4.1: Transposition of bits in *Round 1* of SPOB

If this round is performed repeatedly, the original block will be regenerated after six iterations as shown in table 4.1.

Table 4.1: Regeneration of the original block

Original Block	A B C D E F G H
Iteration 1	H C A D F B G E
Iteration 2	E A H D B C G F
Iteration 3	F H E D C A G B
Iteration 4	B E F D A H G C
Iteration 5	C F B D H E G A
Iteration 6	A B C D E F G H

Any of the intermediate iterations may be taken to form the encrypted block. The block size and the number of iterations together will form the encryption key that will be discussed in chapter 11. It may be noticed that the position of the bit D does not change. That will not affect the encryption process since it will change in the very next round when the block-size becomes 16. Table 4.2 shows the number of iterations required to regenerate the original block for each block-size.

Table 4.2: Number of iterations for a complete cycle

Block size	8	16	32	64	128	256	512
Maximum iterations required for a cycle	6	16	24	3354	13260	15466	23532

4.4 Microprocessor-based implementation

Instead of going through complex computations and exchanges, SPOB has been implemented in an Intel 8085 microprocessor-based system using the mappings among the source and encrypted blocks. The routines for SPOB will be exactly the same as those for BET discussed in chapter 3 barring the look-up tables. Hence the routines for SPOB are not listed here to avoid mere repetitions. The respective look-up tables used for 8-bit and 16-bit (and higher) SPOB are given in sections 4.4.1 and 4.4.2, respectively.

4.4.1 Look-up tables for 8-bit SPOB

Table 1 (FB00H onwards): 01H, 01H, 03H, 01H, 04H, 02H

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

Table 3 (FD00H onwards): 01H, 04H, 08H, 20H, 40H, 80H

4.4.2 Look-up tables for 16-bit (and higher) SPOB

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

Table 2 (FC00H onwards): 00H

The tables for higher block-sizes, like 32, 64, 128 etc., are similar with a slight modification in each case, and hence, not given here.

4.5 Results and comparisons

The same set of twenty files, and more exactly, 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 SPOB algorithm. The results of the tests have been compared with those of Triple DES.

Like all the previous algorithms, the strength and weaknesses of SPOB 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.

4.5.1 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 for the sake of brevity. The variation of frequencies of all the 256 ASCII characters in the .dll, .exe, .jpg and .txt source files and the ones obtained as results of encryption with SPOB and Triple DES are shown in figures 4.2 through 4.5, respectively.

As usual, very high frequencies of few characters in some graphs have been truncated to make the low values quite visible, otherwise they will look like almost zero values.

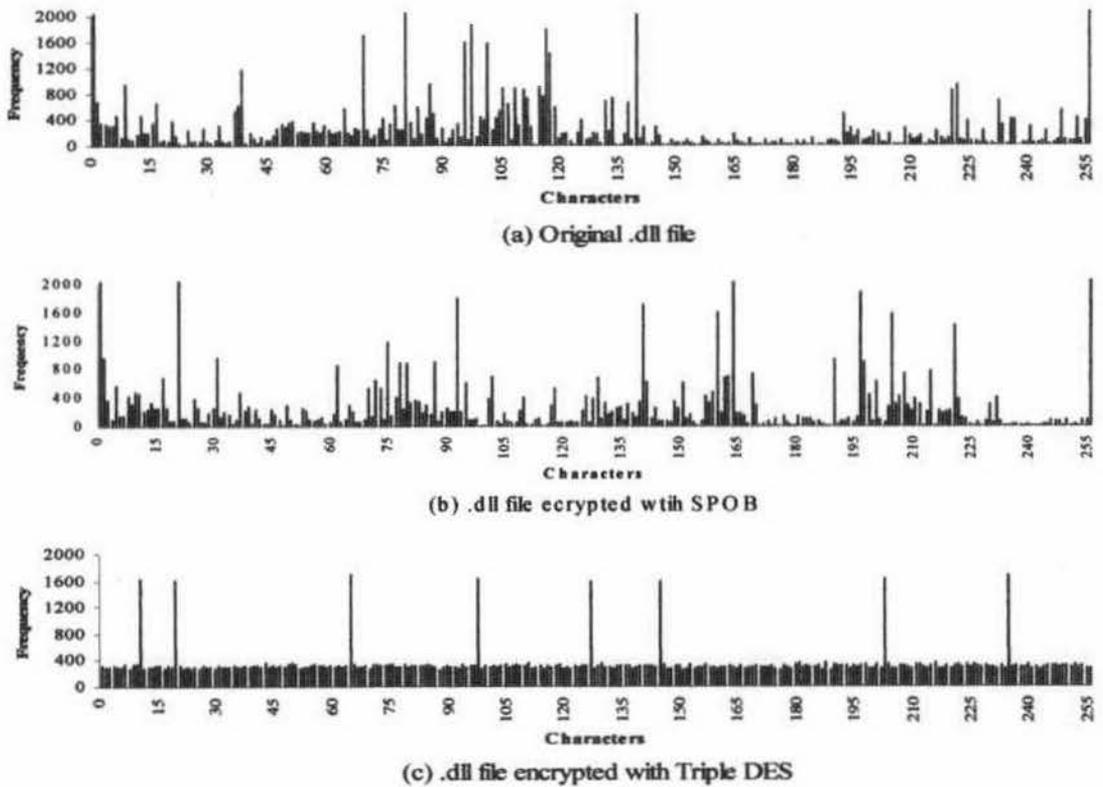


Figure 4.2: Character-frequencies in the source and encrypted .dll files

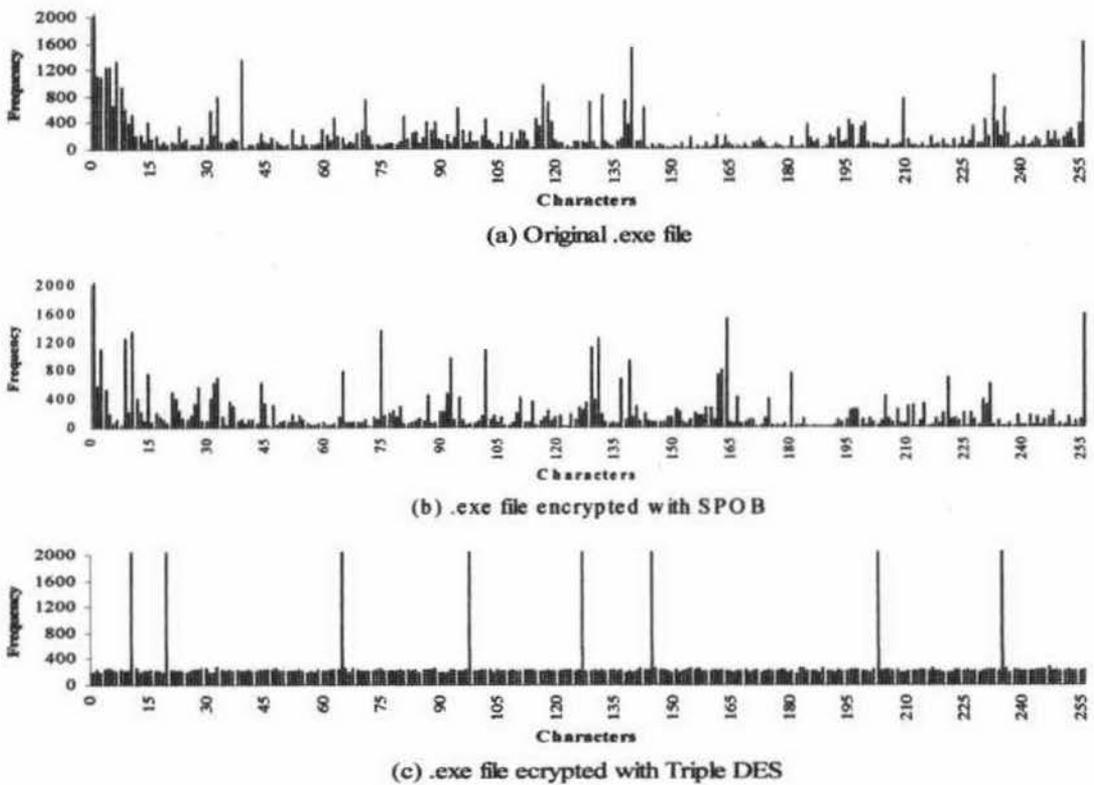


Figure 4.3: Character-frequencies in the source and encrypted .exe files

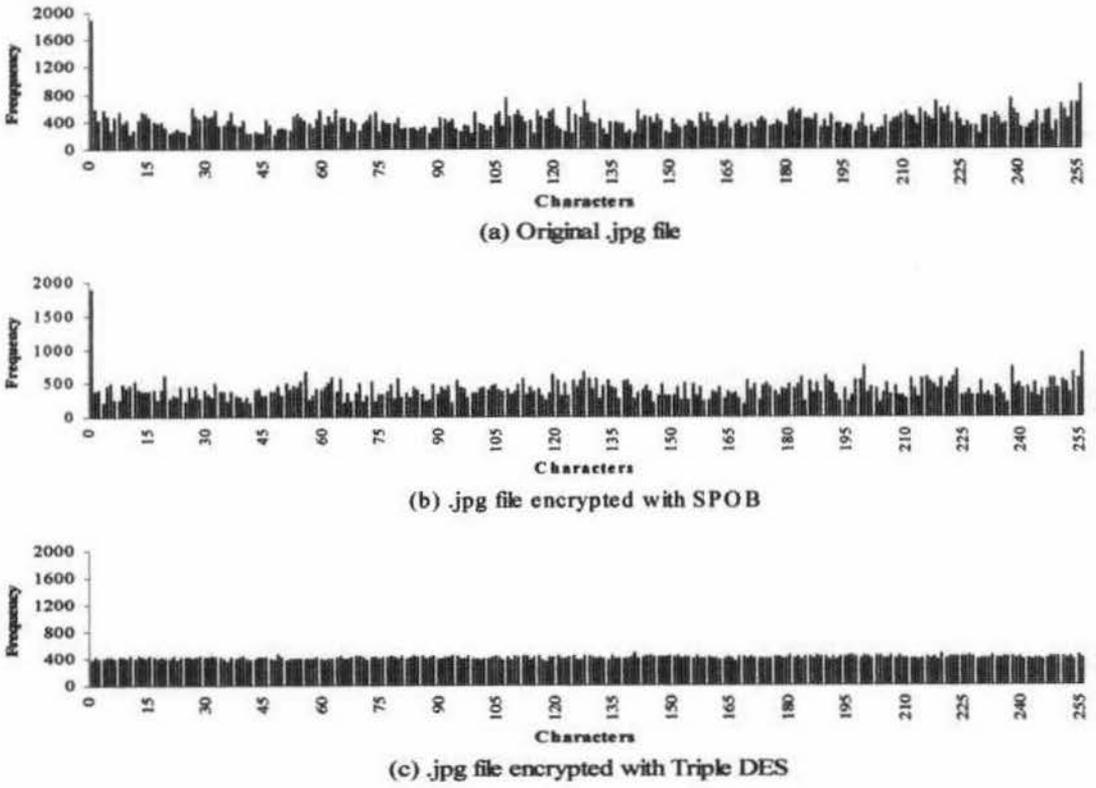


Figure 4.4: Character-frequencies in the source and encrypted .jpg files

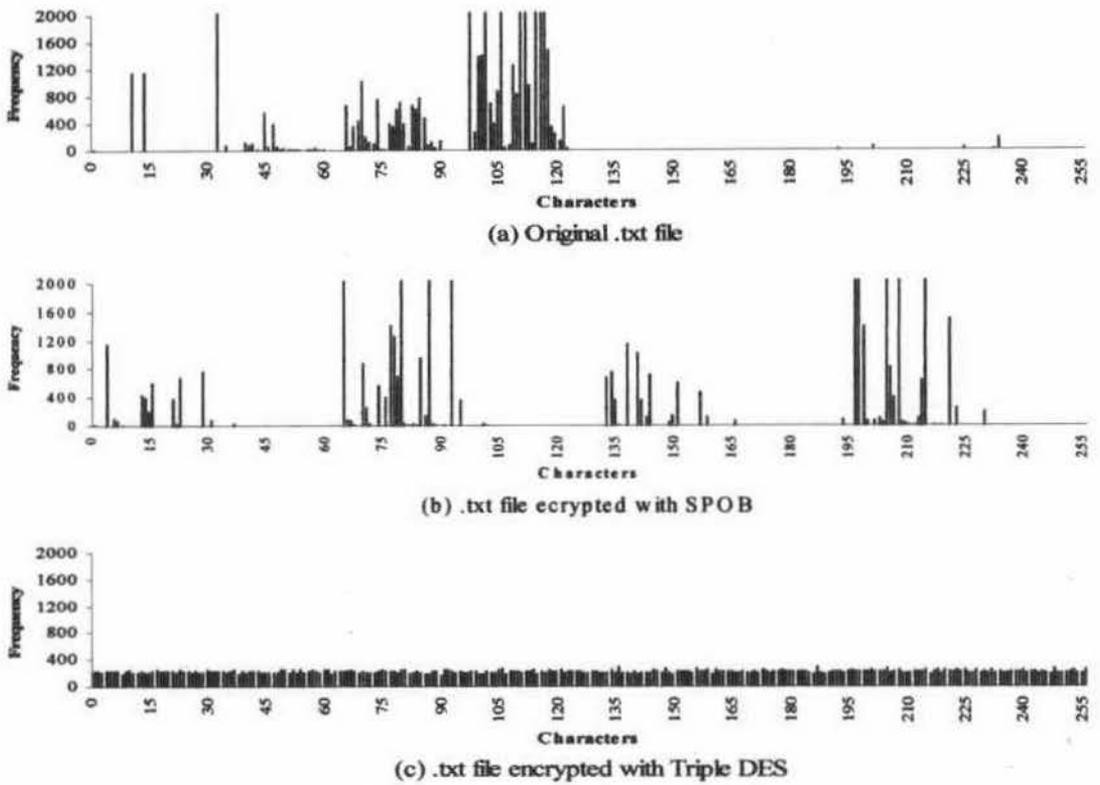


Figure 4.5: Character-frequencies in the source and encrypted .txt files

In case of .dll file, the result shown by SPOB is not as good as that of Triple DES. It has broken the clusters of characters in the original .dll file, but it should have been better. Nevertheless, the frequencies in the original file have been changed by considerable amounts, as it is evident from the χ^2 -test in section 4.5.2.

The result for the .exe file is almost the same as for the .dll file and needs no further explanations. The performances shown by both SPOB and Triple DES in case of .exe file are almost same as in .dll file.

SPOB looks much better in case of .jpg file in comparison to .dll and .exe files. All the characters are fairly distributed over the 0-255 region and the graph is quite comparable to that of Triple DES.

SPOB has not shown a good performance for .txt file compared to other categories of files. Since SPOB is a transposition cipher, this deficiency can be overcome by cascading with a substitution cipher like MAT (to be discussed later).

4.5.2 Chi-Square test and encryption time

One of the most popular statistical tools, 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 time due to SPOB 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 SPOB along with those for Triple DES in case of .dll files are listed in table 4.3. The same is visualised in figure 4.6.

Table 4.3: χ^2 -test for SPOB with .dll files

Sl. No.	Original file	File size (bytes)	SPOB			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.dll	20480	0.109890	4912	157	06	29790	255
2	2.dll	53312	0.274725	20959	255	16	43835	255
3	3.dll	90176	0.494505	48891	255	26	66128	255
4	4.dll	118784	0.604396	1358795	255	34	1211289	255
5	5.dll	204800	1.098901	2897554	255	69	2416524	255

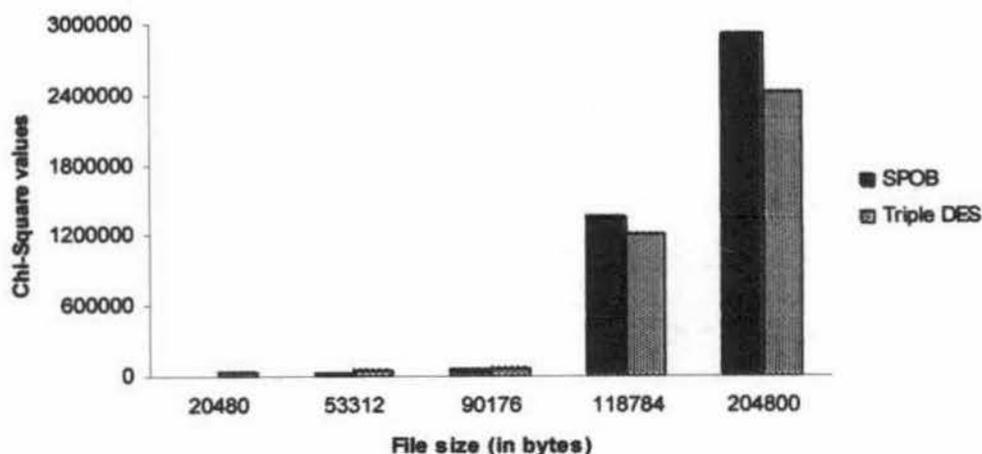


Figure 4.6: SPOB vs. Triple DES in χ^2 -test of .dll files

The performance of SPOB in χ^2 -test with .dll files is quite satisfactory compared to Triple DES. In fact, in case of larger files, the χ^2 values for SPOB are higher than those for Triple DES. Very small encryption time and large χ^2 values indicate the strength of SPOB which may be comparable to that of Triple DES. The results of the test for .exe are given by table 4.4 and figure 4.7.

Table 4.4: χ^2 -test for SPOB with .exe files

Sl. No.	Original file	File size (bytes)	SPOB			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.exe	23104	0.109890	10382	255	12	8772	255
2	2.exe	52736	0.274725	23702	255	15	43426	255
3	3.exe	131136	0.659341	858925	255	29	986693	255
4	4.exe	170496	0.934066	534803	255	49	475893	255
5	5.exe	200832	0.989011	14429014	255	58	1847377	255

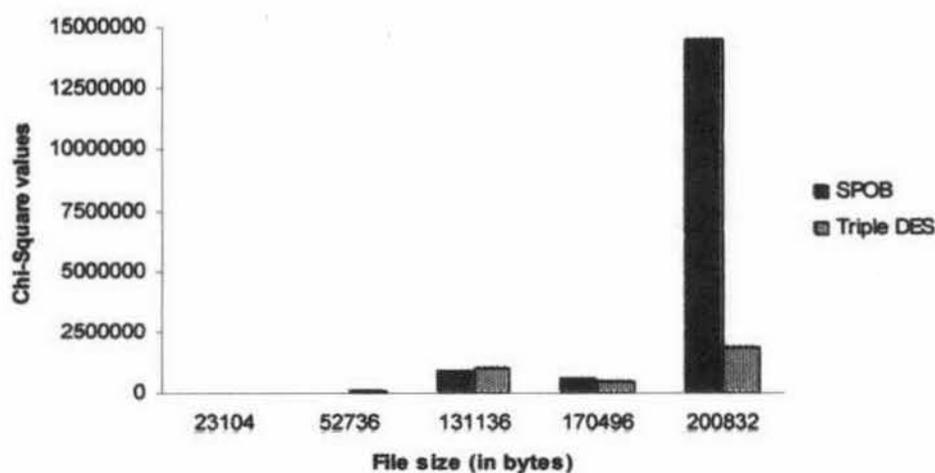


Figure 4.7: SPOB vs. Triple DES in χ^2 -test of .exe files

In case of .exe files, the test results for SPOB are even better. Compared to Triple DES, the χ^2 values for SPOB in case of most of the .exe files are larger and for other files they are too close. The degree of freedom (DF) is 255 for all the files. Further, the encryption times are much less than those of Triple DES. The test results for .jpg files are listed in table 4.5 and illustrated by figure 4.8.

Table 4.5: χ^2 -test for SPOB with .jpg files

Sl. No.	Original file	File size (bytes)	SPOB			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.jpg	28544	0.164835	7414	255	08	4331	255
2	2.jpg	71232	0.329670	3716	255	21	2916	255
3	3.jpg	105600	0.549451	6196	255	31	5227	255
4	4.jpg	160704	0.824176	9001	255	47	22314	255
5	5.jpg	216576	1.153846	12719	255	63	29824	255

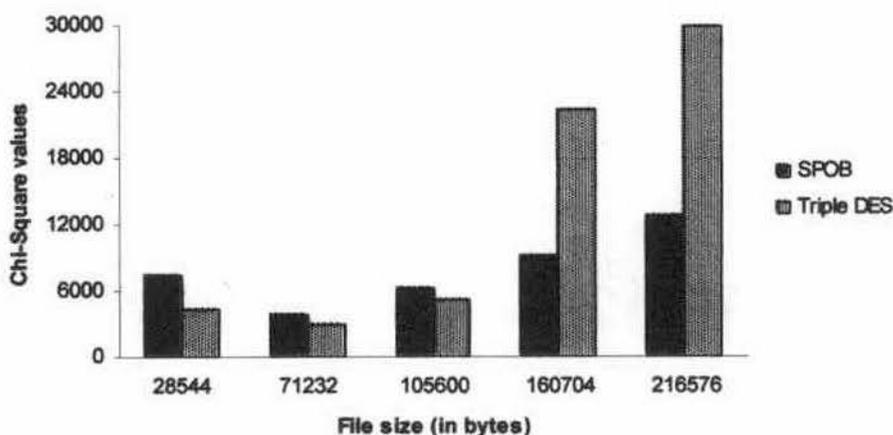


Figure 4.8: SPOB vs. Triple DES in χ^2 -test of .jpg files

Out of five files, the χ^2 values for SPOB are higher than those for Triple DES in case of three files. High χ^2 values and very small encryption times depict a good performance of SPOB. The results for .txt files are given by table 4.6 and figure 4.9.

Table 4.6: χ^2 -test for SPOB with .txt files

Sl. No.	Original file	File size (bytes)	SPOB			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	t1.txt	6976	0.054945	6213	69	02	10629	183
2	t2.txt	23808	0.109890	43081	111	07	32638	255
3	t3.txt	58688	0.329670	102298	123	17	82101	255
4	t4.txt	118784	0.604396	215996	133	35	170557	255
5	t5.txt	190784	0.989011	706203	132	55	430338	255

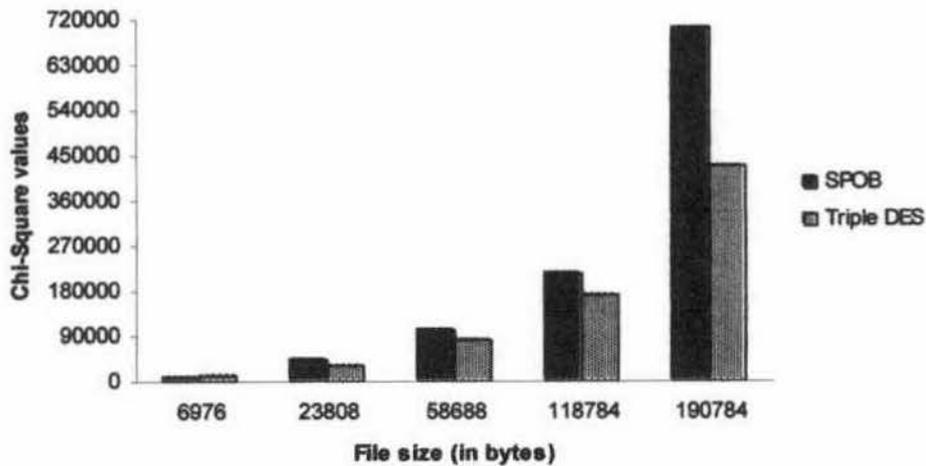


Figure 4.9: SPOB vs. Triple DES in χ^2 -test of .txt files

Although the frequency graph for SPOB was not so good in case of .txt files, it has shown a very good performance in the χ^2 -test. The χ^2 values for all but one .txt files in SPOB are higher than Triple DES. The reason for an unsatisfactory frequency graph is further explained by the only exception that the degrees of freedom (DF) for SPOB in case of all the five .txt files are bit lesser than those for Triple DES, but it does not have so much of significance.

4.5.3 Avalanche and runs

A 32-bit binary string was repeatedly encrypted with SPOB, 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 noted. This was done to examine the diffusion property of SPOB, i.e. to test how much effect can be created in the cipher-text with a very small change in the plain-text. Table 4.7 shows the results of this test for SPOB.

The table shows that SPOB can produce some amount of effect in the cipher-text with a very small change in the plain-text. There are some differences between consecutive cipher-texts in quite a fair number of cases. On average, a high difference of runs between the plain-text and the cipher-text also proves the randomness property of SPOB.

Table 4.7: Avalanche and runs in SPOB

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	848C3C14	19	14	5
1 ST	C145D450	A48C3C14	18	16	2
2 ND	0145D450	808C3C14	17	12	5
3 RD	6145D450	C48C3C24	19	14	5
4 TH	5145D450	348C3C14	21	16	5
5 TH	4945D450	858C3C14	21	14	7
6 TH	4545D450	8C8C3C14	21	14	7
7 TH	4345D450	868C3C14	19	14	5
8 TH	4045D450	048C3C14	17	13	4
9 TH	41C5D450	84AC3C14	17	16	1
10 TH	4105D450	84883C14	17	14	3
11 TH	4165D450	84CC3C14	19	14	5
12 TH	4155D450	839C3C14	21	14	7
13 TH	414DD450	848D3C14	19	16	3
14 TH	4141D450	84843C14	17	14	3
15 TH	4147D450	848E3C14	17	14	3
16 TH	4144D450	84DC3C14	19	12	7
17 TH	41455450	848C1C14	21	14	7
18 TH	41459450	848C3C14	19	15	4
19 TH	4145F450	848C7C14	17	14	3
20 TH	4145C450	848C2C14	17	16	1
21 ST	4145DE50	848C3D14	17	16	1
22 ND	4145D050	848C3414	17	16	1
23 RD	4145D650	848C3E14	19	14	5
24 TH	4145D550	848CBC14	21	16	3
25 TH	4145D4D0	84863C34	19	14	5
26 TH	4145D410	848C3C10	17	12	5
27 TH	4145D470	848C3C54	17	16	1
28 TH	4145D440	848C3C04	17	12	5
29 TH	4145D458	848C3C14	19	15	4
30 TH	4145D454	848C3C1C	21	12	9
31 ST	4145D452	848C3C16	21	14	7
32 ND	4145D451	848C3C94	20	14	6

4.6 Conclusion

SPOB 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. Since it does not involve complex arithmetic computations in its actual implementation, the proposed scheme may be recommended for the intended targets, which are very small systems with less powerful processors and very low capacity memory units.

Modulo-Arithmetic Technique (MAT)

5.1 Introduction

The schemes presented so far in this thesis were transposition ciphers and their strengths and weaknesses have already been discussed. However complicated they may be, the number of runs, and hence the weight (number of 1's or 0's), of the binary string being encrypted remained the same. Some of these algorithms were not very effective to show the diffusion property. To overcome these weaknesses, a new microprocessor-based bit-level cipher has been proposed in this chapter, where the encryption is done through a Modulo-Arithmetic Technique (MAT). Like in the previous cases, the original message is considered as a string of 512 bits, which is then divided into a number of blocks of equal size, each containing n bits, where n is any one of 8, 16, 32, 64, 128, 256. The two adjacent blocks are then added where the modulus of addition is 2^n . The result replaces the second block, first block remaining unchanged. The technique is applied in a cascaded manner, doubling the block-size in each round.

Modulo-addition has been implemented in a very simple manner in order to avoid complex computations, so that the cipher may be easily implemented in an 8-bit microprocessor system. The decryption is done by using the reverse process, i.e. modulo-subtraction.

5.2 The Modulo-Arithmetic Technique

In the proposed scheme, the source file is input as binary strings of 512 bits, though it may also be implemented for strings of larger sizes. The input string, S , is first broken into 8-bit blocks so that $S = B_1B_2B_3\dots B_{63}B_{64}$. Starting from the MSB, the blocks are paired as (B_1, B_2) , (B_3, B_4) , (B_5, B_6) and so on. The MAT operation is applied to each pair of blocks. The process is repeated, each time doubling the block-size till it is 256. Section 5.2.1 presents the process in detail.

5.2.1 Algorithm for MAT

Round 1: After breaking the input stream into blocks of 8 bits each and pairing the blocks as explained, the first member of each pair is added to the second member where the modulus of addition is 2^n for block size n ($n=8$ in *Round 1*). For example, for 4-bit blocks, the modulus of addition will be 16. The second block of the pair is replaced by the result of the modulo-addition, while the first block is kept unchanged.

This round (and also the subsequent ones) is repeated for a finite number of times and the number of iterations will form a part of the key, which is discussed in chapter 11.

Round 2: The same operation as in *Round 1*, over the intermediate string generated by it, is performed with block-size 16.

In this fashion several rounds are performed, each time doubling the block-size, till it is 256 in *Round 6*. The output from *Round 6* is the final encrypted string.

During decryption, similar type of process is followed, but the reverse operation, i.e. modulo-subtraction, has to be performed instead of modulo-addition. Further, the block-size is reduced from 256 down to 8 through the several rounds of the decryption process. The number of iterations to be performed in each round of decryption is obtained from the key.

5.2.2 The modulo-addition operation

An alternative method for modulo-addition is proposed here to make the calculations simple. The need for computation of decimal equivalents of the blocks is avoided here since we will get large decimal integer values for large binary blocks. The method proposed here is just to use binary addition and discard the 'CARRY' bit out of the MSB to get the desired result. For example, the addition of 1101 with 1001, modulus of addition not being considered, will produce 10110. In terms of decimal values, $13 + 9 = 22$. Taking the modulus of addition into consideration, which is 16 in

this case ($2^4 = 16$ for 4-bit addition), the result of addition will be 6 ($22 - 16 = 6$). Discarding the CARRY from 10110 is equivalent to subtracting 10000 (i.e. 16 in decimal). So the result will be 0110, which is equivalent to 6 in decimal. The same is applicable to any block-size. Hence, instead of going through complex computations for performing modulo-addition, just discarding the CARRY bit will serve the purpose and it will also make the microprocessor-based implementation very simple.

5.3 Example of MAT

Although the proposed scheme has been implemented for a 512-bit input string, a string of only 32 bits, say $S = 11001111000110011000110011011011$, is considered here as an example to make it uncomplicated.

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

Input:

B ₁	B ₂	B ₃	B ₄
11001111	00011001	10001100	11011011

Output:

B ₁	B ₂	B ₃	B ₄
11001111	11101000	10001100	01100111

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

Input:

B ₁	B ₂
1100111111101000	1000110001100111

Output:

B ₁	B ₂
1100111111101000	0101110001001111

Since only 32-bit string has been considered, it is not possible to proceed further and just two rounds are performed. The output from *Round 2*, say S' , is the encrypted stream, i.e. $S' = 11001111111010000101110001001111$.

Unlike the algorithms discussed in preceding chapters, it was not possible to regenerate the original block by iterating a round for block-sizes larger than 16. Also if the number of iterations to complete a cycle is too large, then it may take more time during decryption. Hence, it is proposed that the reverse process, the modulo-subtraction technique, be used for decryption. The algorithm for subtraction will not be so much different from that of addition.

Although it has been proposed to break the input string into blocks of sizes that are exponents of 2, the scheme will work equally well with any block-size. The only constraint is that, in a particular round, all the blocks should be of the same size.

5.4 Microprocessor-based implementation

To implement the scheme in an Intel 8085 microprocessor-based system, for block-sizes greater than 8 bits, the addition/subtraction is performed taking 8 bits at a time starting from the least significant byte and passing the carry/borrow to the next set of 8 bits.

In order to realize the scheme, 512 bit data is stored in memory (say FA00H onwards) and the routines for 8, 16, 32, 64, 128 and 256 bits are applied. The routines for 8-bit encryption and decryption are presented in sections 5.4.1 and 5.4.2, respectively. The routines for 16-bit encryption and decryption are put down in sections 5.4.3 and 5.4.4, respectively. Few modifications in 16-bit MAT will be necessary for higher block-sizes and these instruction-by-instruction amendments are listed in table 5.1.

5.4.1 Routine for 8-bit MAT encryption

The routine for 8-bit MAT encryption is very simple. Starting from FA00H, the first byte of each pair is read from memory into the accumulator and added to the second byte that is kept in memory. The first byte and result of addition is sent back to memory at the location reserved as result area (starting from F900H). This is repeated till the last pair of bytes has been processed and sent to the result area. The CARRY flag is ignored during addition.

Program area : F800H onwards
 Data area : FA00H onwards (512 bits, i.e. 64 locations)
 Result area : F900H onwards

Step 1 : Load C with 20H
 Step 2 : Load HL pair to point to memory location FA00H
 Step 3 : Load DE pair to point to memory location F900H
 Step 4 : Move the content of memory to A
 Step 5 : Store A into memory location pointed to by DE pair
 Step 6 : Increment both HL and DE pairs
 Step 7 : Add the content of memory to A (HL pair gives the location)
 Step 8 : Store the result into memory location pointed to by DE pair
 Step 9 : Increment both HL and DE pairs
 Step 10 : Decrement C
 Step 11 : Repeat from step 4 till C is zero
 Step 12 : Return

5.4.2 Routine for 8-bit MAT decryption

Although decryption also can be implemented using modulo-addition, modulo-subtraction has been used to make it simple and efficient. Here also, data-bytes are assumed to be stored in memory from FA00H onwards. Other assumptions are also same as in the routine for encryption. A comparison has been made to decide which byte is larger.

Step 1 : Load C with 20H
 Step 2 : Load HL pair to point to memory location FA00H
 Step 3 : Load DE pair to point to memory location F900H
 Step 4 : Move the content of memory to A
 Step 5 : Store A into memory location pointed to by DE pair
 Step 6 : Increment both HL and DE pairs
 Step 7 : Compare the content of memory with A (location given by HL pair)
 Step 8 : If $Cy = 0$ jump to step 11
 Step 9 : Subtract the content of memory from A (location given by HL pair)
 Step 10 : Jump to step 14
 Step 11 : Move the content of A to B
 Step 12 : Move the content of memory to A
 Step 13 : Subtract the content of B from A
 Step 14 : Store the result into memory location pointed to by DE pair
 Step 15 : Increment both HL and DE pairs
 Step 16 : Decrement C
 Step 17 : Repeat from step 4 till C is zero
 Step 18 : Return

5.4.3 Routine for 16-bit (and higher) MAT encryption

The data is assumed to be in the memory from F900H onwards. The bytes are read, processed and sent back to the same locations.

- Step 1 : Load C with 10H
- Step 2 : Load HL pair to point to memory location F901H
- Step 3 : Load DE pair to point to memory location F903H
- Step 4 : Clear A and Cy (XRA A)
- Step 5 : Load B with 02H
- Step 6 : Load A with the content of memory location pointed to by DE pair
- Step 7 : Add the content of memory to A with CARRY
- Step 8 : Store A into memory location pointed to by DE pair
- Step 9 : Decrement both HL and DE pairs
- Step 10 : Decrement B
- Step 11 : Repeat from step 6 till B is zero
- Step 12 : Load B with 06H
- Step 13 : Increment both HL and DE pairs
- Step 14 : Decrement B
- Step 15 : Repeat from step 13 till B is zero
- Step 16 : Decrement C
- Step 17 : Repeat from step 4 till C is zero
- Step 18 : Return

For higher block sizes, few modifications will have to be made in the routine for 16-bit for MAT encryption. The amendments to be made for each block-size higher than 16 bits are listed in table 5.1.

Table 5.1: Amendments for MAT encryption with higher block-sizes

Steps	To be changed	Block-size			
		32 bit	64 bit	128 bit	256 bit
Step 1	10H	08H	04H	02H	01H
Step 2	F901H	F903H	F907H	F90FH	F91FH
Step 3	F903H	F907H	F90FH	F91FH	F93FH
Step 5	02H	04H	08H	16H	32H
Step 12	06H	0CH	12H	18H	1EH

5.4.4 Routine for 16-bit (and higher) MAT decryption

As in the routine for encryption, the data is assumed to be in the memory from F900H onwards and the bytes are read, processed and sent back to the same locations. The routine will be exactly the same as that of encryption except at step 7, instead of

adding the content of memory to A with CARRY, it should be subtracted from A with BORROW. Hence, the whole routine is not listed here to avoid mere repetition. The modifications needed for higher block sizes will also be the same as in encryption.

5.5 Results and comparisons

The evaluation of MAT has been done using the methods already discussed in section 1.8.3. It has been tested in its weakest form, i.e., having just one pass (no iterations) in each round, so that the strength of the algorithm will increase during actual implementation. The results of the tests have been compared with those of Triple DES.

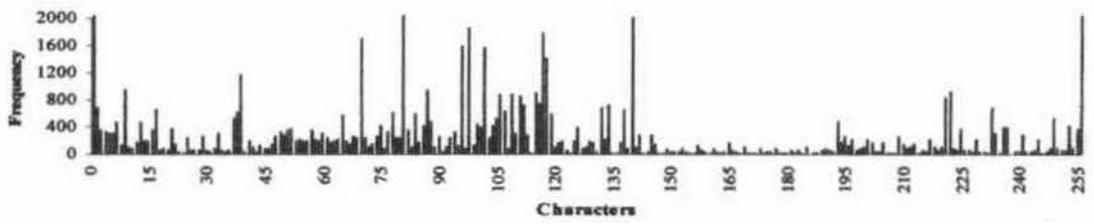
As in all the preceding algorithms, four categories of files, namely .dll, .exe, .jpg and .txt, have been considered for the purpose of testing and five different files of varying sizes have been included in each category and put through the MAT algorithm.

5.5.1 Character frequency

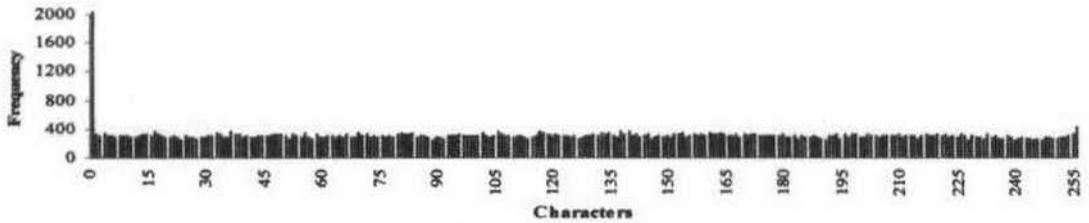
As done in previous cases, among the twenty files encrypted, the results of just one file in each category are shown here for the sake of brevity. Figure 5.1 shows the frequencies of all the 256 characters in the .dll source file and the ones obtained as results of encryption with MAT and Triple DES. Likewise, figures 5.2 through 5.4 illustrate the comparative character-frequencies for files of other three categories.

As usual, very high frequencies of few characters in some graphs have been truncated to make the low values quite visible, otherwise they will look like almost zero values.

In the original .dll file, the characters are clustered in some regions and almost negligible in some portions. The same file, when encrypted with MAT has more or less equal character-frequencies, which means that they are distributed evenly throughout the character space. The character with ASCII value 0 is the only one



(a) Original .dll file

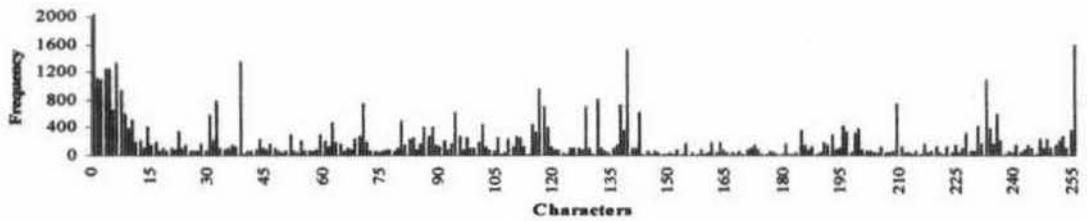


(b) .dll file encrypted with MAT

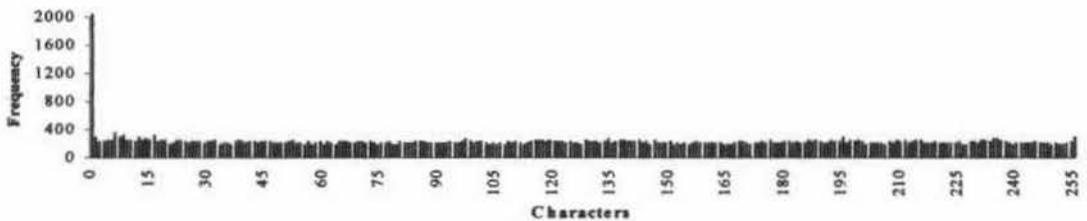


(c) .dll file encrypted with Triple DES

Figure 5.1: Character-frequencies in the source and encrypted .dll files



(a) Original .exe file

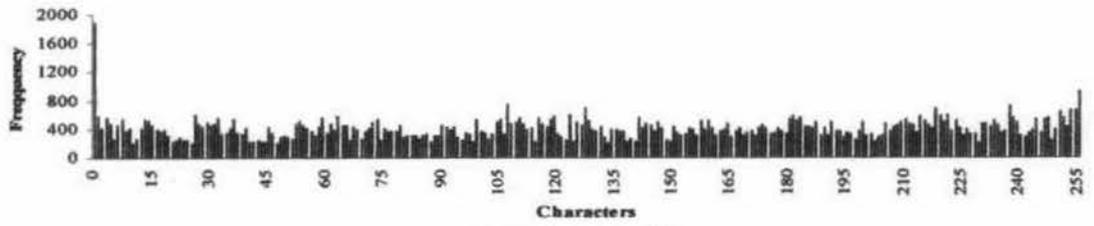


(b) .exe file encrypted with MAT

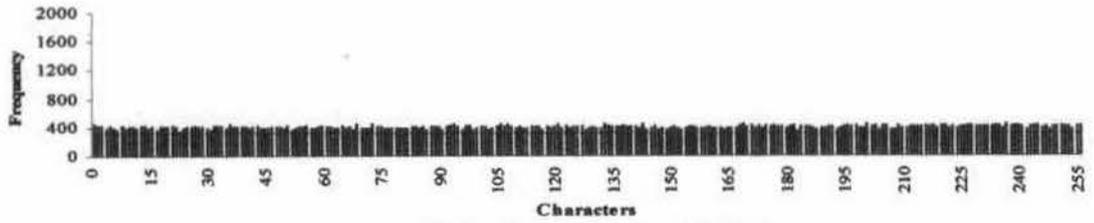


(c) .exe file encrypted with Triple DES

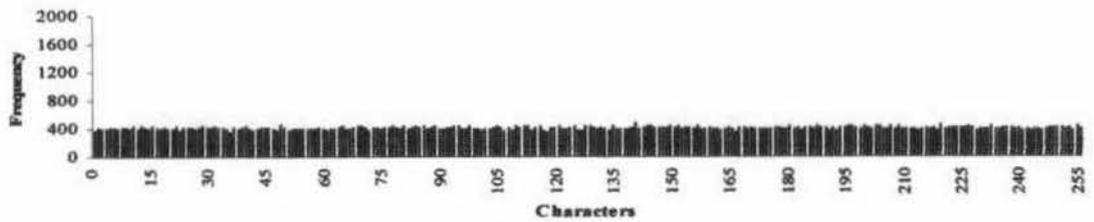
Figure 5.2: Character-frequencies in the source and encrypted .exe files



(a) Original .jpg file

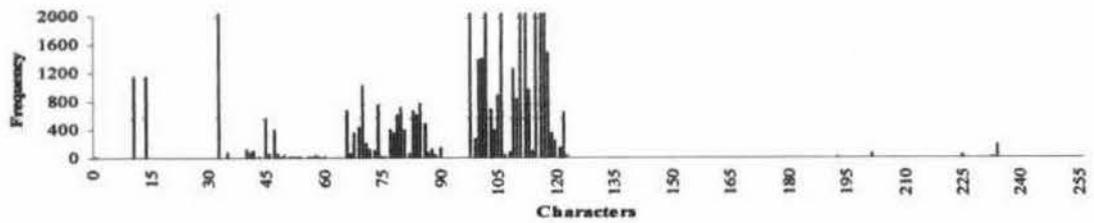


(b) .jpg file encrypted with MAT

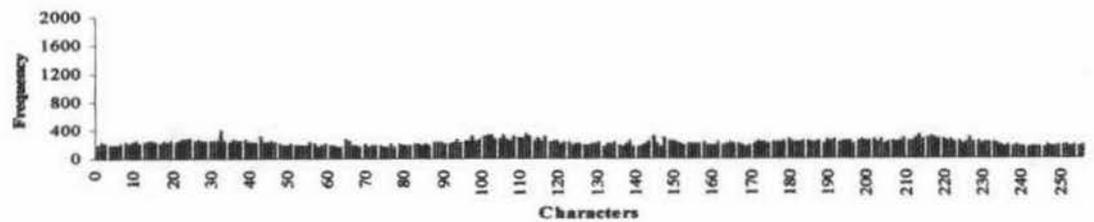


(c) .jpg file encrypted with Triple DES

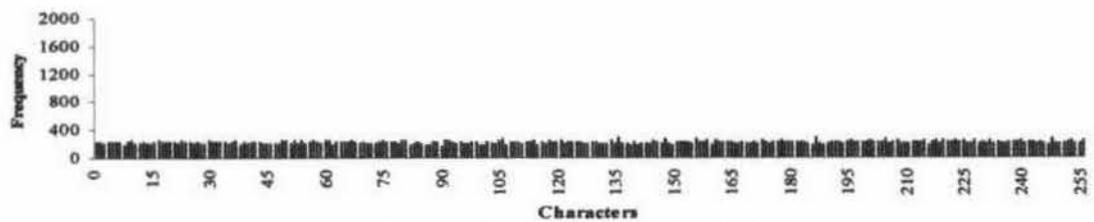
Figure 5.3: Character-frequencies in the source and encrypted .jpg files



(a) Original .txt file



(b) .txt file encrypted with MAT



(c) .txt file encrypted with Triple DES

Figure 5.4: Character-frequencies in the source and encrypted .txt files

exception, which has quite a high frequency. The result of MAT has been compared to that of Triple DES. 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 as in the case of .dll file hold true for .exe file also, because the result for the .exe file is almost similar to that of the .dll file.

The performance of MAT in case of the .jpg file is quite good as it is evident from the frequency graphs. The frequencies obtained from MAT and Triple DES are, more or less the same. The characters are homogeneously distributed in the character space for both MAT and Triple DES encrypted .jpg file.

In the original .txt file, the frequencies of almost half of the total characters are nil. This is due to the absence of the non-printable characters. The characters are fairly distributed over the character space after the file is encrypted by MAT. The result is quite comparable with that of Triple DES.

5.5.2 Chi-Square test and encryption time

To check the heterogeneity between the original and encrypted pairs of all the twenty files, the χ^2 -test, as usual, was performed. The χ^2 values and encryption times due to MAT were compared with those due to Triple DES. Each category of files has been dealt with separately. The comparative χ^2 values and encryption time for MAT along with those for Triple DES in case of .dll files are listed in table 5.2. The comparisons in table 5.2 can be visualised in figure 5.5.

Table 5.2: χ^2 -test for MAT with .dll files

Sl. No.	Original file	File size (bytes)	MAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.dll	20480	0.054945	6658	255	06	29790	255
2	2.dll	53312	0.109890	19750	255	16	43835	255
3	3.dll	90176	0.329670	41532	255	26	66128	255
4	4.dll	118784	0.494505	136269	255	34	1211289	255
5	5.dll	204800	0.714286	534082	255	69	2416524	255

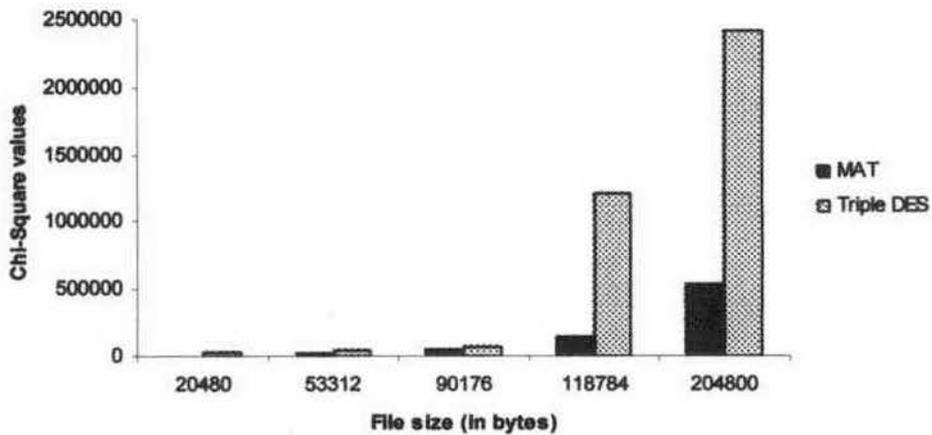


Figure 5.5: MAT vs. Triple DES in χ^2 -test of .dll files

The performance of MAT 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 all five .dll files indicate the strength of MAT. Since Triple DES is quite complicated compared to MAT, it takes a long time to encrypt a file. Table 5.3 and figure 5.6 gives the results of the test for .exe files.

Table 5.3: χ^2 -test for MAT with .exe files

Sl. No.	Original file	File size (bytes)	MAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.exe	23104	0.054945	7077	255	12	8772	255
2	2.exe	52736	0.164835	21113	255	15	43426	255
3	3.exe	131136	0.384615	978811	255	29	986693	255
4	4.exe	170496	0.549451	238430	255	49	475893	255
5	5.exe	200832	0.714286	2601366	255	58	1847377	255

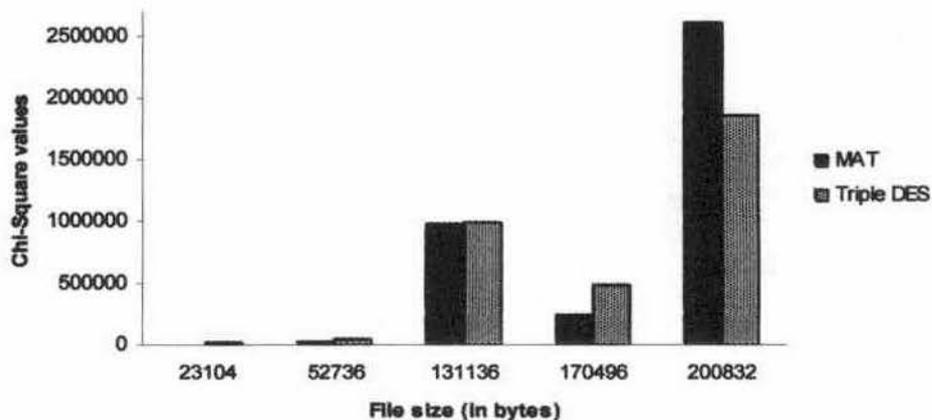


Figure 5.6: MAT vs. Triple DES in χ^2 -test of .exe files

The test results of MAT for .exe files are better than those for .dll files and, in case of some files, 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 5.4 and illustrated by figure 5.7.

Table 5.4: χ^2 -test for MAT with .jpg files

Sl. No.	Original file	File size (bytes)	MAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.jpg	28544	0.054945	4179	255	08	4331	255
2	2.jpg	71232	0.219780	2671	255	21	2916	255
3	3.jpg	105600	0.329670	5087	255	31	5227	255
4	4.jpg	160704	0.494505	21061	255	47	22314	255
5	5.jpg	216576	0.714286	28771	255	63	29824	255

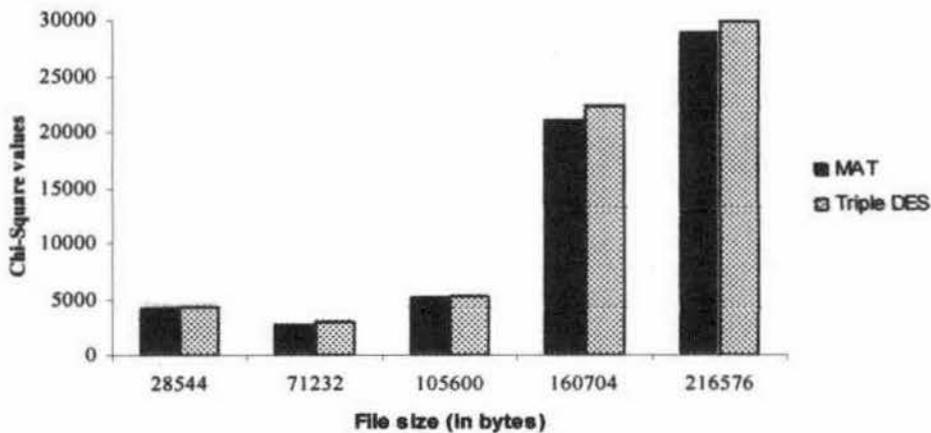


Figure 5.7: MAT vs. Triple DES in χ^2 -test of .jpg files

The result for .jpg files shows a great scope of reliability for MAT and is very close to that of Triple DES. High χ^2 values, all with 255 degrees of freedom (DF), and very small encryption time compared to Triple DES for all five files, prove the strength of MAT. The results for .txt files are given by table 5.5 and figure 5.8.

Table 5.5: χ^2 -test for MAT with .txt files

Sl. No.	Original file	File size (bytes)	MAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	t1.txt	6976	0.000000	10269	210	02	10629	183
2	t2.txt	23808	0.109890	31443	255	07	32638	255
3	t3.txt	58688	0.219780	78276	255	17	82101	255
4	t4.txt	118784	0.329670	161141	255	35	170557	255
5	t5.txt	190784	0.659341	407030	255	55	430338	255

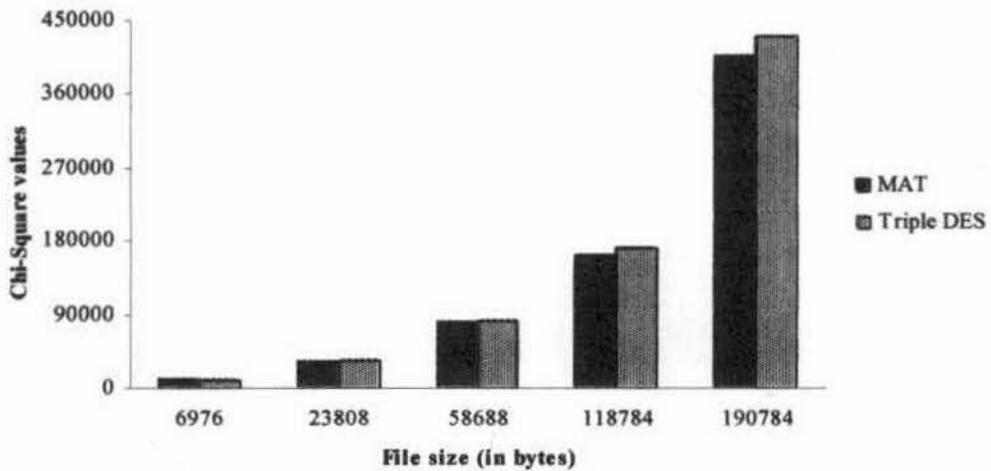


Figure 5.8: MAT vs. Triple DES in χ^2 -test of .txt files

The nature shown by MAT for .txt files is very much similar to that shown for .jpg files and, hence, needs no further explanations. The encryption time listed as 0.000000 secs. is not actually a zero value but very close to zero and has been truncated to be accommodated in the table.

5.5.3 Avalanche and runs

A small change in the plain-text has to create an avalanche in the cipher-text to prove the strength of the cipher. To examine this aspect of MAT, 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 5.6 shows the results of this test for MAT.

A close look into the table reveals that MAT can produce a good amount of effect in the cipher-text with a very small change in the plain-text. The only exception is the first byte which does not change during encryption. Nevertheless, MAT 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. Since MAT involves modulo-addition up to 256 bit, it causes a good substitution that is quite evident from the results.

Table 5.6: Avalanche and runs in MAT

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	418615AA	19	19	0
1 ST	C145D450	C106962A	18	20	2
2 ND	0145D450	A1E6760A	17	16	1
3 RD	6145D450	91D665FA	19	18	1
4 TH	5145D450	89CE5DF2	21	16	5
5 TH	4945D450	85CA59EE	21	18	3
6 TH	4545D450	83C857EC	21	14	7
7 TH	4345D450	80C554E9	19	19	0
8 TH	4045D450	8146556A	17	22	5
9 TH	41C5D450	818655AA	17	20	3
10 TH	4105D450	81E6560A	17	16	1
11 TH	4165D450	81D655FA	19	18	1
12 TH	4155D450	81CE55F2	21	16	5
13 TH	414DD450	81C255E6	19	18	1
14 TH	4141D450	81C855EC	17	16	1
15 TH	4147D450	81C555E9	17	19	2
16 TH	4144D450	81C6DC6A	19	18	1
17 TH	41455450	81C61CAA	21	18	3
18 TH	41459450	81C6760A	19	14	5
19 TH	4145F450	81C645DA	17	16	1
20 TH	4145C450	81C65DF2	17	14	3
21 ST	4145DE50	81C651E6	17	14	3
22 ND	4145D050	81C657EC	17	14	3
23 RD	4145D650	81C656EB	19	17	2
24 TH	4145D550	81C6566A	21	18	3
25 TH	4145D4D0	81C656AA	19	20	1
26 TH	4145D410	81C6560A	17	16	1
27 TH	4145D470	81C655DA	17	18	1
28 TH	4145D440	81C655F2	17	16	1
29 TH	4145D458	81C655F2	19	16	3
30 TH	4145D454	81C655EE	21	16	5
31 ST	4145D452	81C655EC	21	16	5
32 ND	4145D451	81C655EB	20	17	3

5.6 Conclusion

The use of arithmetic involving large integers has been avoided by means of a very simple alternative. This makes MAT very much feasible for implementation into the intended target without losing its credibility. Due to its simplicity and other advantages, it may be a good algorithm to be employed independently or in a cascaded manner with a transposition cipher. MAT takes little time to encode and decode though the block length is high. The encoded string will not generate any overhead bits.

Overlapped Modulo-Arithmetic Technique (OMAT)

6.1 Introduction

In the Modulo-Arithmetic Technique (MAT) discussed in chapter 5, the first member of a pair of blocks was not substituted. In the first round with block-size 8 bits, every alternate byte remained the same. Although, in the very next round, all but the first byte changed due to the change in the pairing of blocks, some patterns may still pass into the cipher-text. To overcome this problem, a variation of MAT, i.e. the **Overlapped Modulo-Arithmetic Technique (OMAT)**, is proposed in this chapter. The only modification made in this technique is the way the blocks are paired, all other operations remaining the same.

6.2 The Overlapped Modulo-Arithmetic Technique

As in MAT, the 512-bit input string, S , is first broken into 8-bit blocks so that $S = B_1B_2B_3\dots B_{63}B_{64}$. Now, instead of pairing the blocks as (B_1, B_2) , (B_3, B_4) , (B_5, B_6) etc., the blocks in OMAT are paired as (B_1, B_2) , (B_2, B_3) , (B_3, B_4) and so on, starting from the MSB. Except for the first and the last blocks, each block belongs to two adjacent pairs, as the first member of one pair and the second of the other. In other words, there is a common member in any two adjacent block-pairs, i.e. the block-pairs are overlapping and hence the name given to the technique. The rest of the algorithm is same as MAT and is not repeated here.

Due to the modification in pairing of blocks, the effect of a small change in a block reaches all the following blocks. In the first pair, B_2 gets modified after B_1 is added to it. The new value of B_2 is then added to B_3 , then B_3 to B_4 , and so on. Hence a small change in B_1 will have its impact till the last block.

So, unlike MAT, all but the first 8 bits gets modified due to substitution. This will cause a great amount of diffusion as will be evident from the results.

6.3 Example of OMAT

Although the proposed scheme has been implemented for a 512-bit input string, a string of only 64 bits has been taken to illustrate the OMAT operations. Say, $S = 1100111100011001100011001101101100101110010100101100100100001010$, is the 64-bit string representing the plain-text.

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

Input:

B ₁	B ₂	B ₃	B ₄	B ₅	B ₆	B ₇	B ₈
11001111	00011001	10001100	11011011	00101110	01010010	11001001	00001010

Output:

B ₁	B ₂	B ₃	B ₄	B ₅	B ₆	B ₇	B ₈
11001111	11101000	01110100	01001111	01111101	11001111	10011000	10100010

Round 2: Block-size = 16, number of blocks = 4

Input:

B ₁	B ₂	B ₃	B ₄
1100111111101000	01111010001001111	0111110111001111	1001100010100010

Output:

B ₁	B ₂	B ₃	B ₄
1100111111101000	0100010000110111	1100001000000110	0101101010101000

Round 3: Block-size = 32, number of blocks = 2

Input:

B ₁	B ₂
11001111111010000100010000110111	11000010000001100101101010101000

Output:

B ₁	B ₂
11001111111010000100010000110111	10010001111011101001111011011111

Since the input was only 64-bit, it is not possible to proceed further and just three rounds are performed. The output from *Round 3*, say S' , is the cipher-text, i.e. $S' = 1100111111101000010001000011011110010001111011101001111011011111$.

Similar to the case of MAT, it was not possible to regenerate the original block by iterating a round for block-sizes larger than 16. Hence, as in MAT, the reverse process, i.e. modulo-subtraction is used for decryption.

As in MAT, the input string need not always be broken into blocks of sizes that are exponents of 2, but also of any other size, of course with the only constraint that, in a particular round, all the blocks will have to be of the same size.

6.4 Microprocessor-based implementation

In order to implement the scheme in an Intel 8085 microprocessor-based system, 512-bit data is stored in memory (say F900H onwards) and the routines for 8, 16, 32, 64, 128 and 256 bits are applied. The routines for 8-bit encryption and decryption are presented in sections 6.4.1 and 6.4.2, respectively. The same for 16-bit block-size including the modifications needed for higher block-sizes are given in sections 6.4.3 and 6.4.4.

6.4.1 Routine for 8-bit OMAT encryption

Program area : F800H onwards
 Data area : F900H onwards (512 bits, i.e. 64 locations)
 Result area : F900H onwards

- Step 1 : Load C with 3EH
- Step 2 : Load HL pair to point to memory location F900H
- Step 3 : Move the content of memory to A
- Step 4 : Increment HL pair
- Step 5 : Add the content of memory to A (HL pair gives the location)
- Step 6 : Move content of A into memory (A's content does not change)
- Step 7 : Decrement C
- Step 8 : Repeat from step 4 till C is zero
- Step 9 : Return

6.4.2 Routine for 8-bit OMAT decryption

- Step 1 : Load C with 3EH
- Step 2 : Load HL pair to point to memory location F93FH
- Step 3 : Move the content of memory to A
- Step 4 : Decrement HL pair
- Step 5 : Subtract the content of memory from A (HL pair gives the location)
- Step 6 : Move content of A into memory (A's content does not change)
- Step 7 : Decrement C
- Step 8 : Repeat from step 4 till C is zero
- Step 9 : Return

6.4.3 Routine for 16-bit (and higher) OMAT encryption

The data is assumed to be in the memory from F900H onwards. The bytes are read, processed and sent back to the same locations. The routine for 16-bit OMAT is presented here and the amendments needed for block-sizes higher than 16 bits are listed in table 6.1.

- Step 1 : Load C with 20H
- Step 2 : Load HL pair to point to memory location F903H
- Step 3 : Load DE pair to point to memory location F901H
- Step 4 : Clear A and Cy (XRA A)
- Step 5 : Load B with 02H
- Step 6 : Load A with the content of memory location pointed to by DE pair
- Step 7 : Add the content of memory to A with CARRY
- Step 8 : Store A into memory (location given by HL pair)
- Step 9 : Decrement both HL and DE pairs
- Step 10 : Decrement B
- Step 11 : Repeat from step 6 till B is zero
- Step 12 : Load B with 02H
- Step 13 : Increment both HL and DE pairs
- Step 14 : Decrement B
- Step 15 : Repeat from step 13 till B is zero
- Step 16 : Decrement C
- Step 17 : Repeat from step 4 till C is zero
- Step 18 : Return

Table 6.1: Amendments for OMAT encryption with higher block-sizes

Steps	To be changed	Block-size			
		32 bit	64 bit	128 bit	256 bit
Step 1	1FH	0FH	07H	03H	01F
Step 2	F903H	F907H	F90FH	F91FH	F93FH
Step 3	F901H	F903H	F907H	F90FH	F91FH
Step 5	02H	04H	08H	10H	20H
Step 12	02H	04H	08H	10H	20H

6.4.4 Routine for 16-bit (and higher) OMAT decryption

The routines 16-bit and higher OMAT decryptions will be exactly the same as that of encryption except at step 8, instead of adding with CARRY, the content of memory should be subtracted from A with BORROW. The modifications needed for higher block sizes will also be the same as in encryption. Therefore, the whole routine is not listed here to avoid repetitions.

6.5 Results and comparisons

As in MAT, the strength and weaknesses of OMAT 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 results of the tests have been compared with those of Triple DES.

As usual, five different files of varying sizes in each of the four categories, namely .dll, .exe, .txt and .jpg, were considered for the purpose of testing and were encrypted using the OMAT algorithm.

6.5.1 Character frequency

Among the twenty files encrypted, the results of just one file in each of the four categories are shown here for the sake of brevity. The variation of frequencies of all the 256 ASCII characters in the .dll source file and the ones obtained as results of encryption with OMAT and Triple DES are shown in figure 6.1. Similarly, figures 6.2 through 6.4 illustrate the comparative character-frequencies for files of the other three categories.

As done before, very high frequencies of few characters in some graphs have been truncated to make the low values quite visible, otherwise they will look like almost zero values.

The characters in the original .dll file are clustered in some regions and negligible in some portions. In the OMAT encrypted .txt file, characters are more or

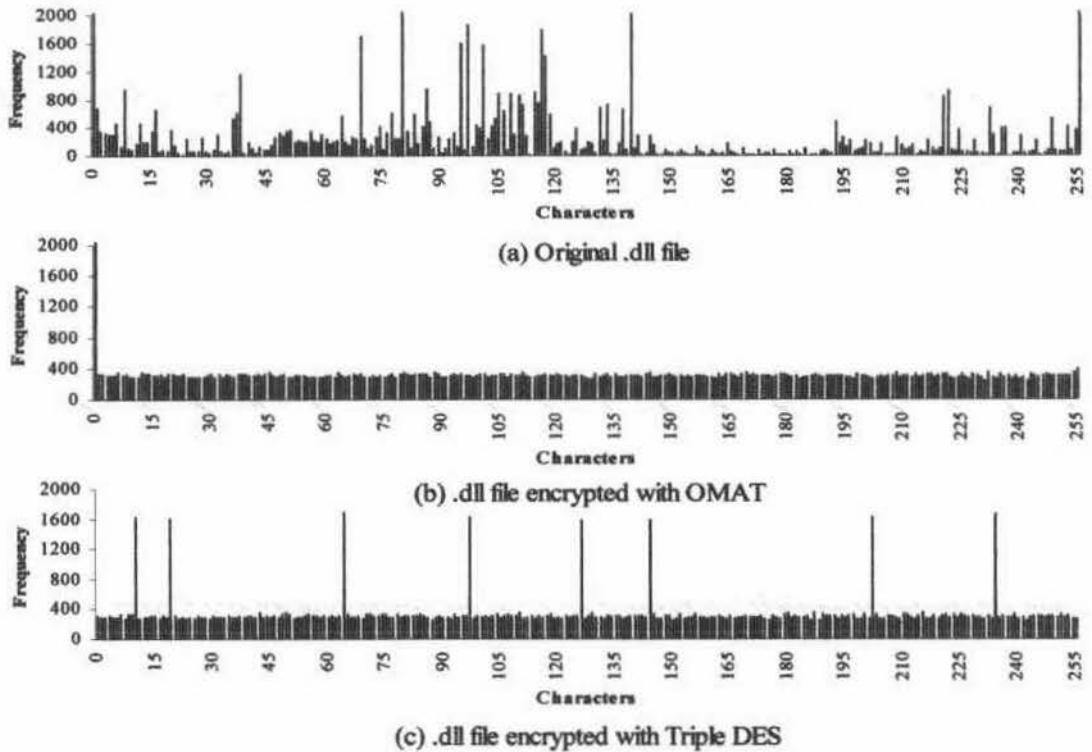


Figure 6.1: Character-frequencies in the source and encrypted .dll files

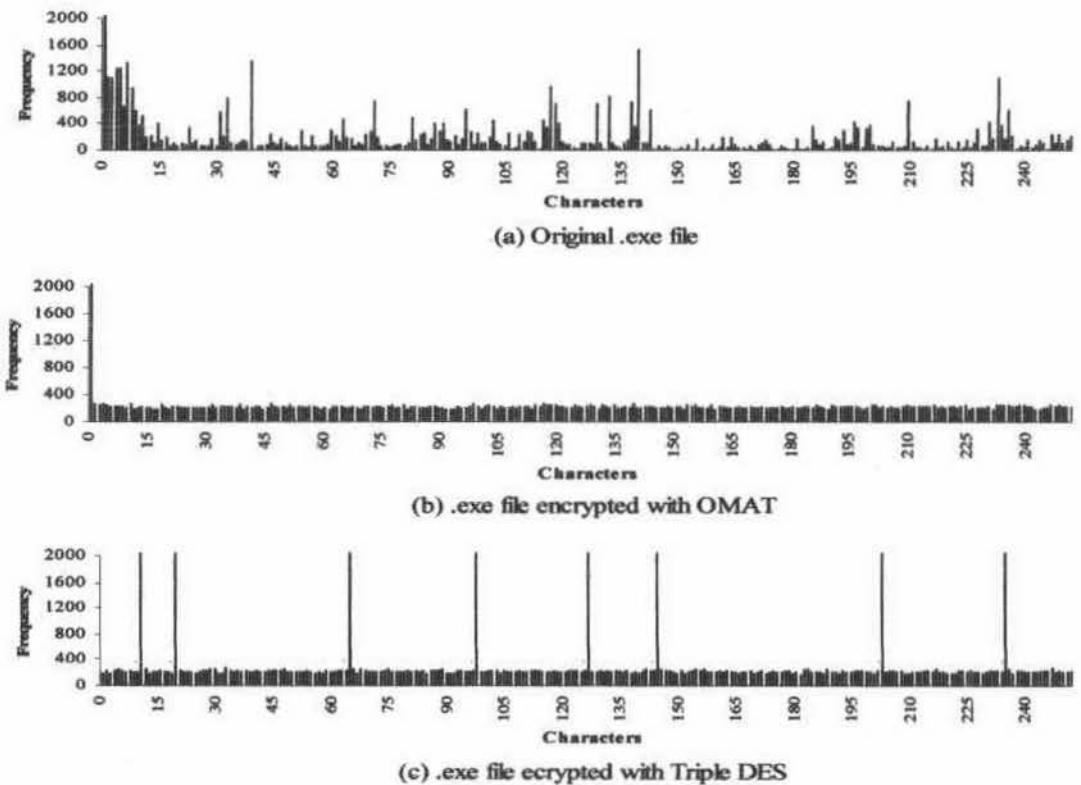


Figure 6.2: Character-frequencies in the source and encrypted .exe files

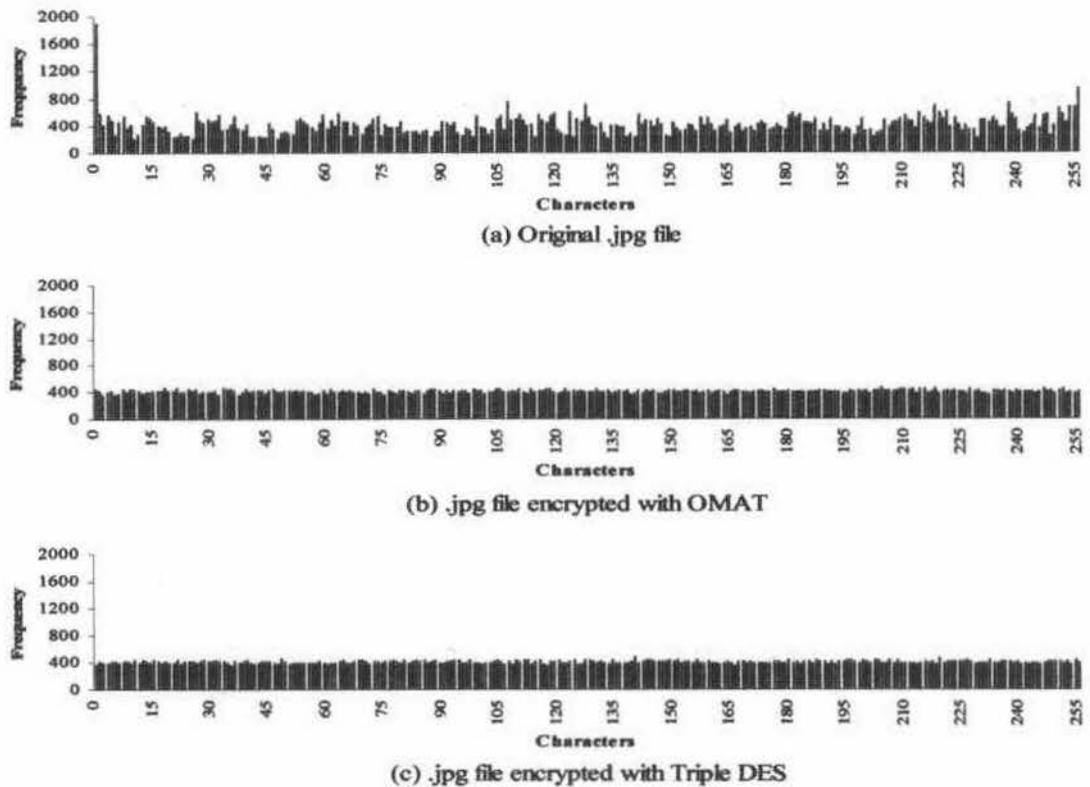


Figure 6.3: Character-frequencies in the source and encrypted .jpg files

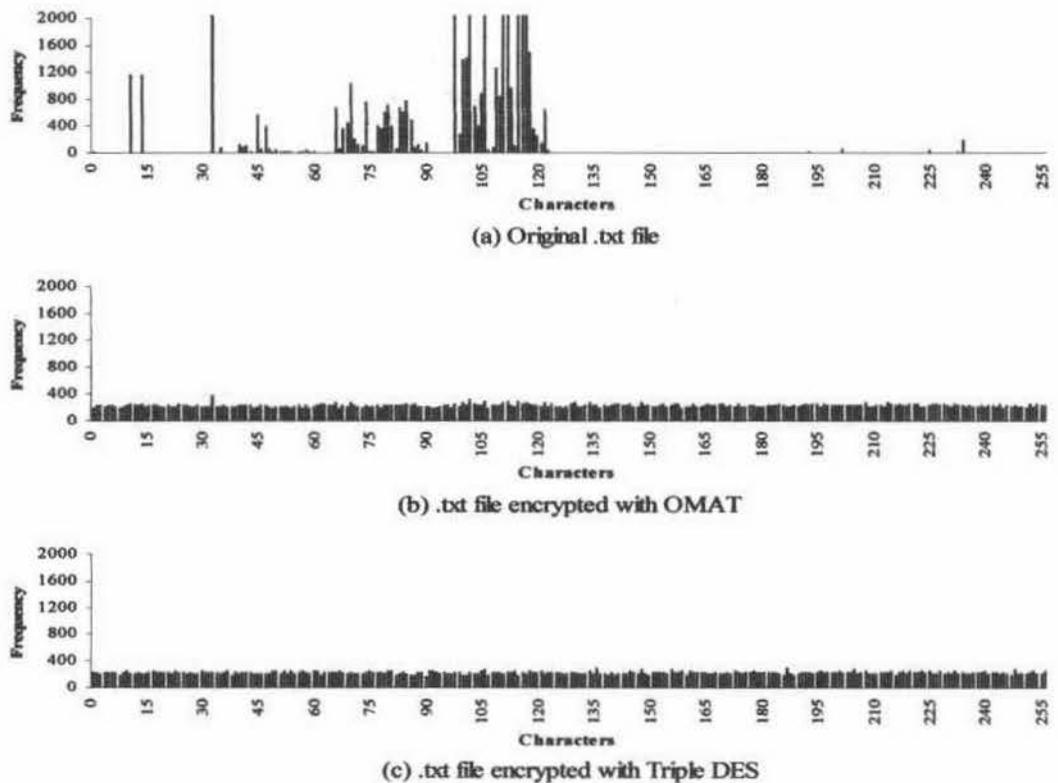


Figure 6.4: Character-frequencies in the source and encrypted .txt files

less evenly distributed throughout the character space except for the character with ASCII value 0, which has a very high frequency compared to others.

In the result of Triple DES, most of the characters are distributed evenly in the character space, but quite a few have abruptly high frequencies. It is not claimed that OMAT excels over Triple DES, but it is worth mentioning here that OMAT has shown a good result case of .dll files.

Similar explanations as in the case of .dll file hold true for .exe file also because the results for .exe file are almost similar to that of the .dll file.

OMAT has shown a very good performance for .jpg files also. It is very much clear from the frequency-graph that the characters are uniformly distributed in the character space for both OMAT and Triple DES encrypted .jpg file. The frequencies obtained from OMAT and Triple DES are more or less the same.

Due to the absence of the non-printable characters, the frequencies of almost half of the total characters are nil in the original .txt file. The characters are evenly distributed over the character space after the file is encrypted with OMAT. Both OMAT and Triple DES have shown the same degree of performance for .txt files.

6.5.2 Chi-Square test and encryption time

The heterogeneity between the original and encrypted pairs of all the twenty files was tested with the χ^2 -test. The test results due to OMAT as compared to those of Triple DES for .dll files are listed in table 6.2, which can be visualised in figure 6.5.

Table 6.2: χ^2 -test for OMAT with .dll files

Sl. No.	Original file	File size (bytes)	OMAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.dll	20480	0.054945	7808	255	06	29790	255
2	2.dll	53312	0.164835	20495	255	16	43835	255
3	3.dll	90176	0.274725	42195	255	26	66128	255
4	4.dll	118784	0.384615	14224	255	34	1211289	255
5	5.dll	204800	0.659341	142224	255	69	2416524	255

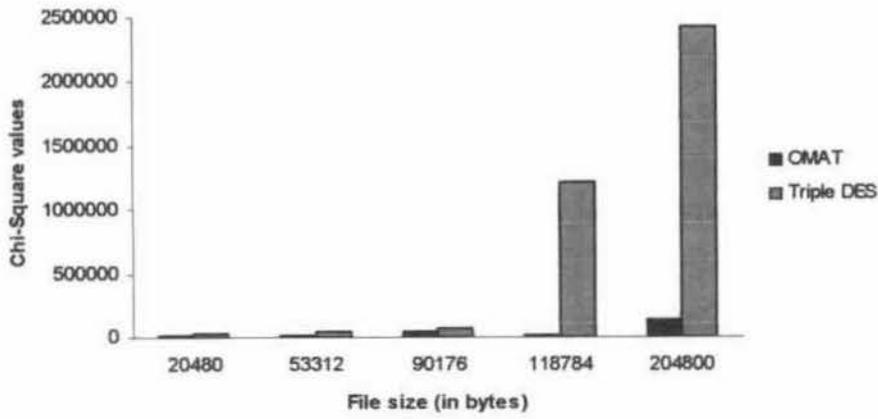


Figure 6.5: OMAT vs. Triple DES in χ^2 -test of .dll files

Large χ^2 values with 255 degrees of freedom (DF) and very small encryption time for all five .dll files indicate a good performance of OMAT. However, it is bit weak compared to Triple DES. On the other hand, compared to OMAT, Triple DES takes a long time to encrypt a file.

The results of the test for .exe files are listed in table 6.3 and illustrated by figure 6.6. OMAT shows a better performance in case of .exe files. Besides, for some files, OMAT looks even better than Triple DES. Moreover, the encryption times are much less than that of Triple DES.

Table 6.3: χ^2 -test for OMAT with .exe files

Sl. No.	Original file	File size (bytes)	OMAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.exe	23104	0.054945	7521	255	12	8772	255
2	2.exe	52736	0.164835	22526	255	15	43426	255
3	3.exe	131136	0.384615	985682	255	29	986693	255
4	4.exe	170496	0.494505	250534	255	49	475893	255
5	5.exe	200832	0.549451	2690803	255	58	1847377	255

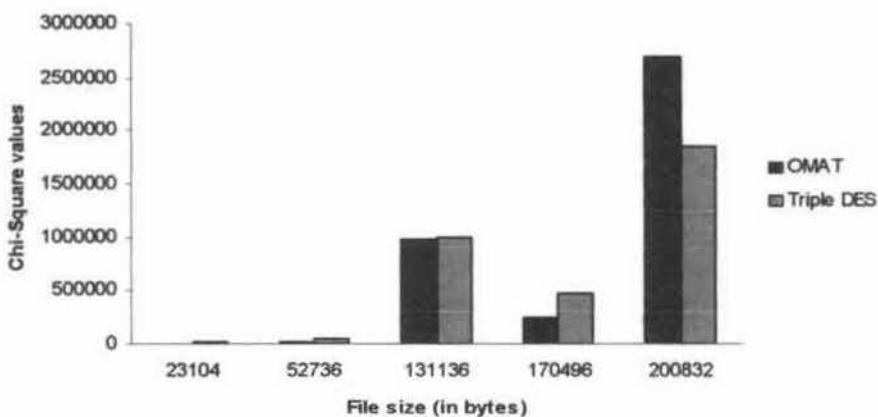


Figure 6.6: OMAT vs. Triple DES in χ^2 -test of .exe files

Table 6.4 and figure 6.7 illustrate the test results for .jpg files. For .jpg files, OMAT shows a very good performance and is very close to that of Triple DES. The strength of OMAT is proved by high χ^2 values, all with 255 degrees of freedom (DF), and very small encryption time compared to Triple DES.

Table 6.4: χ^2 -test for OMAT with .jpg files

Sl. No.	Original file	File size (bytes)	OMAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.jpg	28544	0.054945	4258	255	08	4331	255
2	2.jpg	71232	0.219780	2696	255	21	2916	255
3	3.jpg	105600	0.329670	5198	255	31	5227	255
4	4.jpg	160704	0.494505	21824	255	47	22314	255
5	5.jpg	216576	0.659341	29028	255	63	29824	255

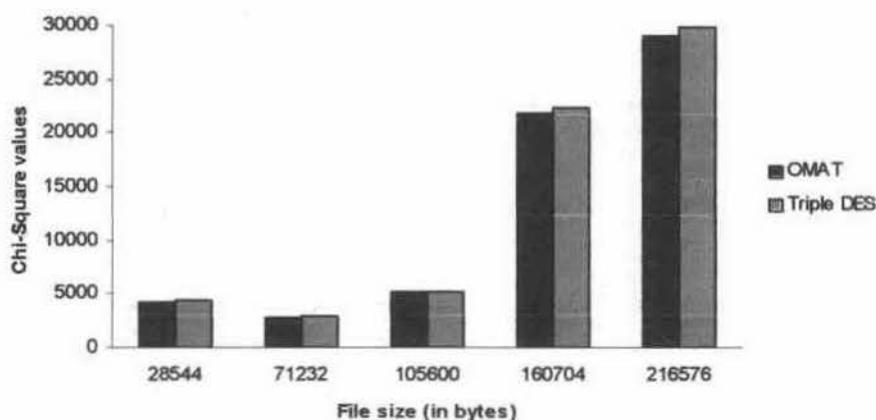


Figure 6.7: OMAT vs. Triple DES in χ^2 -test of .jpg files

The results for .txt files are given by table 6.5 and figure 6.8. Similar types of performances are shown by OMAT for .jpg and .txt files and, hence, needs no further explanations. The encryption time listed as 0.000000 secs. is not actually a zero value but very close to zero and has been truncated to be accommodated in the table.

Table 6.5: χ^2 -test for OMAT with .txt files

Sl. No.	Original file	File size (bytes)	OMAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	t1.txt	6976	0.000000	10225	228	02	10629	183
2	t2.txt	23808	0.054945	31685	255	07	32638	255
3	t3.txt	58688	0.164835	79093	255	17	82101	255
4	t4.txt	118784	0.384615	161812	255	35	170557	255
5	t5.txt	190784	0.604396	415275	255	55	430338	255

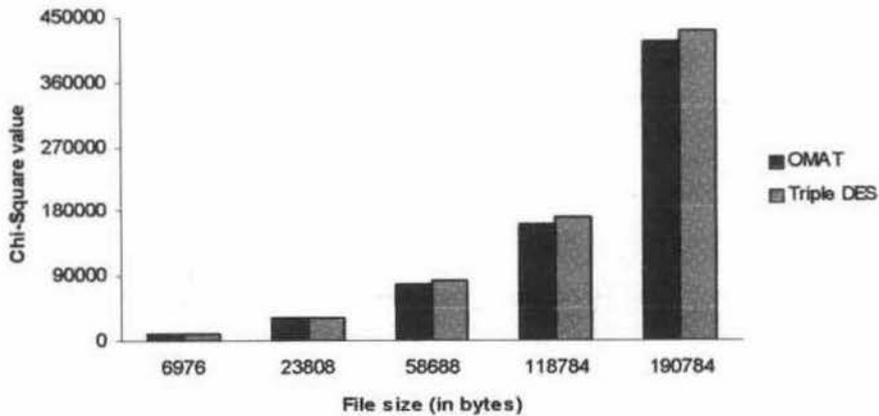


Figure 6.8: OMAT vs. Triple DES in χ^2 -test of .txt files

6.5.3 Avalanche and runs

To examine the effect of a small change in the plain-text on the cipher-text, a 32-bit binary string was repeatedly encrypted with OMAT, first keeping the original string unaltered, and subsequently each time complementing one bit of the plain-text. As done before, 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 6.6 shows the results of this test for OMAT.

A close look into the table reveals that OMAT can produce a good amount of effect in the cipher-text with a very small change in the plain-text. This can be understood by examining the consecutive cipher-texts, since there is difference of only one bit between any two consecutive plain-texts. The only exception is the first byte which does not change during encryption. Nevertheless, OMAT causes a sufficient amount of diffusion.

In addition, there are a lot of differences in runs between the plain-text and the cipher-text in most of the cases. Since OMAT involves modulo-addition up to 256 bit, it causes a good substitution that is quite evident from the results.

Table 6.6: Avalanche and runs in OMAT

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	41869C30	19	13	6
1 ST	C145D450	C1069B30	18	14	4
2 ND	0145D450	01461BD0	17	13	4
3 RD	6145D450	61A6DC70	19	15	4
4 TH	5145D450	5196BC50	21	19	2
5 TH	4945D450	498EAC40	21	17	4
6 TH	4545D450	458AA438	21	19	2
7 TH	4345D450	4388A034	19	15	4
8 TH	4045D450	40859A2E	17	17	0
9 TH	41C5D450	41061B30	17	13	4
10 TH	4105D450	41465BB0	17	17	0
11 TH	4165D450	41A6BC70	19	15	4
12 TH	4155D450	4196AC50	21	19	2
13 TH	414DD450	418EA440	19	15	4
14 TH	4141D450	41A29828	17	15	2
15 TH	4147D450	41889E34	17	15	2
16 TH	4144D450	41859B2E	19	17	2
17 TH	41455450	41831BB0	21	13	8
18 TH	41459450	41835BF0	19	13	6
19 TH	4145F450	4183BC50	17	15	2
20 TH	4145C450	41838C40	17	13	4
21 ST	4145DE50	4183A438	17	15	2
22 ND	4145D050	4183982C	17	15	2
23 RD	4145D650	41869E32	19	15	4
24 TH	4145D550	41869D32	21	16	5
25 TH	4145D4D0	41869BB0	19	15	4
26 TH	4145D410	41869BF0	17	13	4
27 TH	4145D470	41869C50	17	15	2
28 TH	4145D440	41869C40	17	13	4
29 TH	4145D458	41869C38	19	13	6
30 TH	4145D454	41869C34	21	15	6
31 ST	4145D452	41869C32	21	15	6
32 ND	4145D451	41869C31	20	14	6

6.6 Conclusion

Although quite a lot of modulo-arithmetic is involved, OMAT is very much feasible for implementation into the intended targets without losing its credibility. Its performance, compared to MAT, is better due to the modifications made in the algorithm. Its strength may be enhanced by using it in a cascaded manner with a transposition cipher like BET or SPOB. The time taken to encode and decode by OMAT is very little compared to Triple DES. Moreover, the encoded string will not generate any overhead bits. Creating an option for the choice of block within a block-pair for substitution after the modulo-addition may enhance security and this option may form a part of the key, which is discussed in chapter 11.

Modified Modulo-Arithmetic Technique (MMAT)

7.1 Introduction

In chapter 6, OMAT had been proposed to overcome some deficiencies of MAT. Although OMAT showed a better performance compared to MAT, one can notice from the test results that the first byte of the plain-text never changes. The rest of the plain-text shows a lot of changes in the cipher-text even with a small change before encryption. The Modified Modulo-Arithmetic Technique (MMAT), a better variation of MAT, has been proposed in this chapter. In OMAT, the modification was done in the pairing of blocks whereas in MMAT it the modification in the arithmetic operation that causes the difference. The pairing of blocks is done in the same manner as in simple MAT. It can be noticed through the results that MMAT will cause changes even in the first byte of the plain-text, which was not achievable in MAT or OMAT.

7.2 The Modified Modulo-Arithmetic Technique

As in MAT, the source file is input as binary strings of 512 bits, though it may also be implemented for strings of larger sizes. The input string, S , is first broken into 8-bit blocks so that $S = B_1B_2B_3\dots B_{63}B_{64}$. Starting from the MSB, the blocks are paired as (B_1, B_2) , (B_3, B_4) , (B_5, B_6) and so on. The rounds in MMAT are same as in MAT but the operation part in each round is different.

MMAT can be treated as a two-pass MAT. In the first pass, the two members of a block-pair are added and the first member is replaced by the result. In the second pass, the two blocks are added again to replace the second block. The whole operation can be done in a single pass, one after the other, before moving to the next block-pair. Apart from the number of iterations in each round, the order in which the two blocks of a pair are substituted may be altered and may form a part of the key, which is discussed in chapter 11.

As in MAT and OMAT, modulo-subtraction is performed during decryption with the block-size reducing from 256 down to 8 through the several rounds.

Due to the modification in the operation part in each round, each and every block gets substituted in MMAT. In the first pair, B_1 gets modified after B_2 is added to it. Immediately B_2 gets modified after the new content of B_1 is added to it. Similarly, substitutions are made in B_3, B_4, B_5, B_6 , and so on. So, unlike MAT, all the blocks get modified due to substitution and in contrast to OMAT, even the first byte gets substituted. Hence a great amount of diffusion may be caused in MMAT.

7.3 Example of MMAT

As in OMAT, though the proposed algorithm has been implemented for a 512-bit input string, only a 64-bit string has been taken to illustrate the MMAT operations. For any (B_i, B_{i+1}) , substitutions have been made in the order B_i first and then B_{i+1} . Say, $S = 1100111100011001100011001101101100101110010100101100100100001010$, is the 64-bit string representing the plain-text.

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

Input:

B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
11001111	00011001	10001100	11011011	00101110	01010010	11001001	00001010

Output:

B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
11101000	00000001	01100111	01000010	10000000	11010010	11010011	11011101

Round 2: Block-size = 16, number of blocks = 4

Input:

B_1	B_2	B_3	B_4
1110100000000001	0110011101000010	1000000011010010	1101001111011101

Output:

B_1	B_2	B_3	B_4
0100111101000011	1011011010000101	0101010010101111	0010100010001100

Round 3: Block-size = 32, number of blocks = 2

Input:

B ₁	B ₂
01001111010000111011011010000101	01010100101011110010100010001100

Output:

B ₁	B ₂
10100011111100101101111100010001	11111000101000100000011110011101

For a 64-bit string, only three rounds are possible. The cipher-text, say S', is obtained by concatenating the blocks from the output of *Round 3*, in other words, S' = 1010001111110010110111110001000111111000101000100000011110011101.

As already suggested in the previous chapters, the input string for MMAT need not always be broken into blocks of sizes that are exponents of 2, but also of any other size, but in a particular round, all the blocks will have to be of the same size.

7.4 Microprocessor-based implementation

The routines for MMAT are similar to those of MAT since there is no difference in the way the blocks are paired. The main difference in is the operation part. In MMAT the operations have to be done twice, once replacing the first block and then again replacing the second block.

In the Intel 8085-based implementation of the scheme, a 512-bit data is assumed to be stored in memory from F900H onwards and the routines for 8, 16, 32, 64, 128 and 256 bits are applied. The routines for 8-bit encryption and decryption are presented in sections 7.4.1 and 7.4.2, respectively. The routines for 16-bit encryption and decryption along with the amendments needed for higher block-sizes are given in sections 7.4.3 and 7.4.4, respectively.

7.4.1 Routine for 8-bit MMAT encryption

The routine for 8-bit MMAT encryption is very simple. Starting from F900H, the first and the second bytes are added twice, once replacing the first byte and again

replacing the second byte. This is repeated till the last pair of bytes has been processed. The CARRY flag is ignored during addition.

- Step 1 : Load C with 20H
- Step 2 : Load HL pair to point to memory location F900H
- Step 3 : Load DE pair to point to memory location F901H
- Step 4 : Move the content of memory location pointed to by DE pair to A
- Step 5 : Add the content of memory to A (HL pair gives the location)
- Step 6 : Store the result into memory (HL pair gives the location)
- Step 7 : Move the content of memory location pointed to by DE pair to A
- Step 8 : Add the content of memory to A (HL pair gives the location)
- Step 9 : Store the result into memory location pointed to by DE pair
- Step 10 : Load B with 02H
- Step 11 : Increment both HL and DE pairs
- Step 12 : Decrement B
- Step 13 : Repeat from step 11 till B is zero
- Step 14 : Decrement C
- Step 15 : Repeat from step 4 till C is zero
- Step 16 : Return

7.4.2 Routine for 8-bit MMAT decryption

In this case also, data-bytes are assumed to be stored in memory from F900H onwards. Other assumptions are also same as in the routine for encryption. The first byte is subtracted from the second byte. Then the second byte is subtracted from the new value of the first byte.

- Step 1 : Load C with 20H
- Step 2 : Load HL pair to point to memory location F900H
- Step 3 : Load DE pair to point to memory location F901H
- Step 4 : Move the content of memory location pointed to by DE pair to A
- Step 5 : Subtract memory content from A (HL pair gives the location)
- Step 6 : Store the result into memory location pointed to by DE pair
- Step 7 : Move the content of A to B
- Step 8 : Move the content of memory to A (HL pair gives the location)
- Step 9 : Subtract B from A
- Step 10 : Store the result into memory (HL pair gives the location)
- Step 11 : Load B with 02H
- Step 12 : Increment both HL and DE pairs
- Step 13 : Decrement B
- Step 14 : Repeat from step 12 till B is zero
- Step 15 : Decrement C
- Step 16 : Repeat from step 4 till C is zero
- Step 17 : Return

7.4.3 Routine for 16-bit (and higher) MMAT encryption

The data is assumed to be in the memory from F900H onwards. The bytes are read, processed and sent back to the same locations.

- Step 1 : Load C with 10H
- Step 2 : Load HL pair to point to memory location F901H
- Step 3 : Load DE pair to point to memory location F903H
- Step 4 : Clear A and Cy (XRA A)
- Step 5 : Load B with 02H
- Step 6 : Load A with the content of memory location pointed to by DE pair
- Step 7 : Add the content of memory to A with CARRY
- Step 8 : Store A into memory location pointed to by DE pair
- Step 9 : Decrement both HL and DE pairs
- Step 10 : Decrement B
- Step 11 : Repeat from step 6 till B is zero
- Step 12 : Load B with 02H
- Step 13 : Increment both HL and DE pairs
- Step 14 : Decrement B
- Step 15 : Repeat from step 13 till B is zero
- Step 16 : Clear A and Cy (XRA A)
- Step 17 : Load B with 02H
- Step 18 : Load A with the content of memory location pointed to by DE pair
- Step 19 : Add the content of memory to A with CARRY
- Step 20 : Store A into memory
- Step 21 : Decrement both HL and DE pairs
- Step 22 : Decrement B
- Step 23 : Repeat from step 18 till B is zero
- Step 24 : Load B with 06H
- Step 25 : Increment both HL and DE pairs
- Step 26 : Decrement B
- Step 27 : Repeat from step 25 till B is zero
- Step 28 : Decrement C
- Step 29 : Repeat from step 4 till C is zero
- Step 30 : Return

The same routine, of course with a few modifications, may be used for MMAT encryption with block-sizes higher than 16 bits. The amendments to be made for each block-size higher than 16 bits are listed in table 7.1.

7.4.4 Routine for 16-bit (and higher) MMAT decryption

With the same assumptions as in encryption, the routine will be exactly the same for decryption except at steps 7 and 19, instead of adding the content of memory to A with CARRY, it should be subtracted from A with BORROW. For this reason,

the whole routine is not listed here to avoid mere repetition. The modifications needed for higher block-sizes will also be the same as in encryption, listed in table 7.1.

Table 7.1: Amendments for MMAT with higher block-sizes

Steps	To be changed	Block size			
		32 bit	64 bit	128 bit	256 bit
Step 1	10H	08H	04H	02H	01H
Step 2	F901H	F903H	F907H	F90FH	F91FH
Step 3	F903H	F907H	F90FH	F91FH	F93FH
Step 5	02H	04H	08H	16H	32H
Step 12	02H	04H	08H	16H	32H
Step 17	02H	04H	08H	16H	32H
Step 24	06H	0CH	12H	18H	1EH

7.5 Results and comparisons

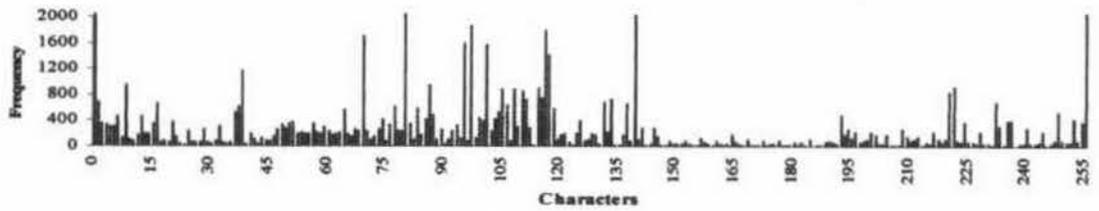
As in MAT and OMAT, the strength and weaknesses of MMAT has also been tested with just one pass (no iterations) in each round, i.e. in its weakest form, so that the strength of the algorithm may increase during actual implementation. The results of the tests have been compared with those of Triple DES.

All the five different files of varying sizes in each of the four categories, namely .dll, .exe, .txt and .jpg, were considered for the purpose of testing and encrypted using the MMAT algorithm.

7.5.1 Character frequency

Presenting the results for all the twenty files will make this thesis too long. Therefore, the results of just one file in each of the four categories are given here for the sake of brevity. The variation of frequencies of all the 256 ASCII characters in the source and the encrypted files have been compared graphically in this section.

The comparative character-frequencies for .dll source file, MMAT encrypted file and Triple DES encrypted file are illustrated in figure 7.1. Similarly, figure 7.2 depicts the comparison for .exe file, figure 7.3 for .jpg file and figure 7.4 for .txt file.



(a) Original .dll file

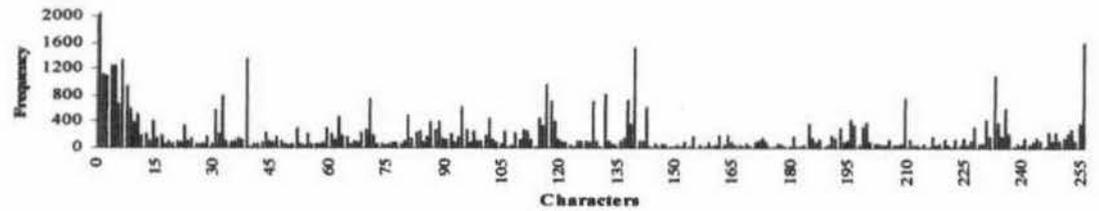


(b) .dll file encrypted with MMAT

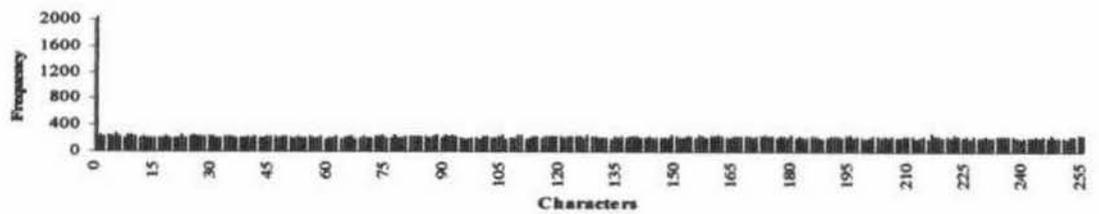


(c) .dll file encrypted with Triple DES

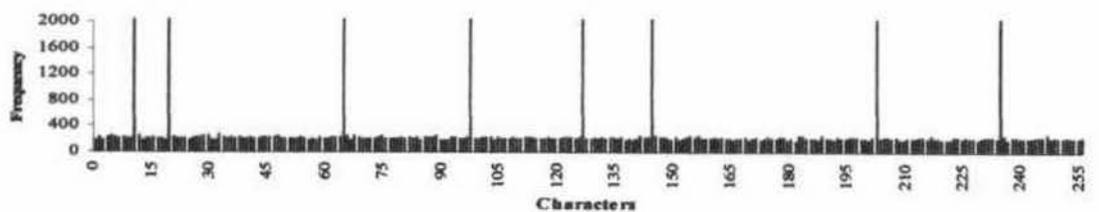
Figure 7.1: Character-frequencies in the source and encrypted .dll files



(a) Original .exe file

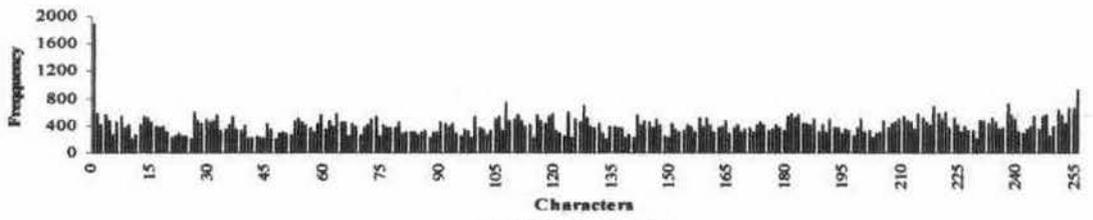


(b) .exe file encrypted with MMAT

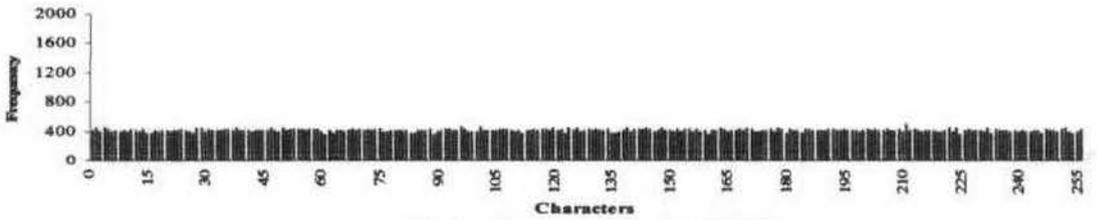


(c) .exe file encrypted with Triple DES

Figure 7.2: Character-frequencies in the source and encrypted .exe files



(a) Original .jpg file

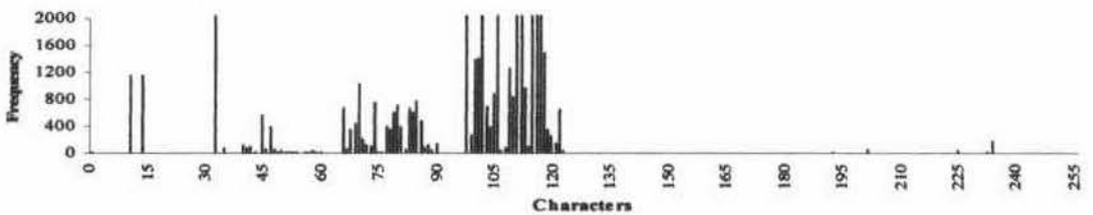


(b) .jpg file encrypted with MMAT

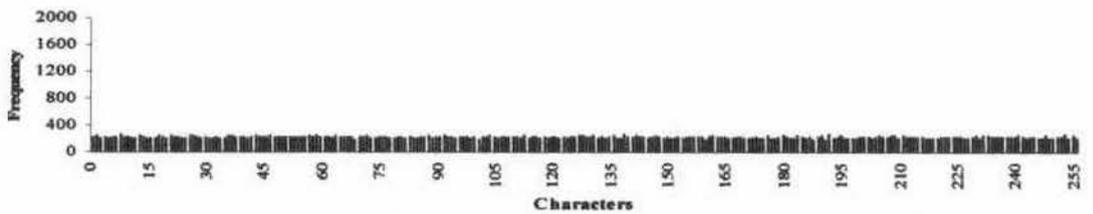


(c) .jpg file encrypted with Triple DES

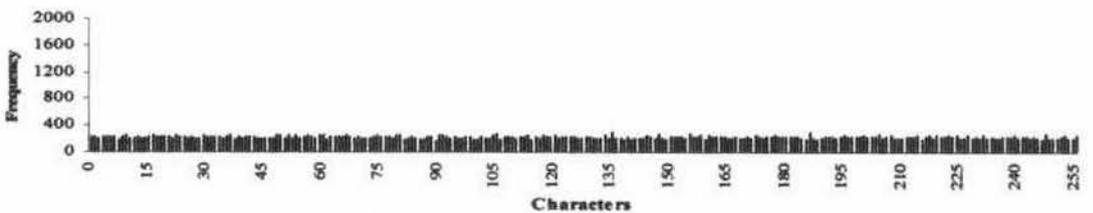
Figure 7.3: Character-frequencies in the source and encrypted .jpg files



(a) Original .txt file



(b) .txt file encrypted with MMAT



(c) .txt file encrypted with Triple DES

Figure 7.4: Character-frequencies in the source and encrypted .txt files

The graphs obtained for MMAT and OMAT are more or less same to the open eye, but the actual data differ a lot. The .dll file, when encrypted with MMAT has more or less equal character-frequencies with an exception that the character with ASCII value 0 has very a high frequency. All other characters are distributed quite evenly throughout the character space. The result of MMAT in case of the .dll file is better compared to that of Triple DES. 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.

The same explanations as in the case of the .dll file hold true for the .exe file also because the results for the .exe file are almost similar to that of the .dll file.

The frequencies obtained from MMAT and Triple DES for the .jpg file are almost same. The characters are uniformly distributed in the character space for both MMAT and Triple DES encrypted .jpg file. Therefore the performance of MMAT in case of the .jpg file may be accepted to be quite a good one and comparable to that of Triple DES.

Due to the absence of the non-printable characters, the frequencies of almost half of the total characters are nil in the case of the original .txt file. The characters are fairly distributed over the character space after the files are encrypted by MMAT and the result is very much comparable to that of Triple DES.

7.5.2 Chi-Square test and encryption time

All the twenty pairs of original and encrypted files were put through the test for heterogeneity using the most popular χ^2 -test. The results for each category of files have been discussed separately. The χ^2 values and encryption times for .dll files due to MMAT as compared to those of Triple DES are listed in table 7.2, which can be visualised in figure 7.5.

Although it is better than MAT and OMAT, from the results produced by MMAT for .dll files, it appears to be quite weak compared to Triple DES. However, very small encryption time and large χ^2 values with 255 degrees of freedom (DF) for

all the five .dll files indicate a good performance of MMAT. Moreover, compared to MMAT, Triple DES takes a long time to encrypt a file.

Table 7.2: χ^2 -test for MMAT with .dll files

Sl. No.	Original file	File size (bytes)	MMAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.dll	20480	0.054945	8539	255	06	29790	255
2	2.dll	53312	0.164835	21592	255	16	43835	255
3	3.dll	90176	0.329670	43890	255	26	66128	255
4	4.dll	118784	0.384615	149143	255	34	1211289	255
5	5.dll	204800	0.714286	620375	255	69	2416524	255

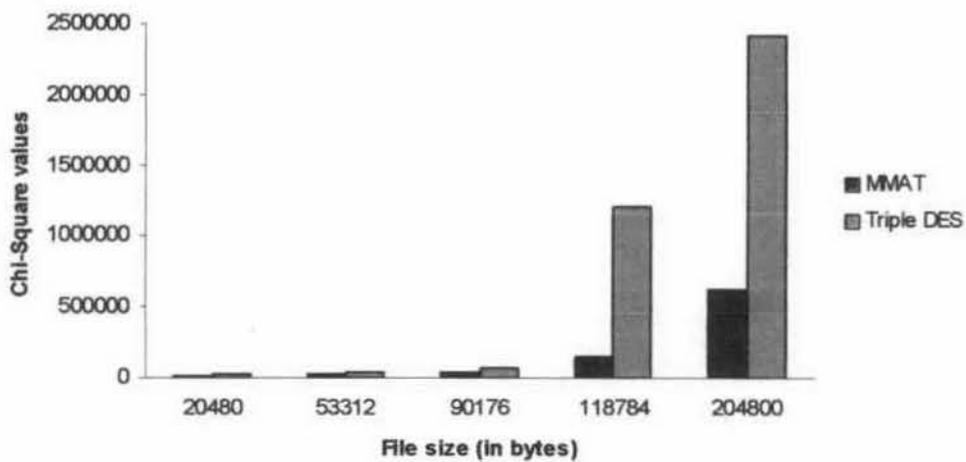


Figure 7.5: MMAT vs. Triple DES in χ^2 -test of .dll files

The results of the test for .exe files are listed in table 7.3 and the same is graphically illustrated by figure 7.6. MMAT shows a better performance in case of .exe files. Besides, for some files, MMAT looks even better than Triple DES. In addition, the encryption times are much less than that of Triple DES.

Table 7.3: χ^2 -test for MMAT with .exe files

Sl. No.	Original file	File size (bytes)	MMAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.exe	23104	0.054945	7633	255	12	8772	255
2	2.exe	52736	0.164835	23766	255	15	43426	255
3	3.exe	131136	0.329670	993854	255	29	986693	255
4	4.exe	170496	0.659341	280797	255	49	475893	255
5	5.exe	200832	0.604396	2748185	255	58	1847377	255

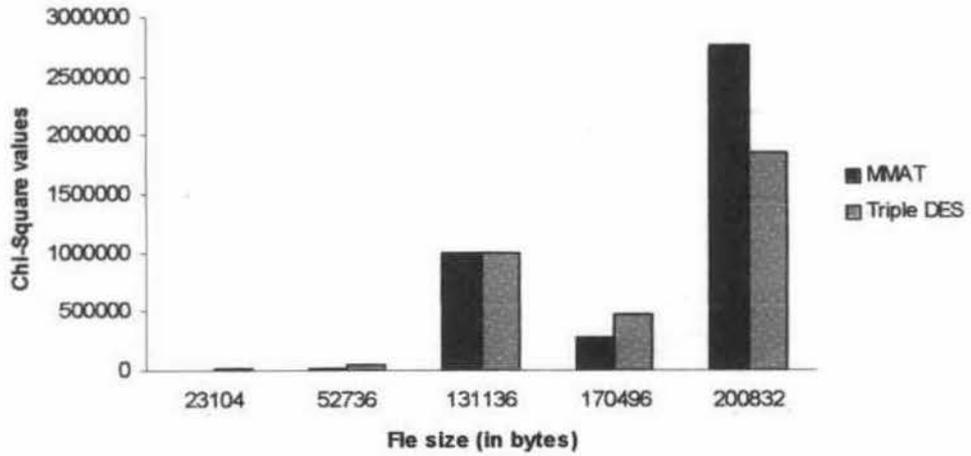


Figure 7.6: MMAT vs. Triple DES in χ^2 -test of .exe files

The results of the test for .jpg files are given in table 7.4 and figure 7.7 gives a graphical view of these results. MMAT shows a very good performance for .jpg files and is almost equal to that of Triple DES. Besides, encryption time for MMAT is very small compared to Triple DES.

Table 7.4: χ^2 -test for MMAT with .jpg files

Sl. No.	Original file	File size (bytes)	MMAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.jpg	28544	0.054945	4313	255	08	4331	255
2	2.jpg	71232	0.274725	2892	255	21	2916	255
3	3.jpg	105600	0.384615	5184	255	31	5227	255
4	4.jpg	160704	0.604396	22284	255	47	22314	255
5	5.jpg	216576	0.769231	29845	255	63	29824	255

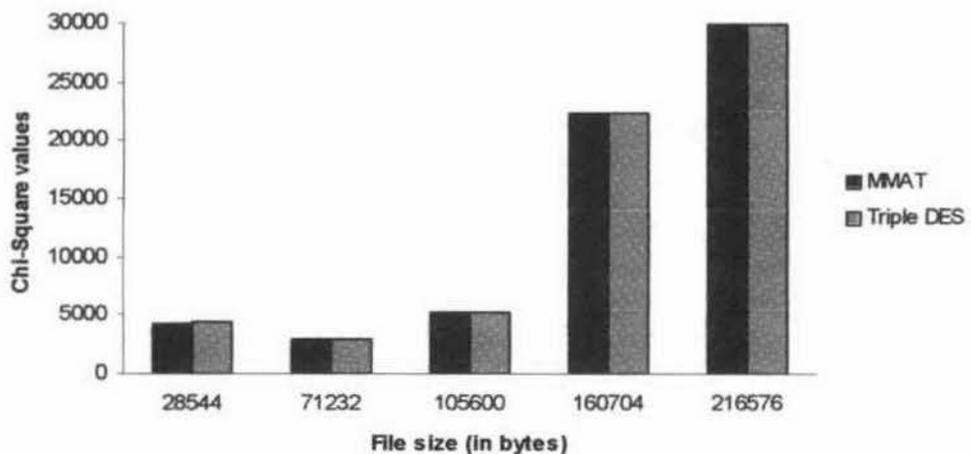


Figure 7.7: MMAT vs. Triple DES in χ^2 -test of .jpg files

The results for .txt files are given by table 7.5 and figure 7.8. The nature shown by MMAT for .txt files is very much similar to that shown for .jpg files. In some cases, the χ^2 values for MMAT are larger than the same for Triple DES with the same degree of freedom (DF).

Table 7.5: χ^2 -test for MMAT with .txt files

Sl. No.	Original file	File size (bytes)	MMAT			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	t1.txt	6976	0.054945	8539	244	02	10629	183
2	t2.txt	23808	0.164835	21592	255	07	32638	255
3	t3.txt	58688	0.329670	43890	255	17	82101	255
4	t4.txt	118784	0.384615	149143	255	35	170557	255
5	t5.txt	190784	0.714286	620375	255	55	430338	255

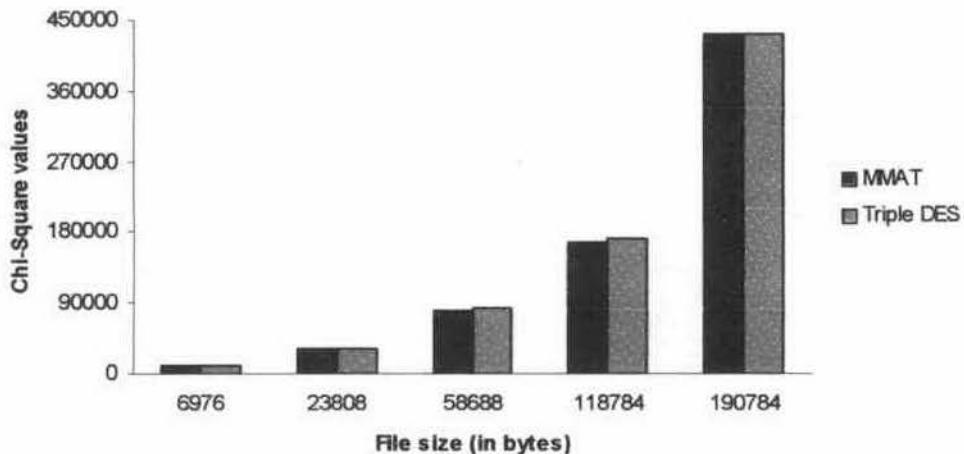


Figure 7.8: MMAT vs. Triple DES in χ^2 -test of .txt files

7.5.3 Avalanche and runs

Like in the previous cases, MMAT was subjected to this test for a 32-bit string. A small change in the plain-text has to create a good amount of effect in the cipher-text to prove the strength of the cipher. Table 7.6 shows the results of this test for MMAT.

Compared to all other previous ciphers, including MAT and OMAT, MMAT has produced the best results with regard to avalanche effect. A great amount of effect on the cipher-text was noticed with a very small change in the plain-text. Moreover,

there was a lot of difference in the first byte of the string being encrypted, which does not change in MAT and OMAT. The fact that there are no common entries under the cipher-text column in the table proves that MMAT causes a great amount of diffusion.

Table 7.6: Avalanche and runs in MMAT

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	17DAECDB	19	19	0
1 ST	C145D450	0EC1E1D8	18	11	7
2 ND	0145D450	1DC8E8DB	17	16	1
3 RD	6145D450	1DBEEED9	19	16	3
4 TH	5145D450	1BBEEDDC	21	16	5
5 TH	4945D450	19BEED5C	21	17	4
6 TH	4545D450	193C6D1B	21	16	5
7 TH	4345D450	98FC2CFB	19	13	6
8 TH	4045D450	D79ACCCB	17	20	3
9 TH	41C5D450	07BAE4DB	17	16	1
10 TH	4105D450	0FBBE809	17	15	2
11 TH	4165D450	1BBCEEDC	19	15	4
12 TH	4155D450	19BB7DDC	21	15	1
13 TH	414DD450	18BB6D5D	19	20	1
14 TH	4141D450	973AAC9C	17	20	3
15 TH	4147D450	57FB0CFB	17	14	3
16 TH	4144D450	F79ADCCB	19	17	1
17 TH	41455450	29C4F4DB	21	18	3
18 TH	41459450	31C8F8D9	19	14	5
19 TH	4145F450	4DDEEEDC	17	15	2
20 TH	4145C450	16BDEBDB	17	18	1
21 ST	4145DE50	193CED5B	17	18	1
22 ND	4145D050	977C6C9C	17	16	1
23 RD	4145D650	58DC2CFB	19	16	3
24 TH	4145D550	37CB0CEB	21	16	5
25 TH	4145D4D0	1FDAF409	19	16	3
26 TH	4145D410	23BBF8DA	17	15	2
27 TH	4145D470	19DCEEDC	17	15	2
28 TH	4145D440	16BDEBDB	17	18	1
29 TH	4145D458	98CB6D5B	19	19	0
30 TH	4145D454	57FB2D1B	21	18	3
31 ST	4145D452	37DB0CFB	21	14	7
32 ND	4145D451	27CAFCEB	20	17	3

7.6 Conclusion

MMAT has shown the best results compared to its siblings and is very much feasible for implementation into the intended targets without losing its credibility. MMAT does not generate any overhead bits in the cipher-text. Its strength may be enhanced by randomizing the order in which the blocks within a pair are chosen for substitution. This order in which the blocks are substituted within a block-pair may form a part of the key, which is discussed in chapter 11.

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 Bit-pair Operation and Separation (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_t, B_{t+1}) will give the bits $C_{(t+1)/2}$ and $C_{(t+1)/2+n/2}$ 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 $C_1, C_2, C_3, \dots, C_{n/2}$ will be called **front bits** and the bits $C_{(n/2)+1}, C_{(n/2)+2}, \dots, C_n$ 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_t, B_{t+1}) is considered for the operations, then the resultant *front bit* will be $C_{(t+1)/2}$ and the *rear bit* will be $C_{(t+1)/2+n/2}$. 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.

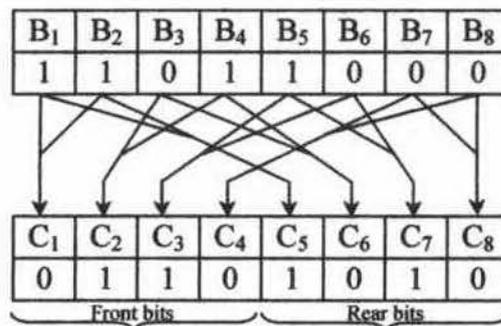


Figure 8.1: Computation of the Front and Rear bits in the first iteration.

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 $3 \cdot 4$, 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.

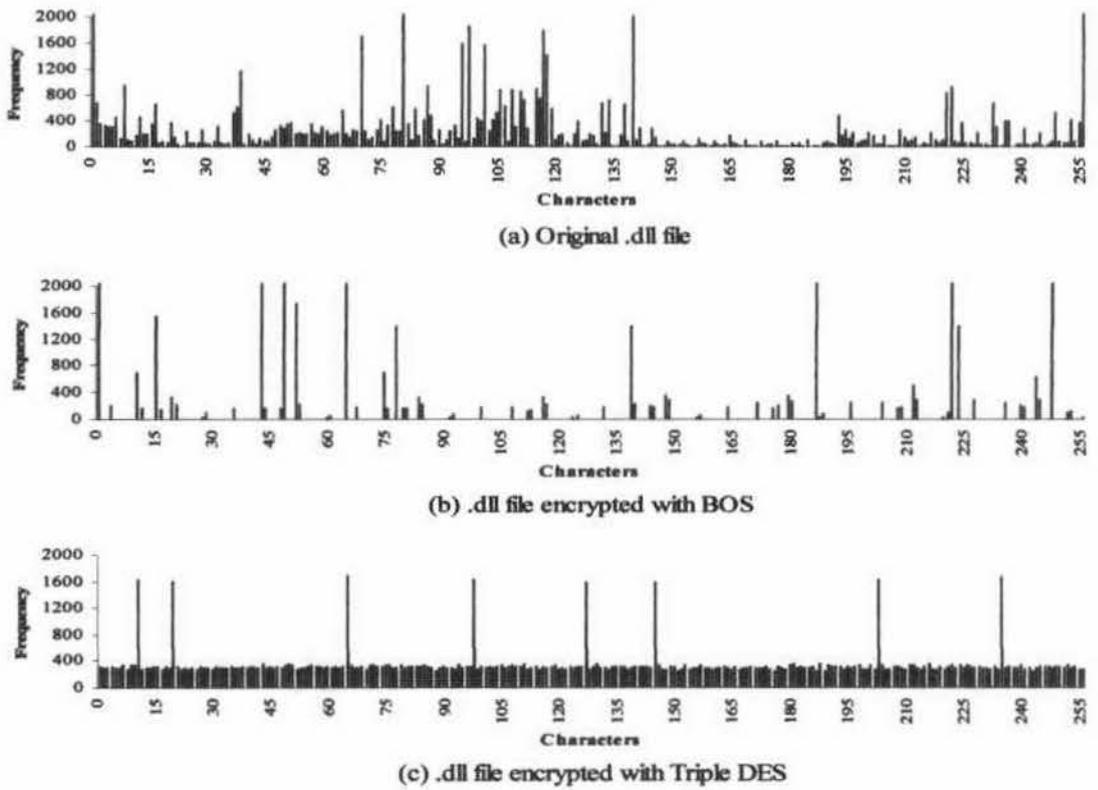


Figure 8.2: Character-frequencies in the source and encrypted .dll files

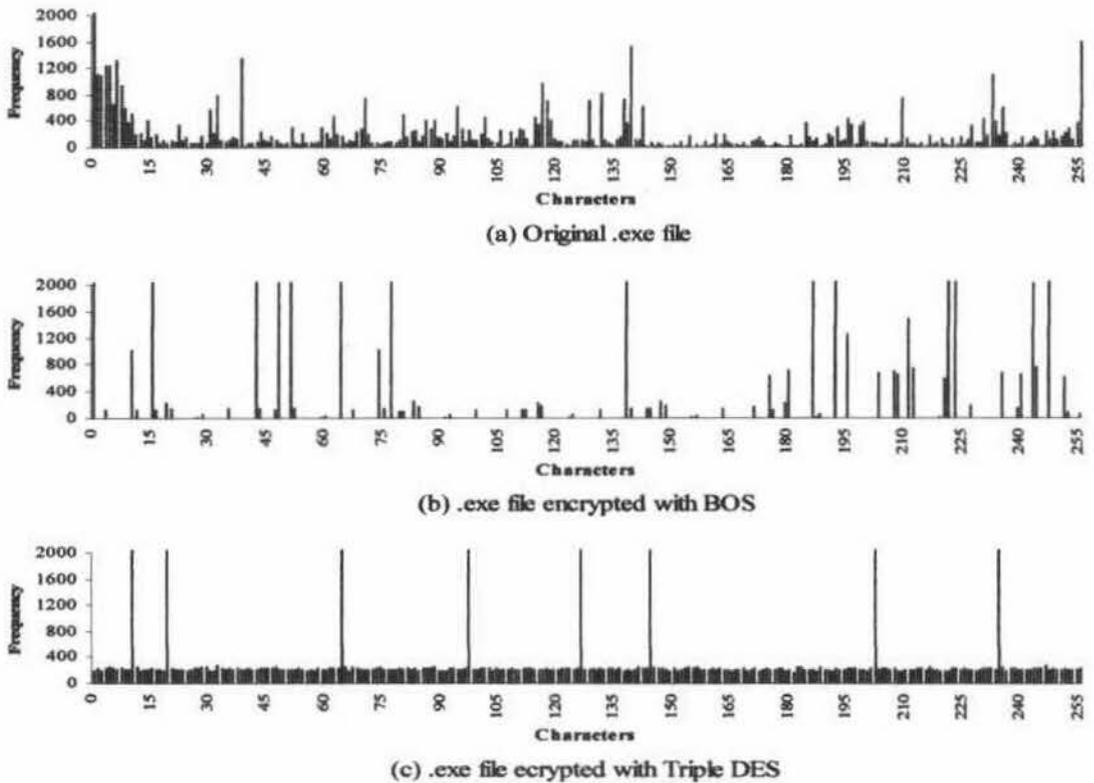


Figure 8.3: Character-frequencies in the source and encrypted .exe files

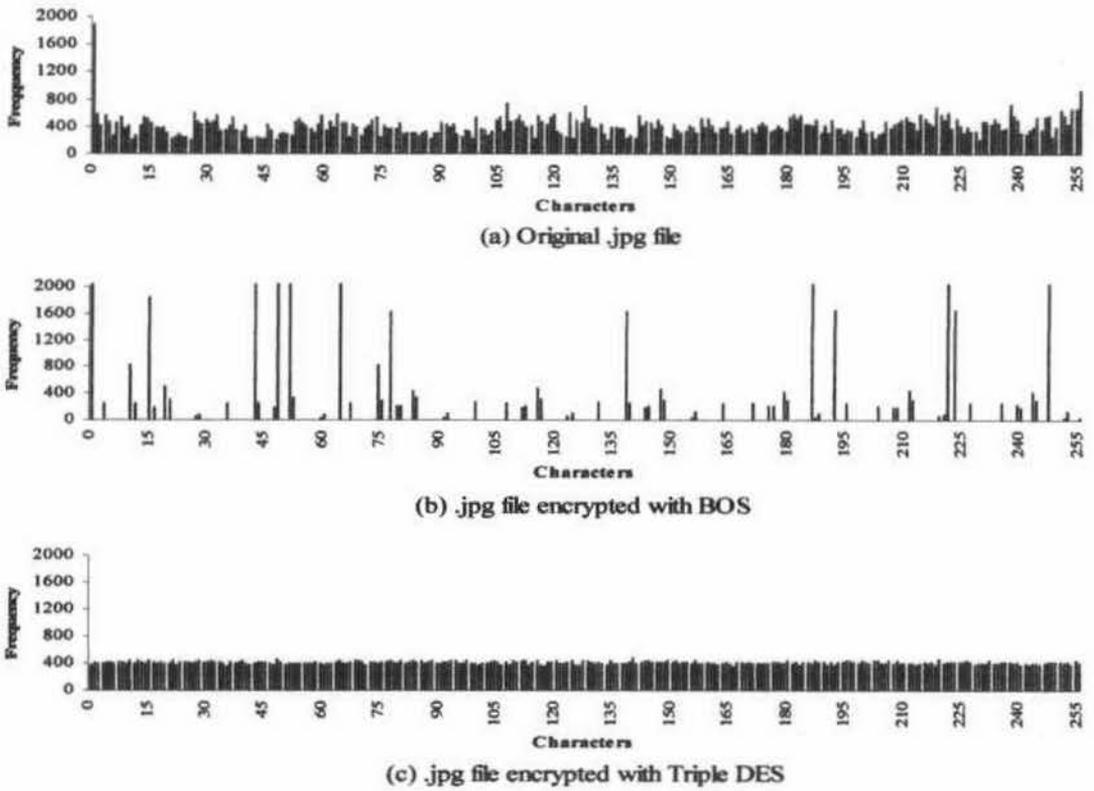


Figure 8.4: Character-frequencies in the source and encrypted .jpg files

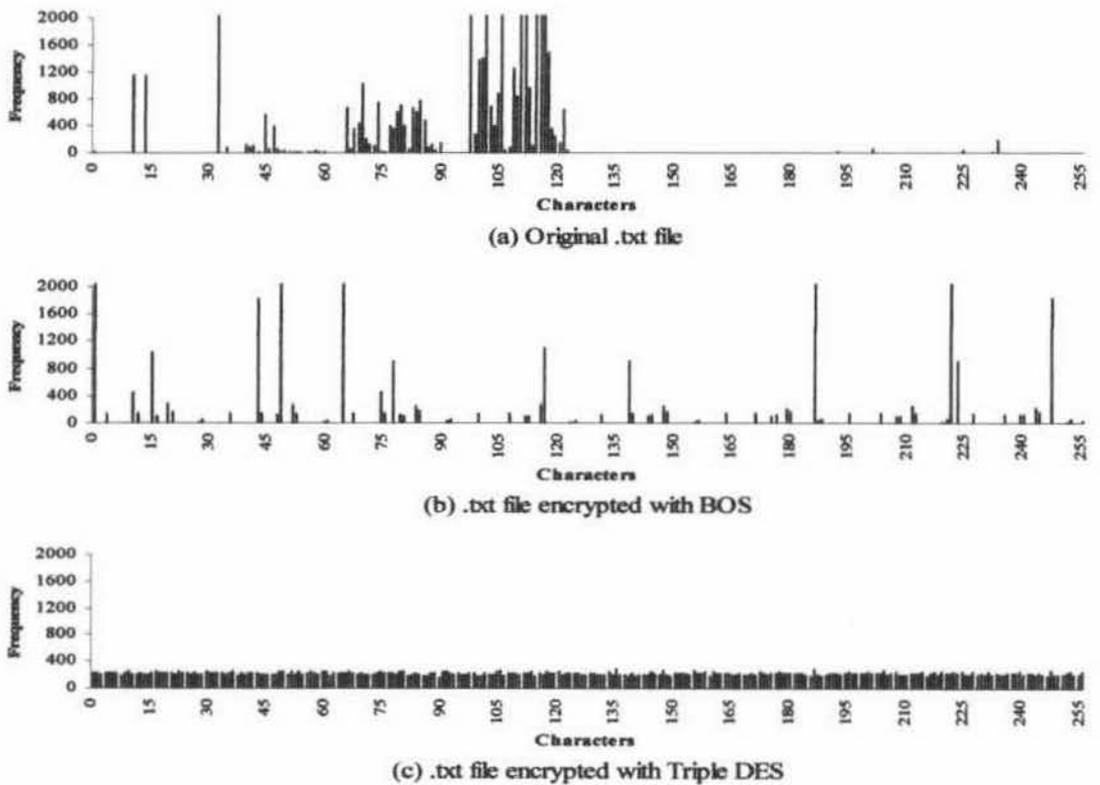


Figure 8.5: Character-frequencies in the source and encrypted .txt files

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

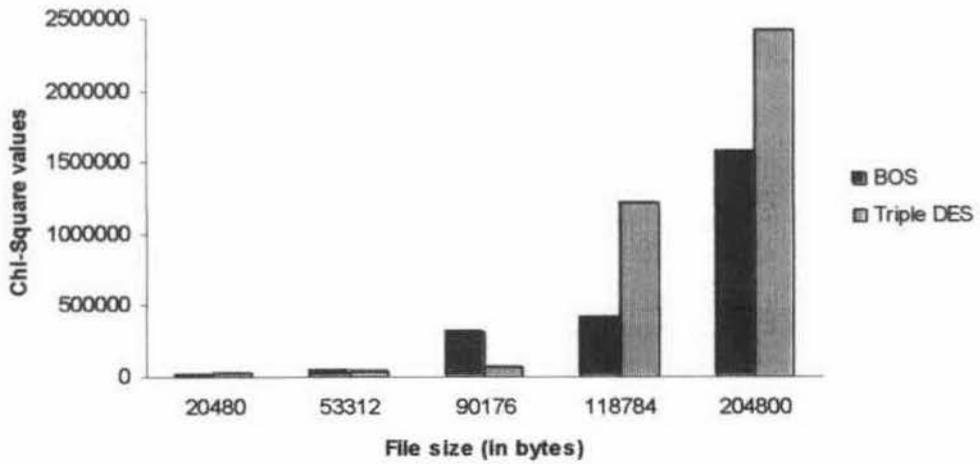


Figure 8.6: BOS vs. Triple DES in χ^2 -test of .dll files

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

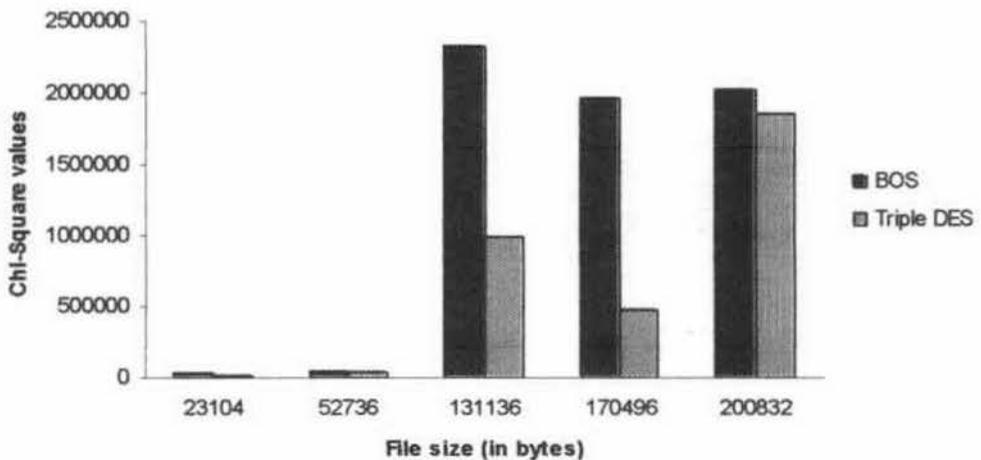


Figure 8.7: BOS vs. Triple DES in χ^2 -test of .exe files

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

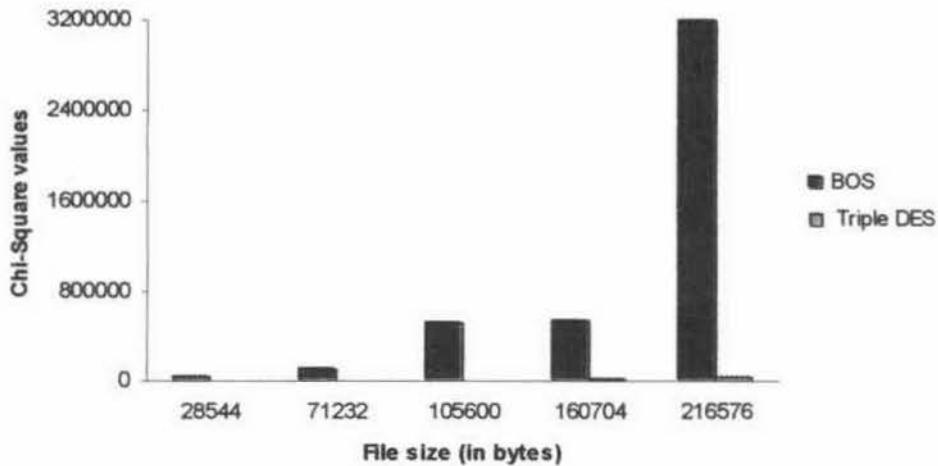


Figure 8.8: BOS vs. Triple DES in χ^2 -test of .jpg files

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

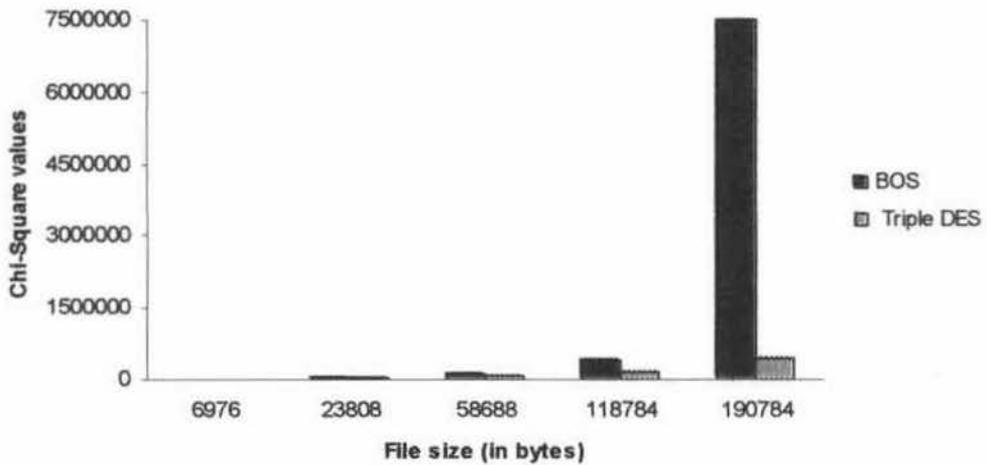


Figure 8.9: BOS vs. Triple DES in χ^2 -test of .txt files

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
1 ST	C145D450	E44E66C6	18	16	2
2 ND	0145D450	A04E22C6	17	16	1
3 RD	6145D450	E84E6AC6	19	18	1
4 TH	5145D450	A84E2AC6	21	20	1
5 TH	4945D450	E44EAA66	21	20	1
6 TH	4545D450	A04EAA66	21	20	1
7 TH	4345D450	E84EAA66	19	20	1
8 TH	4045D450	A84EAA66	17	22	5
9 TH	41C5D450	2C4EAA66	17	21	4
10 TH	4105D450	0A4E88C6	17	17	0
11 TH	4165D450	284EAA66	19	21	2
12 TH	4155D450	084E8AC6	21	17	4
13 TH	414DD450	2C4EAA66	19	21	2
14 TH	4141D450	0A4EAA66	17	21	4
15 TH	4147D450	284EAA66	17	21	4
16 TH	4144D450	084EAA66	19	21	2
17 TH	41455450	2882AA0A	21	21	0
18 TH	41459450	28C6AA4E	19	21	2
19 TH	4145F450	288EAA06	17	19	2
20 TH	4145C450	28CEAA46	17	21	4
21 ST	4145DE50	2882AAC6	17	21	4
22 ND	4145D050	28C6AAC6	17	21	4
23 RD	4145D650	288EAA66	19	21	2
24 TH	4145D550	28CEAA66	21	21	0
25 TH	4145D4D0	288EAAA6	19	23	4
26 TH	4145D410	286CAA6E	17	21	4
27 TH	4145D470	288EAAA6	17	23	6
28 TH	4145D440	286EAA6E	17	21	4
29 TH	4145D458	288EAA66	19	21	2
30 TH	4145D454	286CAA66	21	21	0
31 ST	4145D452	288EAA66	21	21	0
32 ND	4145D451	286EAA66	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.

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 **Decimal Equivalent Positional Substitution (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 $S = s_0 s_1 s_2 s_3 s_4 \dots s_{L-1}$ of length L bits, the scheme is followed in a stepwise manner to generate the target block $T = t_0 t_1 t_2 t_3 t_4 \dots t_{L-1}$ 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.

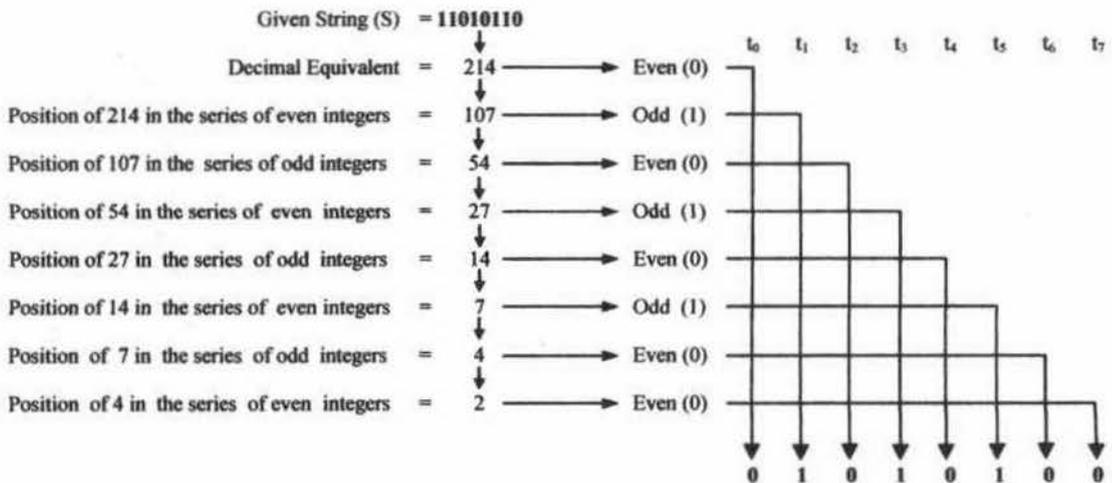


Figure 9.1: Cipher-text generation using DEPS on 8-bit data.

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 $t_2=0$. 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, $S = s_0 s_1 s_2 s_3 s_4 \dots s_{L-1}$, of length L , the decimal equivalent D_L is computed and the target block, $T = t_0 t_1 t_2 t_3 t_4 \dots t_{L-1}$, 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  $TEMP = \text{remainder of } D_{L-P} / 2$ 
    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, $T = t_0 t_1 t_2 t_3 t_4 \dots t_{L-1}$ of length L bits, the decryption algorithm is carried out to generate $S = s_0 s_1 s_2 s_3 s_4 \dots s_{L-1}$, 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  $T = T^{\text{th}}$  number in the series of even integers
    else
        compute  $T = T^{\text{th}}$  number in the series of odd integers
    end if
    set  $P = P - 1$ 
    if  $P \geq 0$  then goto LOOP
end if
compute  $S = s_0 s_1 s_2 s_3 s_4 \dots s_{L-1}$ , 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,

$$\begin{aligned} S_1 &= 01001100011011110110001101100001, \\ S_2 &= 01101100001000000100000101110010, \\ S_3 &= 01100101011000010010000001001110, \\ S_4 &= 01100101011101000111011101101111, \text{ and} \\ S_5 &= 0111001001101011 \end{aligned}$$

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

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

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

$$T_2 = 0111000101111101111101111001001,$$

$$T_3 = 0111000101111101111101111001001,$$

$$T_4 = 10001001000100011101000101011001, \text{ and}$$

$$T_5 = 1110100110110001$$

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}√fM√yYè▶⇐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

$$T_1 = 11010100011110000110010111000000,$$

$$T_2 = 0111000101111101111101111001001,$$

$$T_3 = 0111000101111101111101111001001,$$

$$T_4 = 10001001000100011101000101011001,$$

$$T_5 = 1110100110110001$$

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 $C_y = 0$ then jump to step 14
- Step 10 : Complement C_y
- 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

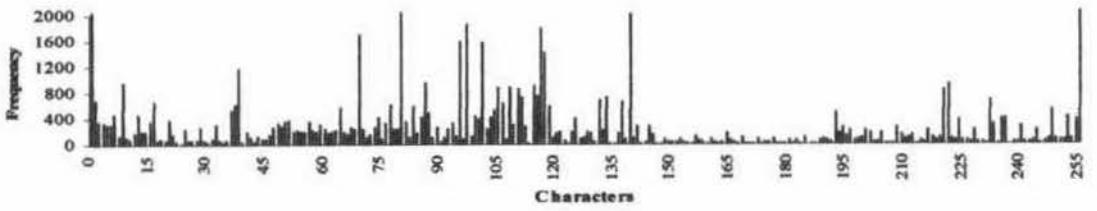
9.5 Results and comparisons

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.

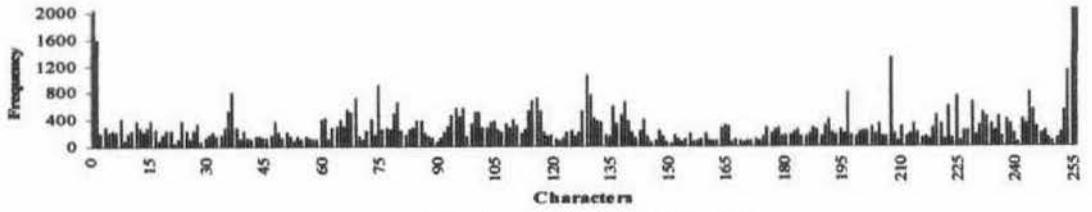
The same set of twenty files, and more precisely, five different files of varying sizes in each of the four categories, namely .dll, .exe, .jpg and .txt, were considered for the purpose of testing and were encrypted using the DEPS algorithm. The results of the tests have been compared with those of Triple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

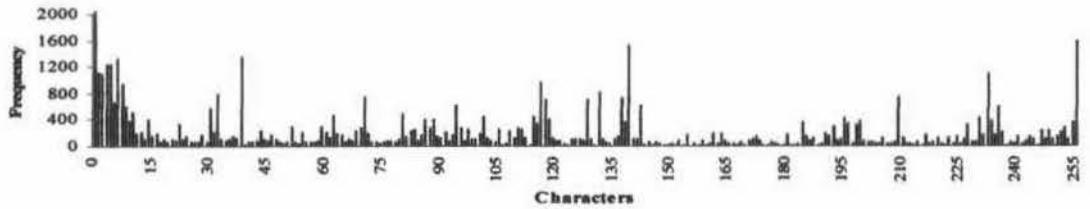


(b) .dll file encrypted with DEPS

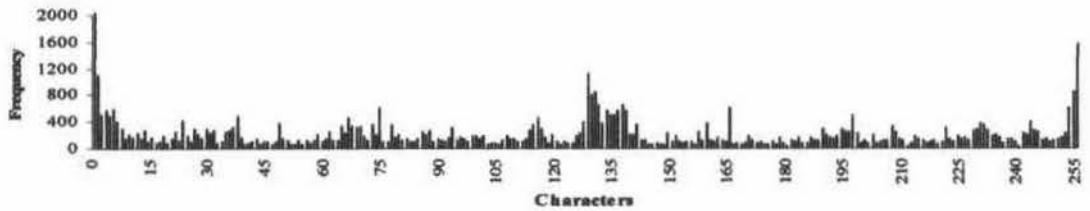


(c) .dll file encrypted with Triple DES

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 encrypted with Triple DES

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

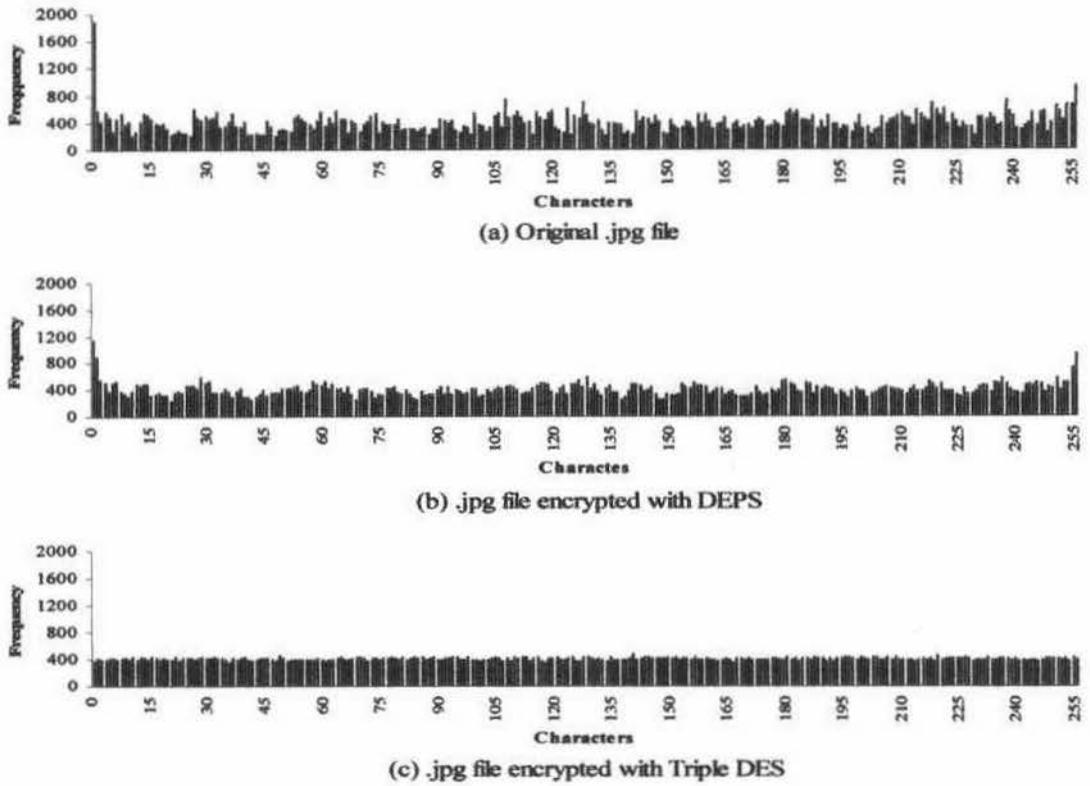


Figure 9.4: Character-frequencies in the source and encrypted .jpg files

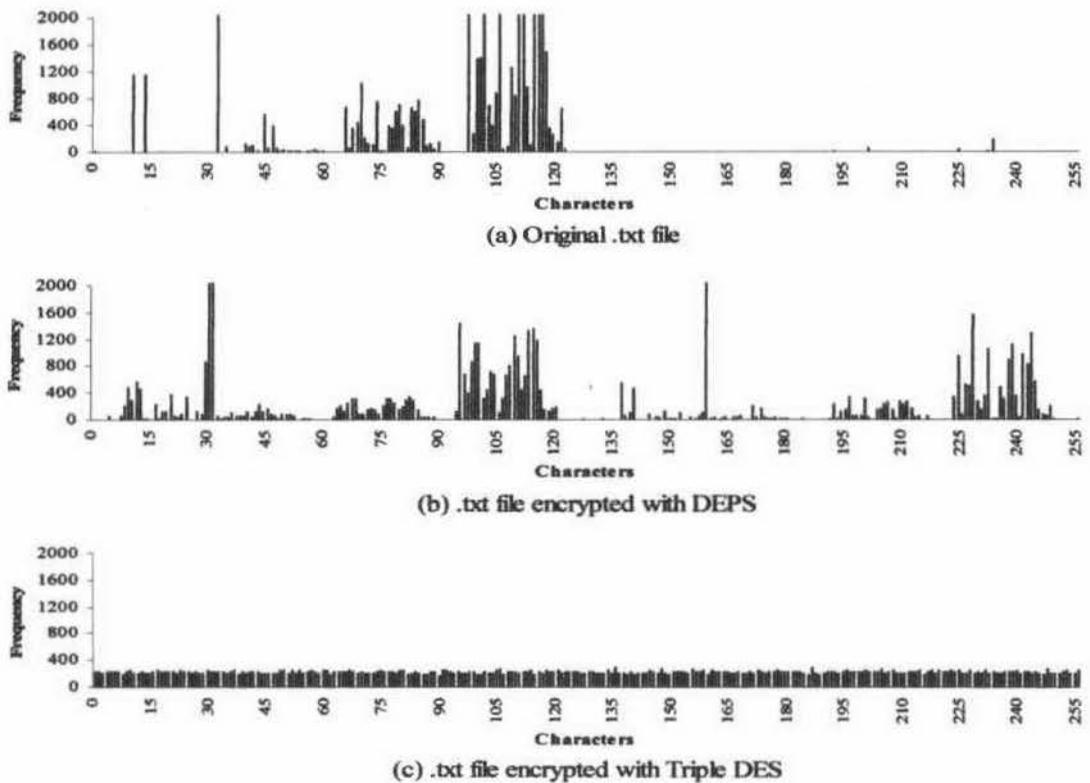


Figure 9.5: Character-frequencies in the source and encrypted .txt files

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.

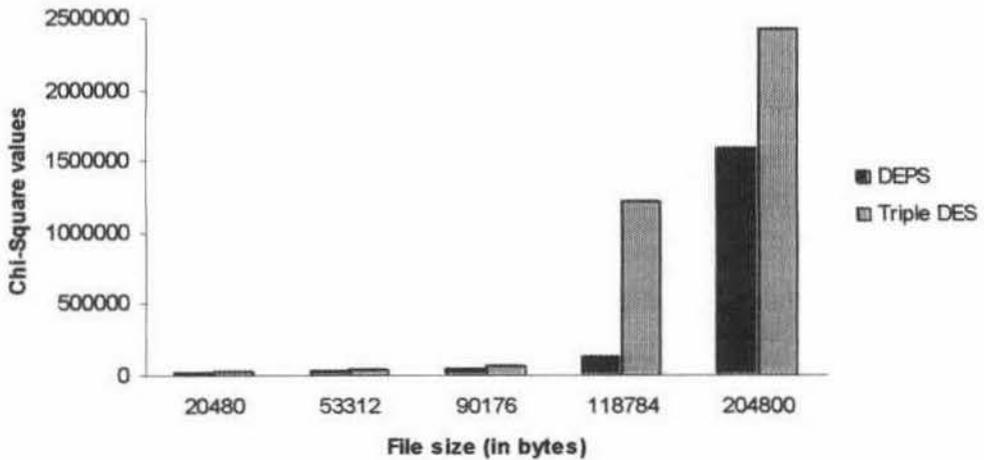
DEPS has not shown a good performance for .txt file compared to other proposed algorithms of this thesis. However, as in the case of transposition ciphers, this deficiency can be overcome by cascading with another cipher. Since DEPS is a substitution cipher, it can be cascaded with a transposition cipher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

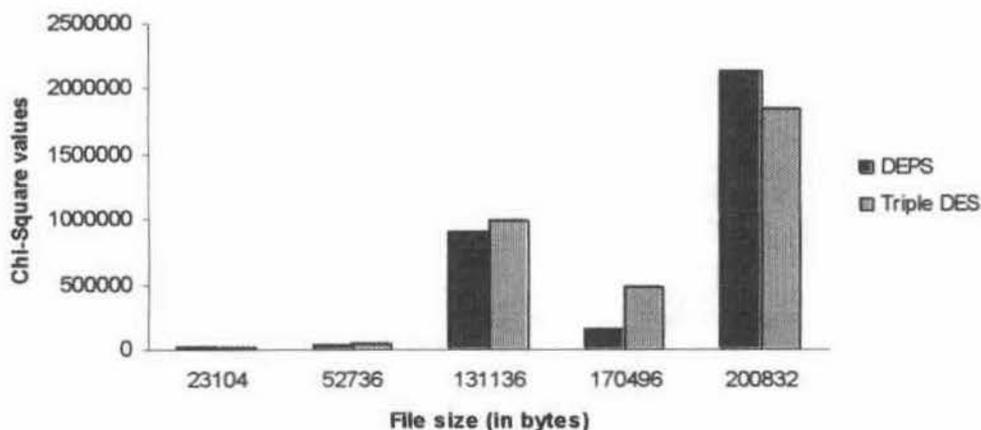


Figure 9.7: DEPS vs. Triple DES in χ^2 -test of .exe files

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

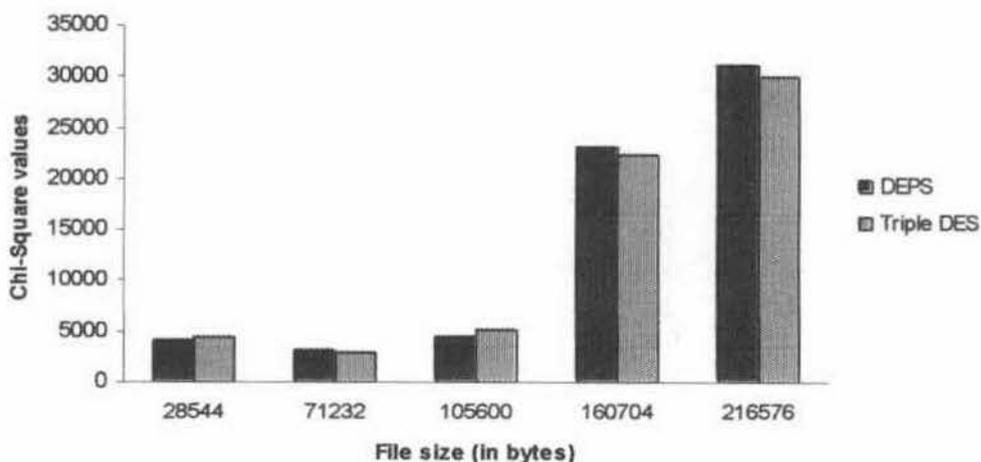


Figure 9.8: DEPS vs. Triple DES in χ^2 -test of .jpg files

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

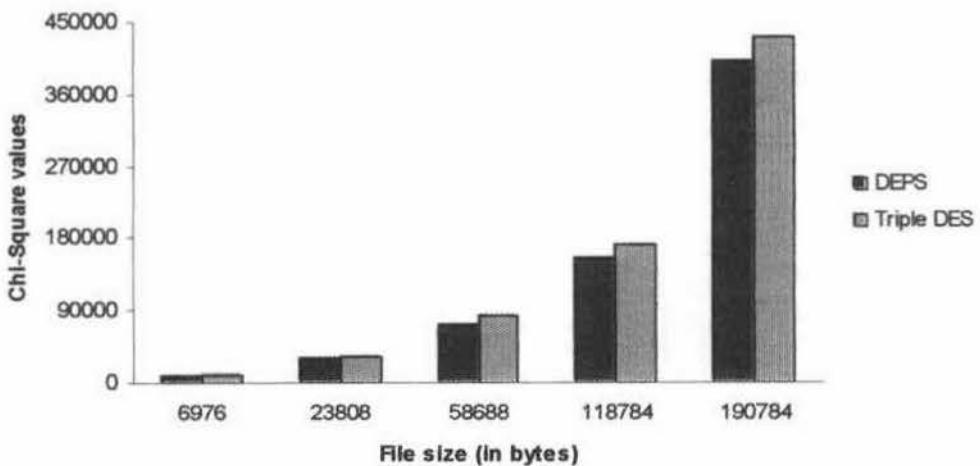


Figure 9.9: DEPS vs. Triple DES in χ^2 -test of .txt files

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
1 ST	C145D450	FCDD340D	18	15	3
2 ND	0145D450	FFDD340D	17	14	3
3 RD	6145D450	F9DD340D	19	15	4
4 TH	5145D450	F5DD340D	21	17	4
5 TH	4945D450	EDDDC40D	21	17	4
6 TH	4545D450	DDDD340D	21	17	4
7 TH	4345D450	DDDD340D	19	17	2
8 TH	4045D450	03DD340D	17	14	3
9 TH	41C5D450	FDDC340D	17	13	4
10 TH	4105D450	FDDF340D	17	13	4
11 TH	4165D450	FDD9340D	19	15	4
12 TH	4155D450	FDD5340D	21	17	4
13 TH	414DD450	FDCD340D	19	15	4
14 TH	4141D450	FDFD340D	17	13	4
15 TH	4147D450	FD9D340D	17	15	2
16 TH	4144D450	FD3D340D	19	15	4
17 TH	41455450	FDDD350D	21	17	4
18 TH	41459450	FDDD360D	19	15	4
19 TH	4145F450	FDDD300D	17	14	3
20 TH	4145C450	FDDD3D0D	17	13	4
21 ST	4145DE50	FDDD240D	17	15	2
22 ND	4145D050	FDDD0C0D	17	13	4
23 RD	4145D650	FDDD540D	19	17	2
24 TH	4145D550	FDDDD40D	21	15	6
25 TH	4145D4D0	FDDD340F	19	15	4
26 TH	4145D410	FDDD3409	17	13	4
27 TH	4145D470	FDDD3403	17	15	2
28 TH	4145D440	FDDD3415	17	13	4
29 TH	4145D458	FDDD3435	19	17	2
30 TH	4145D454	FDDD3435	21	17	4
31 ST	4145D452	FDDD3475	21	17	4
32 ND	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.

Cascaded Techniques: Product Ciphers

10.1 Introduction

Some of the proposed transposition ciphers have not shown very good results. This issue has already been discussed in the respective chapters. Since a transposition cipher is involved in permutation of bits only, its strength may not be at par with a substitution cipher. If a substitution cipher is combined with a transposition cipher in a cascaded manner, the strength of the combination will be much better than any of the constituent ciphers performing independently. Theoretically, such a combination is called a *product cipher*.

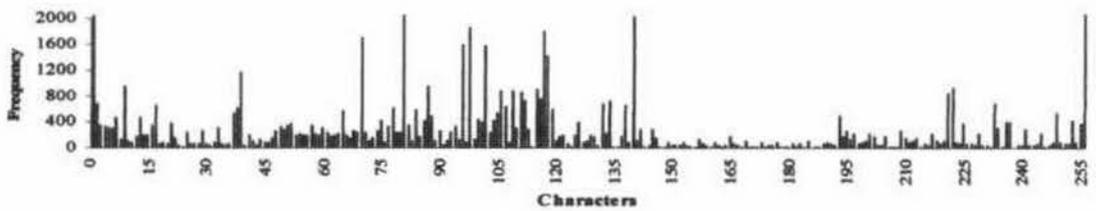
The transposition ciphers that have been proposed in this thesis are LSPB, RSPB, DSPB, CSPB, BET and SPOB. On the other hand, the proposed substitution ciphers are MAT, OMAT, MMAT, BOS and DEPS. Although any transposition cipher may be clubbed with any substitution cipher, it is not possible to discuss all possible combinations at this instance. Thus only two such cascaded techniques are being discussed in this chapter.

10.2 OMAT and BET

OMAT and BET has already been discussed individually. In this product cipher the cipher-text generated by OMAT is fed as a plain-text into BET to get the final cipher-text. Since the programs are run one after the other, the algorithms or routines are not listed here to avoid repetitions. The prime importance has been given to the performance of the combination only. The same set of tests has been performed to determine the strength of this combination.

10.2.1 Character frequency

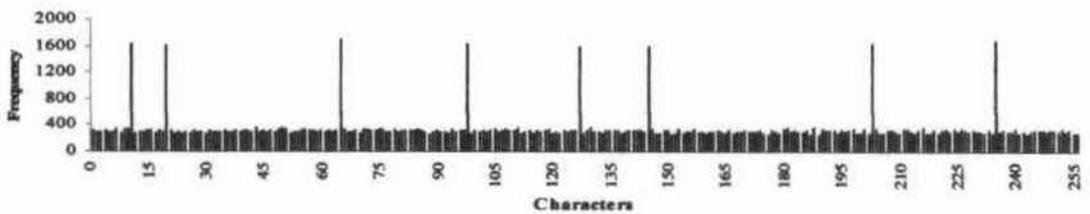
The results of the test for .dll, .exe, .jpg and .txt files are separately illustrated in the figures 10.1 through 10.4, respectively. The graphs for the original file and one encrypted with Triple DES have also been considered for comparison.



(a) Original .dll file

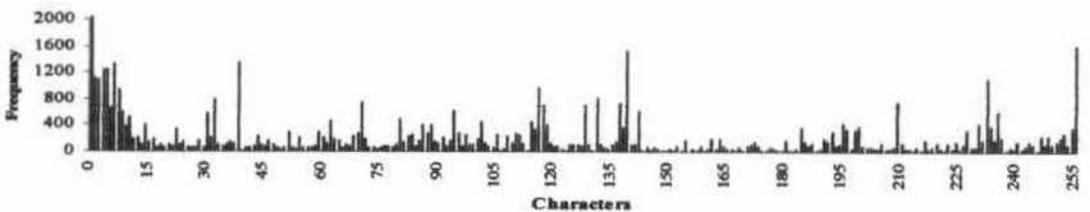


(b) .dll file encrypted with O MAT+BET

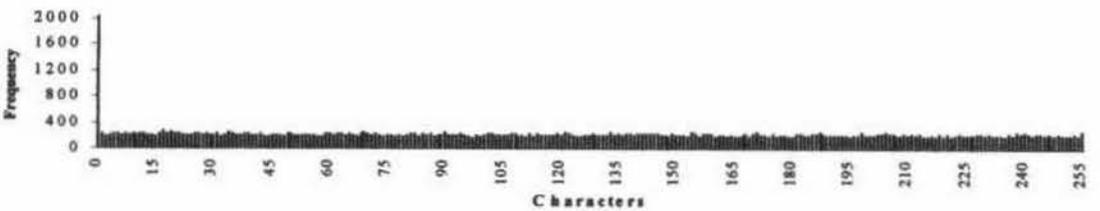


(c) .dll file encrypted with Triple DES

Figure 10.1: Character-frequencies in the source and encrypted .dll files



(a) Original .exe file

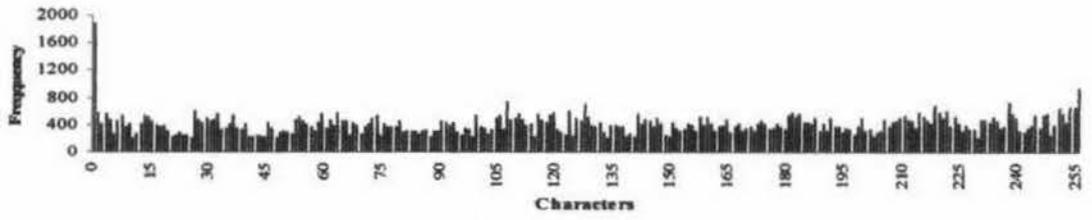


(b) .exe file encrypted with O MAT+BET

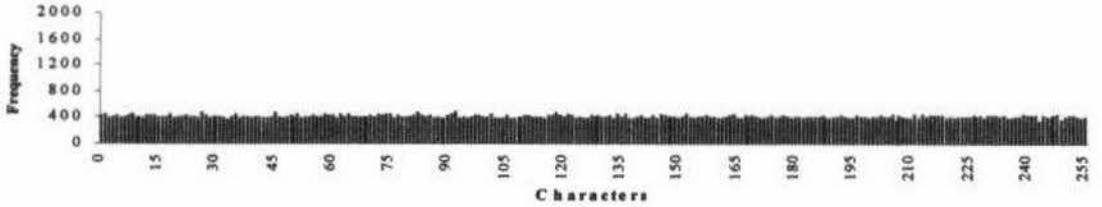


(c) .exe file encrypted with Triple DES

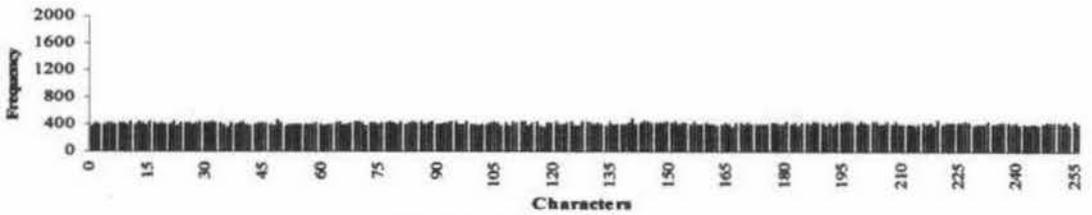
Figure 10.2: Character-frequencies in the source and encrypted .exe files



(a) Original .jpg file

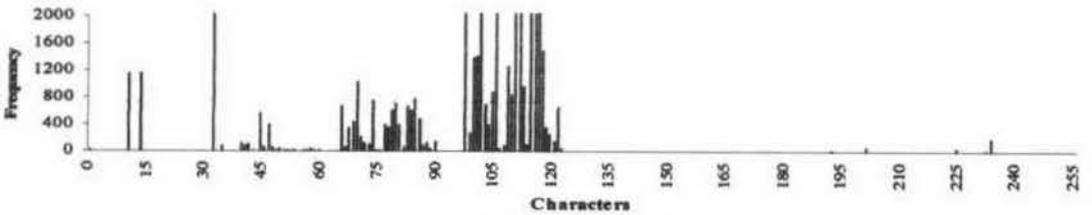


(b) .jpg file encrypted with O MAT+BET

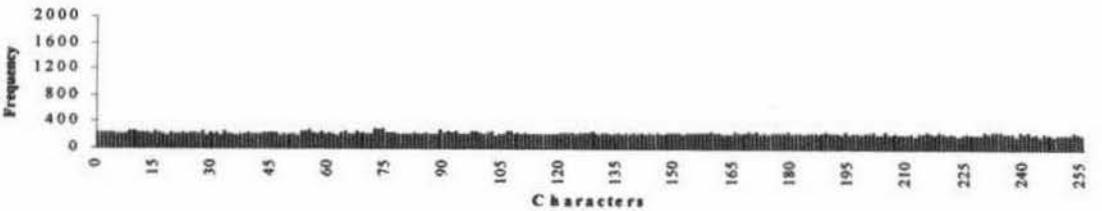


(c) .jpg file encrypted with Triple DES

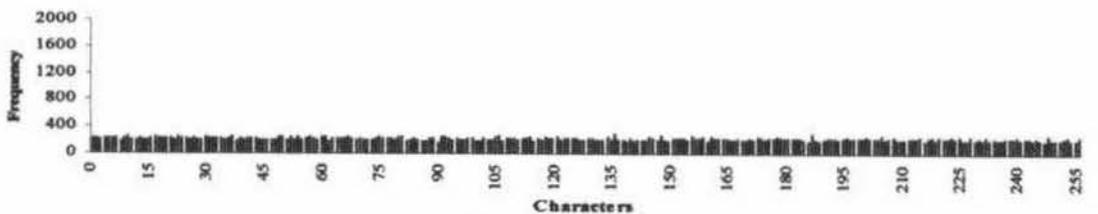
Figure 10.3: Character-frequencies in the source and encrypted .jpg files



(a) Original .txt file



(b) .txt file encrypted with O MAT+BET



(c) .txt file encrypted with Triple DES

Figure 10.4: Character-frequencies in the source and encrypted .txt files

It is quite evident from the graph that in case of .dll and .exe files, OMAT+BET has shown better performance than any of OMAT or BET taken individually. The characters have been evenly distributed over the 0-255 space. Moreover, in case of Triple DES, there are some characters with abruptly high frequencies.

The performances of this cascaded combination for .jpg and .txt files are more or less same. In no way they are poorer than the performance of Triple DES.

10.2.2 Chi-Square test and encryption time

The χ^2 test and encryption times due to OMAT+BET have been compared with those of Triple DES, each category of files being dealt with separately. The comparative χ^2 values and encryption times for OMAT+BET along with those for Triple DES in case of .dll files are given by table 10.1 and figure 10.5.

Table 10.1: χ^2 -test for OMAT+BET with .dll files

Sl. No.	Original file	File size (bytes)	OMAT+BET			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.dll	20480	0.10989	7827	255	06	29790	255
2	2.dll	53312	0.32967	21078	255	16	43835	255
3	3.dll	90176	0.54945	43415	255	26	66128	255
4	4.dll	118784	0.76923	142618	255	34	1211289	255
5	5.dll	204800	1.26374	563669	255	69	2416524	255

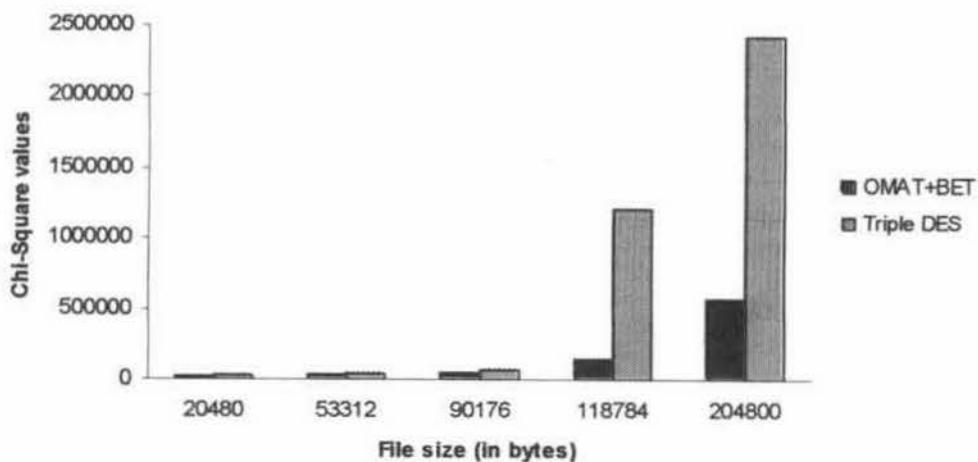


Figure 10.5: OMAT+BET vs. Triple DES in χ^2 -test of .dll files

The performance of OMAT+BET in χ^2 -test with .dll files is quite satisfactory, although not as good as that of Triple DES. Very small encryption time and large χ^2 values indicate the strength of OMAT+BET. The results of the test for .exe files are given by table 10.2 and figure 10.6.

Table 10.2: χ^2 -test for OMAT+BET with .exe files

Sl. No.	Original file	File size (bytes)	OMAT+BET			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.exe	23104	0.10989	7518	255	12	8772	255
2	2.exe	52736	0.32967	22893	255	15	43426	255
3	3.exe	131136	0.76923	986123	255	29	986693	255
4	4.exe	170496	1.04396	256526	255	49	475893	255
5	5.exe	200832	1.15385	2696344	255	58	1847377	255

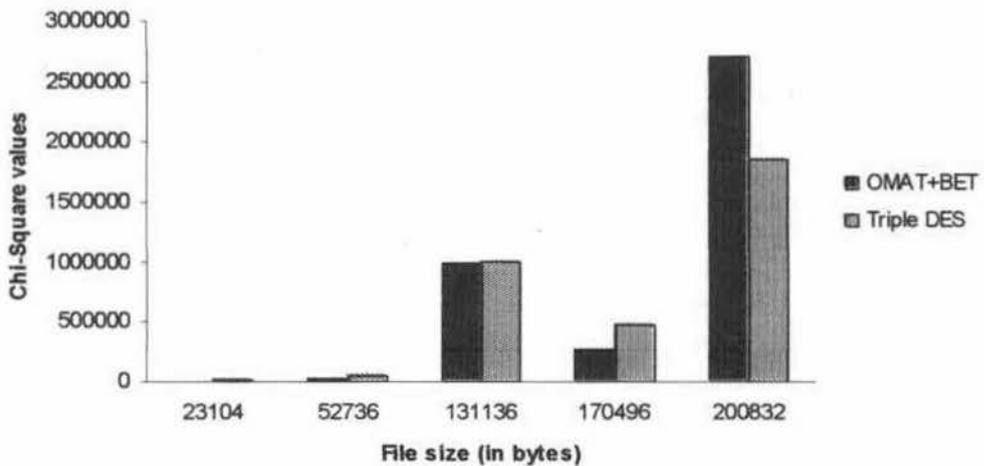


Figure 10.6: OMAT+BET vs. Triple DES in χ^2 -test of .exe files

In case of .exe files, the χ^2 values for OMAT+BET quite large and at par with Triple DES. Moreover, in one case, the χ^2 value is much larger and the encryption times are much less than Triple DES. The degree of freedom (DF) is 255 for all the files. The results for .jpg files are listed in table 10.3 and illustrated by figure 10.7.

Table 10.3: χ^2 -test for OMAT+BET with .jpg files

Sl. No.	Original file	File size (bytes)	OMAT+BET			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.jpg	28544	0.10989	4489	255	08	4331	255
2	2.jpg	71232	0.43956	2847	255	21	2916	255
3	3.jpg	105600	0.65934	5032	255	31	5227	255
4	4.jpg	160704	0.93407	22045	255	47	22314	255
5	5.jpg	216576	1.31869	29190	255	63	29824	255

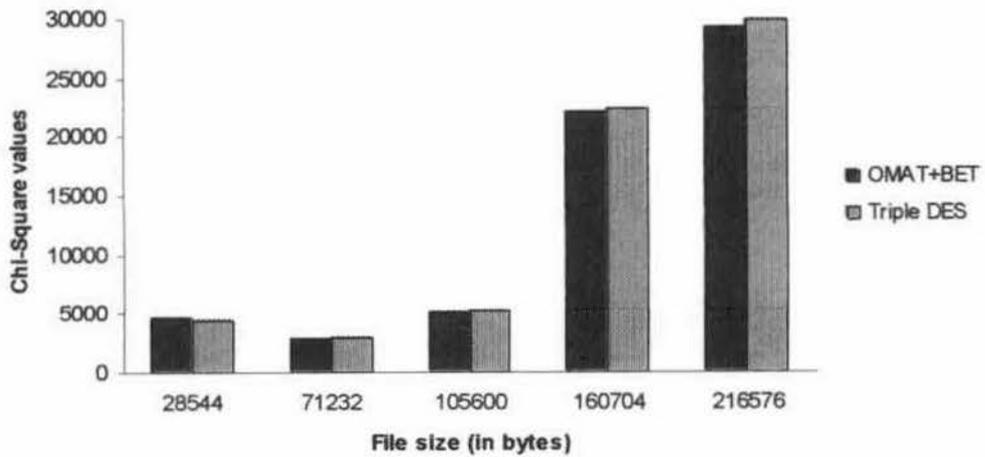


Figure 10.7: OMAT+BET vs. Triple DES in χ^2 -test of .jpg files

The nature shown by OMAT+BET in case of .jpg files is almost like that of Triple DES. High χ^2 values, all with 255 degree of freedom (DF), along with very small encryption times depict a good performance of OMAT+BET. The results for .txt files are given by table 10.4 and figure 10.8.

Table 10.4: χ^2 -test for OMAT+BET with .txt files

Sl. No.	Original file	File size (bytes)	OMAT+BET			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	t1.txt	6976	0.054946	10423	190	02	10629	183
2	t2.txt	23808	0.164835	32829	255	07	32638	255
3	t3.txt	58688	0.384615	81725	255	17	82101	255
4	t4.txt	118784	0.769230	166308	255	35	170557	255
5	t5.txt	190784	1.153847	434533	255	55	430338	255

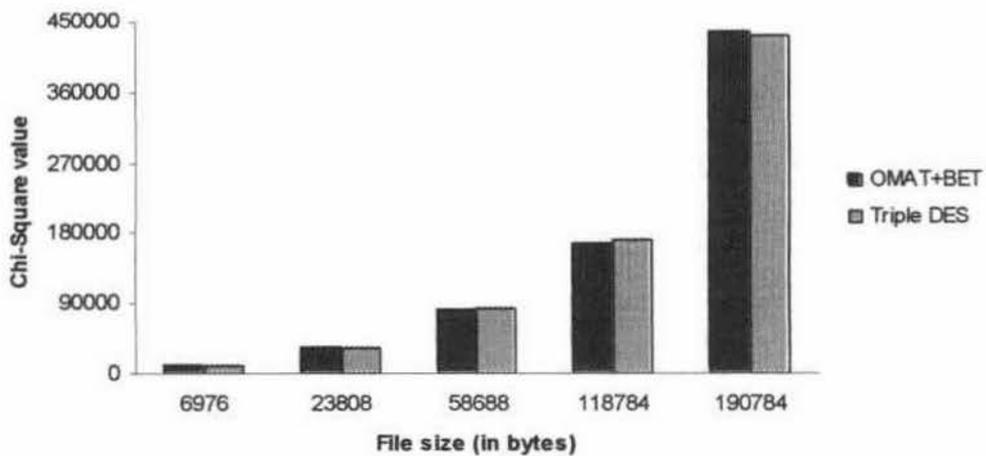


Figure 10.8: OMAT+BET vs. Triple DES in χ^2 -test of .txt files

OMAT+BET has shown a very good performance in the χ^2 -test for .txt files. The χ^2 values for all .txt files in OMAT+BET are almost same as Triple DES, and even higher in two cases.

10.2.3 Avalanche and runs

Table 10.5 lists the results of avalanche and runs test for OMAT+BET. As usual, this test has been carried out to examine the randomness and diffusion property of OMAT+BET, i.e. to test how much effect can be created in the cipher-text with a very small change in the plain-text.

Table 10.5: Avalanche and runs in OMAT+BET

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	540D83A0	19	14	5
1 ST	C145D450	D409B320	18	15	3
2 ND	0145D450	14C93160	17	16	1
3 RD	6145D450	564DC3A2	19	18	1
4 TH	5145D450	554F81A1	21	17	4
5 TH	4945D450	5C4E8048	21	16	5
6 TH	4545D450	58068BA4	21	16	5
7 TH	4345D450	78028780	19	18	1
8 TH	4045D450	4429AE90	17	18	1
9 TH	41C5D450	54093320	17	16	1
10 TH	4105D450	54897360	17	18	1
11 TH	4165D450	564F83A0	19	14	5
12 TH	4155D450	554E81A0	21	18	3
13 TH	414DD450	5C4680A0	19	14	5
14 TH	4141D450	50098AA0	17	16	1
15 TH	4147D450	580DA780	17	12	5
16 TH	4144D450	5429BE90	19	18	1
17 TH	41455450	508933B0	21	16	5
18 TH	41459450	50C973B0	19	16	3
19 TH	4145F450	504F81B0	17	12	5
20 TH	4145C450	504C80B0	17	14	3
21 ST	4145DE50	50068BB0	17	14	3
22 ND	4145D050	50098EB0	17	14	3
23 RD	4145D650	542DA3A0	19	18	1
24 TH	4145D550	542D93A0	21	18	3
25 TH	4145D4D0	5489B3A0	19	18	1
26 TH	4145D410	54C9B3A0	17	18	1
27 TH	4145D470	544DB1A0	17	16	1
28 TH	4145D440	544D80A0	17	16	1
29 TH	4145D458	540D8BA0	19	16	3
30 TH	4145D454	540D87A0	21	14	7
31 ST	4145D452	542D83A0	21	16	5
32 ND	4145D451	541D83A0	20	14	6

The table shows that none of the consecutive cipher-texts are same. This proves that OMAT+BET can produce a good amount of effect in the cipher-text with a very small change in the plain-text. The randomness property of OMAT+BET is also proved by a high average difference of runs between the plain-text and the corresponding cipher-text.

10.3 MMAT and DSPB

Another interesting combination of a substitution and a transposition cipher would be MMAT and DSPB. This product cipher generates the cipher-text by feeding the cipher-text generated by MMAT as a plain-text into DSPB. As in OMAT+BET, the algorithms or routines are not listed here to avoid repetitions. The same tests have been performed to determine the strength of this combination.

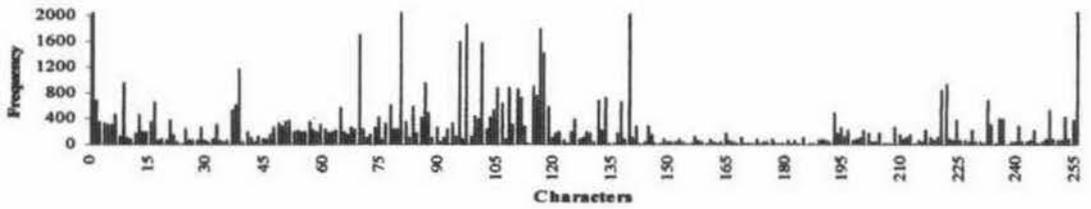
10.3.1 Character frequency

Figures 10.9 through 10.12 illustrate the results of the character frequency tests for .dll, .exe, .jpg and .txt files, respectively. Each category of file is treated separately. As in the case of other proposed algorithms, the result of only one file in each category has been considered. The graphs for the original files and the ones encrypted with Triple DES have also been considered for comparison.

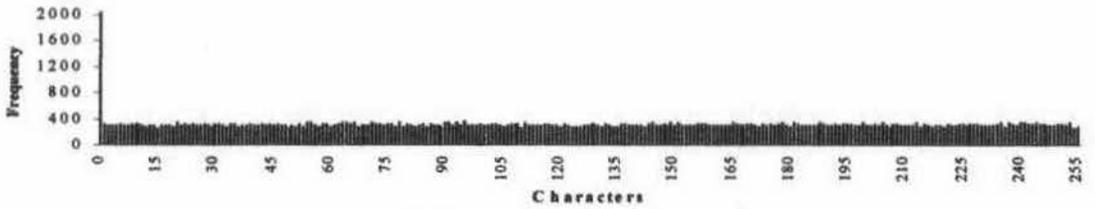
MMAT+DSPB has also demonstrated a similar nature as OMAT+BET in case of .dll and .exe files. The performance of the combination is better than any of MMAT or DSPB considered separately. The characters have been more evenly distributed over the 0-255 space than Triple DES, where some characters have abruptly high frequencies. The performances of this cascaded combination for .jpg and .txt files are more or less same and are at with those of Triple DES.

10.3.2 Chi-Square test and encryption time

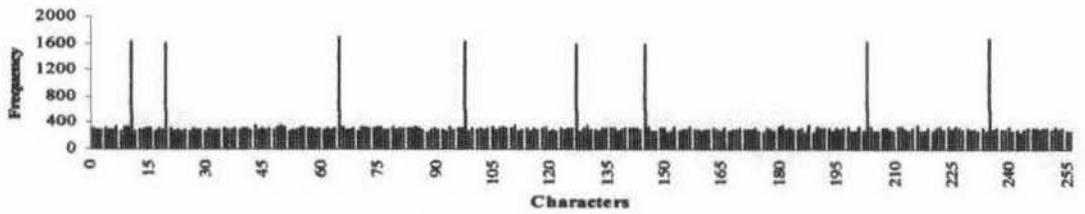
The χ^2 values and encryption times for MMAT+DSPB have been compared with those of Triple DES, each category of files being dealt with separately. Table 10.6 and figure 10.13 together reveal the test results and encryption times for MMAT+DSPB along with those for Triple DES in case of .dll files.



(a) Original .dll file

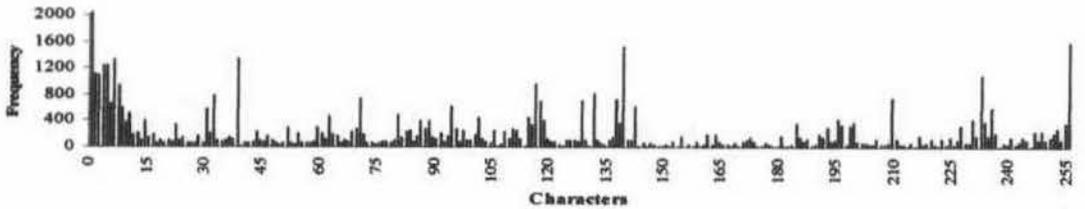


(b) .dll file encrypted with MMAT+DSPB

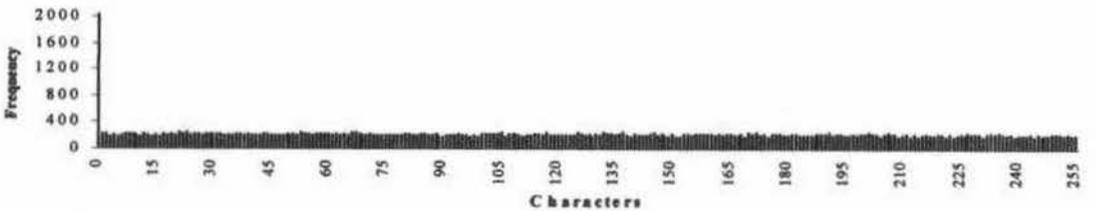


(c) .dll file encrypted with Triple DES

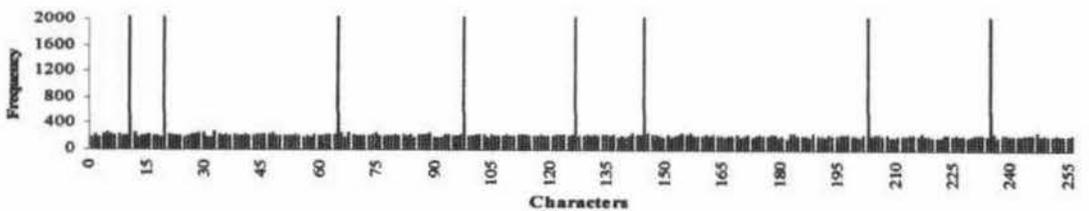
Figure 10.9: Character-frequencies in the source and encrypted .dll files



(a) Original .exe file



(b) .exe file encrypted with MMAT+DSPB

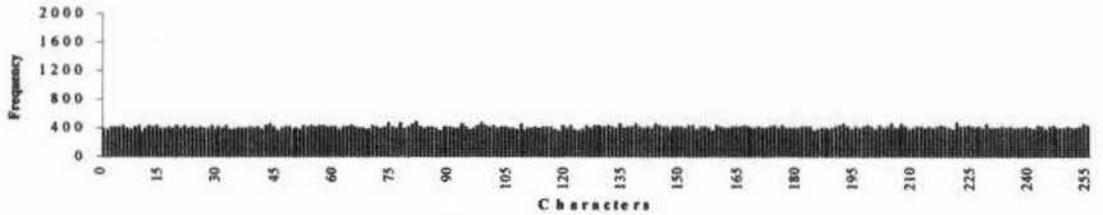


(c) .exe file encrypted with Triple DES

Figure 10.10: Character-frequencies in the source and encrypted .exe files



(a) Original .jpg file

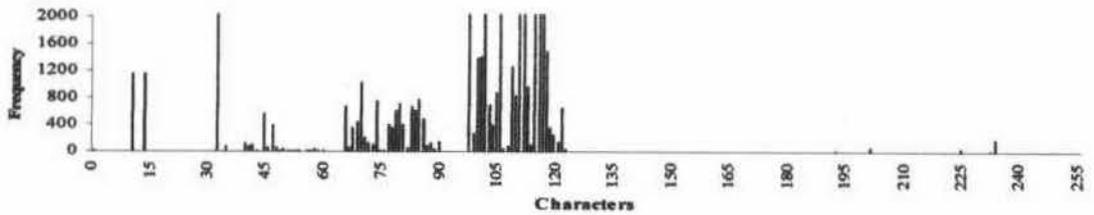


(b) .jpg file encrypted with MMAT+DSPB

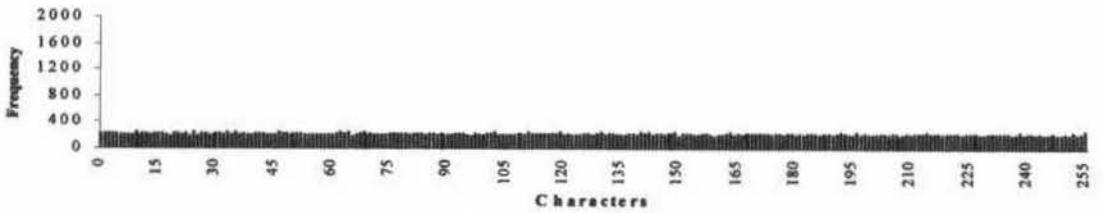


(c) .jpg file encrypted with Triple DES

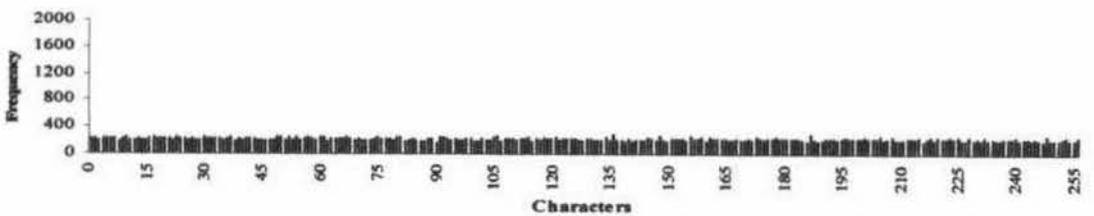
Figure 10.11: Character-frequencies in the source and encrypted .jpg files



(a) Original .txt file



(b) .txt file encrypted with MMAT+DSPB

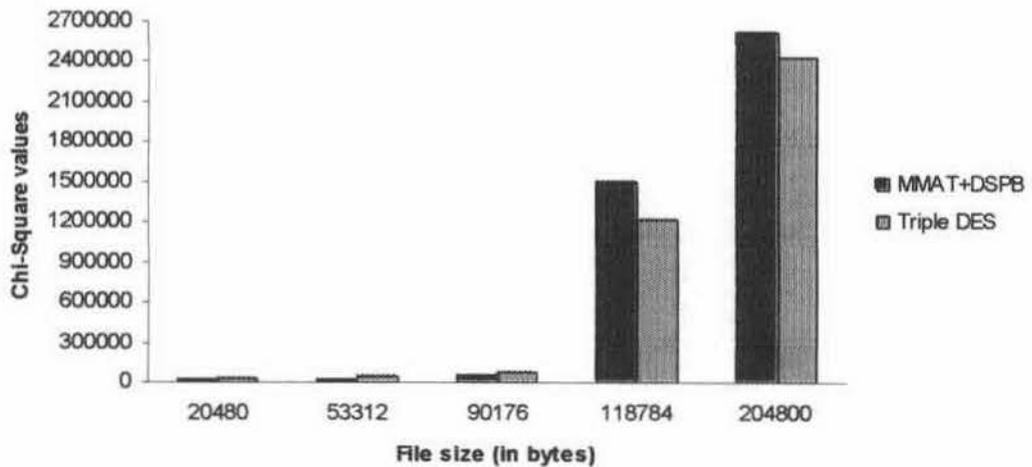


(c) .txt file encrypted with Triple DES

Figure 10.12: Character-frequencies in the source and encrypted .txt files

Table 10.6: χ^2 -test for MMAT+DSPB with .dll files

Sl. No.	Original file	File size (bytes)	MMAT+DSPB			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.dll	20480	0.10989	8610	255	06	29790	255
2	2.dll	53312	0.32967	21600	255	16	43835	255
3	3.dll	90176	0.54945	44073	255	26	66128	255
4	4.dll	118784	0.76923	1490116	255	34	1211289	255
5	5.dll	204800	1.26374	2618682	255	69	2416524	255

Figure 10.13: MMAT+DSPB vs. Triple DES in χ^2 -test of .dll files

The combination of MMAT and DSPB has shown a reasonably good performance in χ^2 -test with .dll files. Although the χ^2 values for MMAT+DSPB for smaller .dll files are little less than Triple DES, the values are better for larger files. Very small encryption time, compared to that of Triple DES, and very large χ^2 values, all with 255 degree of freedom (DF) for 256 characters, signify the strength of MMAT+DSPB.

Table 10.7 and figure 10.14 together defend the performance of MMAT+DSPB in the χ^2 -test in the instance of .exe files.

Table 10.7: χ^2 -test for MMAT+DSPB with .exe files

Sl. No.	Original file	File size (bytes)	MMAT+DSPB			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.exe	23104	0.10989	7690	255	12	8772	255
2	2.exe	52736	0.32967	23639	255	15	43426	255
3	3.exe	131136	0.76923	992862	255	29	986693	255
4	4.exe	170496	1.04396	279552	255	49	475893	255
5	5.exe	200832	1.15385	2801766	255	58	1847377	255

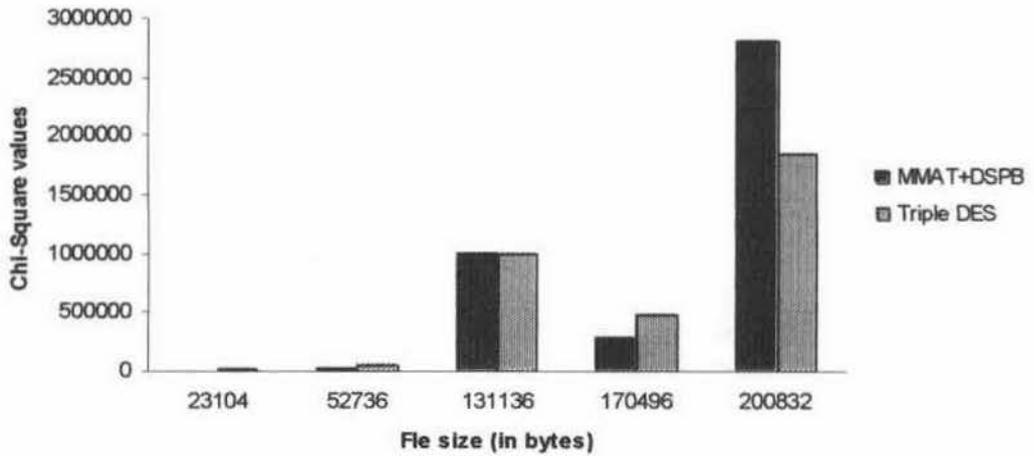


Figure 10.14: MMAT+DSPB vs. Triple DES in χ^2 -test of .exe files

MMAT+DSPB has given a mixed performance in the χ^2 -test for .exe files in comparison to Triple DES. The χ^2 values for two out of five .exe files are larger than Triple DES and they are more or same for the rest. The test results for .jpg files are revealed in table 10.8 and illustrated by figure 10.15.

Table 10.8: χ^2 -test for MMAT+DSPB with .jpg files

Sl. No.	Original file	File size (bytes)	MMAT+DSPB			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	1.jpg	28544	0.10989	4357	255	08	4331	255
2	2.jpg	71232	0.43956	2926	255	21	2916	255
3	3.jpg	105600	0.65934	5242	255	31	5227	255
4	4.jpg	160704	0.93407	22212	255	47	22314	255
5	5.jpg	216576	1.31869	30035	255	63	29824	255

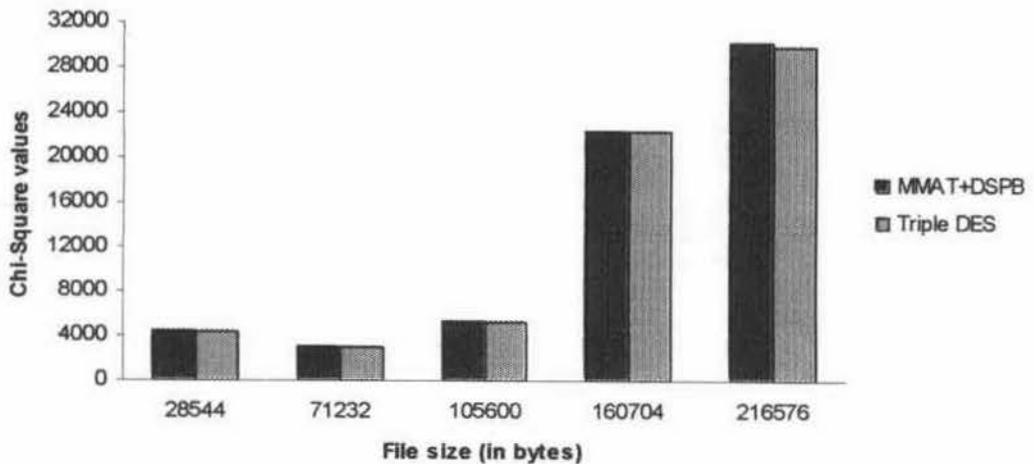


Figure 10.15: MMAT+DSPB vs. Triple DES in χ^2 -test of .jpg files

The nature shown by MMAT+DSPB in case of .jpg files is quite remarkable. The χ^2 values for four out of five .jpg files are larger than Triple DES and for the remaining one, it is almost equal. High χ^2 values, all with degree of freedom (DF) 255 except 239 for one, along with very small encryption times depict a good performance of MMAT+DSPB. Table 10.9 and figure 10.16 exhibit the results for .txt files.

Table 10.9: χ^2 -test for MMAT+DSPB with .txt files

Sl. No.	Original file	File size (bytes)	MMAT+DSPB			Triple DES		
			Time (secs.)	χ^2	DF	Time (secs.)	χ^2	DF
1	t1.txt	6976	0.054946	10669	190	02	10629	239
2	t2.txt	23808	0.164835	32763	255	07	32638	255
3	t3.txt	58688	0.384615	81525	255	17	82101	255
4	t4.txt	118784	0.769230	165616	255	35	170557	255
5	t5.txt	190784	1.153847	496903	255	55	430338	255

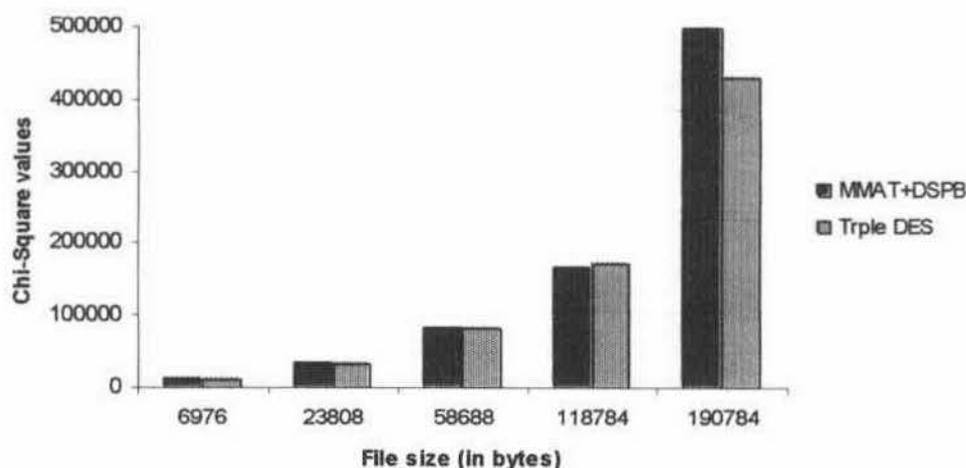


Figure 10.16: MMAT+DSPB vs. Triple DES in χ^2 -test of .txt files

The performance MMAT+DSPB in the χ^2 -test for .txt files are more or less similar to that of .jpg files and hence needs no further explanation.

10.3.3 Avalanche and runs

As usual, this test has been carried out to examine the diffusion and randomness property of the cipher, i.e. to test how much effect can be created in the cipher-text with a very small change in the plain-text and how random the cipher-texts. The results of this test for MMAT+DSPB are listed in table 10.10.

Table 10.10: Avalanche and runs in MMAT+DSPB

Bit complemented	Plain-text (Hex)	Cipher-text (Hex)	Number of runs		
			Plain-text	Cipher-text	Difference
None	4145D450	71B597BB	19	17	1
1 ST	C145D450	588DB511	18	19	1
2 ND	0145D450	619D793	17	15	2
3 RD	6145D450	617F97BD	19	13	6
4 TH	5145D450	317FB5F9	21	13	8
5 TH	4945D450	217FB5F8	21	12	9
6 TH	4545D450	217AA79A	21	18	3
7 TH	4345D450	A0FE1F9B	19	10	9
8 TH	4045D450	F33593AB	17	18	1
9 TH	41C5D450	5175973B	17	19	1
10 TH	4105D450	597D97B1	17	17	0
11 TH	4165D450	317F95DD	19	15	4
12 TH	4155D450	297CF5F9	21	15	6
13 TH	414DD450	287CB7F8	19	12	7
14 TH	4141D450	F17105F9	17	12	5
15 TH	4147D450	7BF41FAB	17	13	4
16 TH	4144D450	F735D3AB	19	18	1
17 TH	41455450	058FD71B	21	13	8
18 TH	41459450	2595D791	19	19	0
19 TH	4145F450	61BF95FD	17	13	4
20 TH	4145C450	7877B797	17	13	4
21 ST	4145DE50	217FB79A	17	14	3
22 ND	4145D050	F1F285D9	17	14	3
23 RD	4145D650	22BE1F9B	19	13	4
24 TH	4145D550	74941BAB	21	17	4
25 TH	4145D4D0	71BDC338	19	12	7
26 TH	4145D410	1D75D5B3	17	19	2
27 TH	4145D470	21BF95DD	17	15	2
28 TH	4145D440	7877B797	17	13	4
29 TH	4145D458	A89CB7BA	19	19	0
30 TH	4145D454	7BF427BA	21	14	7
31 ST	4145D452	7DB41FAB	21	15	6
32 ND	4145D451	5595DBBB	20	21	1

The results produced by MMAT+DSPB in this test are better than the combination of OMAT with BET. It is evident from the table that none of the consecutive cipher-texts are same or close to each other. This fact, along with very high differences of runs between the plain-text and the corresponding cipher-text, confirm the diffusion and randomness properties of MMAT+DSPB.

10.4 Conclusion

It has been observed with sufficient evidences that the OMAT+BET and MMAT+DSPB combinations are better than any individual algorithm proposed in this thesis. Besides, there is no difficulty in microprocessor-based implementations.

Proposed Key Formats

11.1 Introduction

An encryption scheme is unconditionally secure if the cipher-text generated by the scheme does not contain enough information to determine uniquely the corresponding plain-text, no matter how much cipher-text is available. An encryption algorithm can withstand any type of attack if the cost of breaking the cipher exceeds the value of the encrypted information and/or the time required to break the cipher exceeds the useful lifetime of the information. It is worth mentioning that it is very difficult to estimate the amount of effort required to cryptanalyze cipher-text successfully.

Cryptanalysis has long relied on the lengthy and patient application of trial and error method in order to crack a code. The advent of electronic computing dramatically enhanced that process. Today, anyone with a workstation can crack cipher-text encrypted with a short key. Clever software developers have even produced utility programs to recover passwords, often using 'brute force', which simply involves trying every possible key until the right one is found to get an intelligible translation of the secret code. The principal defence against brute force attack is to produce as long a list of legal keys as possible. As the list gets longer, so does the amount of work it could take to guess the right key. Table 11.1 illustrates this notion. For each key size, the results are shown assuming that it takes 1 μ s to perform a single encryption/decryption, which is a reasonable order of magnitude for today's machines. The final column of the table considers the results for a system that can process 1 million keys per microsecond.

Table 11.1: Average time required for exhaustive key search

Key-size (bits)	Number of keys ($2^{\text{key-size}}$ bits)	Time required at	
		1 encryption/ μ s	10^6 encryption/ μ s
16	6.6×10^{04}	0.033 secs	0.33 μ s
32	4.3×10^{09}	36 mins.	2.2×10^3 μ s
56 (DES)	7.2×10^{16}	1142 years	1001 hours
64	1.8×10^{19}	2.9×10^5 years	110 days
128	3.4×10^{38}	5.4×10^{24} years	5.4×10^{18} years

Keeping in mind the fact that a key should be long enough to remain safe, few formats of the key are proposed for the different algorithms that have been put forward in this thesis. Since each algorithm uses a different technique to encrypt data, each proposed format of the key may fit different proposed algorithms. Besides, although there are so many existing formats of the key that are quite secure, the proposed formats have been designed in such a manner so that it would not be too difficult to implement them in an Intel 8085 microprocessor-based system. It is worth mentioning here that it is quite difficult to handle data which are not multiples of 8 bits in the stated environment.

11.2 Format 1

In most of the proposed algorithms, seven rounds have been considered, each for 8, 16, 32, 64, 128, 256, and 512 bits block-size. MAT, OMAT and MMAT have six rounds each. The actual number of iterations in a particular round, less than the number of iterations to form a cycle for that block-size, is represented by a binary string. For example, the number of iterations in each round may be represented by a 16-bit binary string. Hence the maximum number of iterations that can be performed in each round will be $2^{16} = 65536$. Here, the maximum number of iterations does not mean the number of iterations to form a cycle in a particular round. One can check out from table 2.5 in chapter 2 that the number of iterations required by LSPB to form a cycle for block-size 128 is 117180. Similarly, DSPB needs 261744 iterations for 256-bit blocks and CSPB needs 68532 iterations for 512-bit blocks to complete one cycle. Any intermediate iteration may be chosen for generating the cipher-text. Hence selecting out 65536 iterations may be enough to generate the cipher-text and this number of iterations can be represented by a 16-bit binary string. Concatenating the strings representing the number of iterations in all the rounds, an encryption key of 112 bit ($16 \times 7 = 112$) is obtained. For those algorithms, which have only 6 rounds, the length of the key will be 96 bits only.

This key format may fit any of the proposed algorithms except BOS and BET, since very little number of iterations are required to regenerate the original block in any round of these algorithms. The other algorithms for which this format is intended

may be divided into two groups. The first group may include LSPB, RSPB, DSPB, CSPB, and SPOB where there is no separate decryption algorithm and the plain-text can be regenerated by just iterating the encryption process. The second group may include MAT, OMAT, MMAT and DEPS, where some iterations are given in the encryption and the same number of iterations is needed by the reverse process in decryption. In addition, since MAT, OMAT and MMAT involves in pairing of blocks, only 6 rounds will be possible for a 512-bit input string.

A particular session of a cipher may be considered where the source file is encrypted using a number of iterations in each round for block-sizes 8, 16, 32, 64, 128, 256 and 512 bits. Table 11.2 shows the corresponding binary value for the number of iterations in each round. The binary strings are concatenated together, starting from Round 7, to form the 128-bit binary string 11000011 01100101 11000010 11001110 10111111 00110110 10101101 10011011 10110100 10101010 00010000 11100001 00000010 10110010. This 128-bit binary string will be the encryption key for this particular session. During decryption, the same key is taken to iterate each round of the algorithm for the specified number of times.

Table 11.2: Formation of key in format 1

Round No.	No. of Iterations	
	Decimal	Binary
7	50021	1100001101100101
6	49870	1100001011001110
5	48950	1011111100110110
4	44443	1010110110011011
3	46250	1011010010101010
2	4321	0001000011100001
1	690	0000001010110010

For LSPB, RSPB, DSPB, CSPB, and SPOB, the number of iterations for each round of decryption is calculated by subtracting the actual number of iterations performed during encryption from the number of iterations required to form a cycle in that particular round. These remaining iterations are performed to decrypt the ciphertext. During decryption in MAT, OMAT, MMAT and DEPS, the same number of iterations in each round as in encryption is performed where the algorithm for decryption is the reverse of encryption.

11.3 Format 2

One of the disadvantages of the Format 1 discussed in the previous section is that, due to large number of iterations in each round, the encryption process may take too much time which is undesirable for large files since the files are encrypted in chunks of 512 bits and each chunk may need a lot of time to be encrypted. To overcome this problem, another format of the encryption key is proposed without compromising so much for the length of the key. A lesser number of iterations are allowed in each round by using by a shorter binary string for each round. For example, 10 bits may be used to represent the number of iterations to allow a maximum of $2^{10} = 1024$ iterations. In addition, a 3 bit string to represent the round number 1 to 7 may be included in the key. Hence, for each round, a total of 13 bits (10 + 3) are required to form a part of the key. Table 11.3 will illustrate this format in detail.

Table 11.3: Formation of key in format 2

Round No.		No. of Iterations
Decimal	Binary	
7	111	1001101101
6	110	1101011010
5	101	1010110110
4	100	0110011011
3	011	1010100010
2	010	0001100001
1	001	0010110010
0*	000	1101010111

For a particular round, 3 bit for representing the round and 10 bits for denoting the number iterations together make a 13 bit string. Since 3 bits will have 8 different combinations of 0's and 1's, the string '000' cannot be ignored. Hence, a dummy round, represented by 0* in the table, is also included with any random string for the number of iterations. Thus a $13 \times 8 = 104$ bit key may be formed by concatenating the strings for all the rounds including the dummy one. This format of encryption key is applicable to the same set of the proposed algorithms chosen for format 1. Moreover, the string '111' also becomes a dummy like '000' in case of MAT, OMAT and MMAT since there are only 6 rounds.

Format 2 of the encryption key has twofold advantage over format 1. Since the number of iterations in each round has reduced a lot in format 2, the process of encryption will be faster. Another very important advantage is that the different rounds of the encryption algorithm may be executed in a random order rather than following a sequence from 1 to 7. The key may be formed according to the order in which the various rounds are performed. This will further enhance the security.

11.4 Format 3

This format has been proposed exclusively for MMAT. Formats 1 and 2 can be used without any problem for MMAT. Format 3 has been proposed to avoid the iterations in each round to further reduce the encryption time. In MAT and OMAT, only one block of a pair is replaced by the result of the modulo-addition, and in MMAT, both the blocks are modified by the operation. A small variation in MMAT will make it more efficient and it will act as MAT for some blocks and as MMAT for others. To make the change, an option may be added by using the key that will decide whether to replace only the first block, or only the second block, or both the blocks. For this purpose, a 2-bit string is needed, where '01' will indicate the replacement of the second block (0 for no replacement and 1 for replacement), '10' will denote the replacement of the first block, and '00' or '11' are meant for changes in both the blocks of a pair.

In *round 1* of MMAT, there are 64 blocks of 8 bits each, i.e. there are 32 pairs of blocks. Hence, for this round 32 numbers of 2-bit strings will be required for the key. This means that the part of the key representing this round will have $32 \times 2 = 64$ bits. Two bits of the key are read at a time and the particular block-pair is modified depending on this 2-bit string. Similarly, for *round 2*, the part of the key will have 16 numbers of 2-bit strings, which is equal to a 32-bit string. In the same manner, the last round with block-size 256 will have just a single 2-bit string for the key. In this way, including all the rounds, the key will be $64 + 32 + 16 + 8 + 4 + 2 = 126$ bits long. The formation of the key will be more understandable from the following example.

Round 1: Block-size = 8 bits Number of blocks = 64

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	1	0	1	0	0	1	0	0	1	1	0	0	0	1	0	1	1	0	1	1	1	0	1	0	1	1	0	1	1	1	1
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
0	0	1	0	0	1	1	0	0	0	1	0	1	1	0	1	1	1	0	0	1	0	0	1	1	1	1	0	1	0	1	0

This means that in the first round of MMAT, where the block-size is 8 bits, the blocks of 8 bits will be substituted according to the supplied key. For example, for the first pair of block 1 and block 2, the 2-bit string is '11'. Therefore, both the blocks will be substituted as discussed in chapter 7. Similarly, for the pair of block 7 and block 8, only block 7 will be replaced as in MAT since the key string is '10'. For the pair of blocks 13 and 14, both will be replaced since the key-string is '00'. Hence, the part of the key for this round is 11010010 01100010 11011101 01101111 00100110 00101101 11001001 11101010.

Round 2: Block-size = 16 bits Number of blocks = 32

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	0	0	1	1	1	0	0	1	0	1	0	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	1	1	0	0	0

Part of the key for this round is 10011100 10101111 01001110 01011000.

Round 3: Block-size = 32 bits Number of blocks = 16

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	1	1	1	1	0	0	1	1	0	1	1	1

Part of the key for this round is 00001111 00110111.

Round 4: Block-size = 64 bits Number of blocks = 8

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

Part of the key for this round is 01101100.

Round 5: Block-size = 128 bits Number of blocks = 4

1	2	3	4
1	1	0	1

Part of the key for this round is 1101.

Round 6: Block-size = 256 bits Number of blocks = 2

1	2
0	1

Part of the key for this round is 01.

Concatenating all the parts of the key for the individual rounds, the final key of 126 bits is obtained as 11010010 01100010 11011101 01101111 00100110 00101101 11001001 11101010 10011100 10101111 01001110 01011000 00001111 00110111 01101100 1101 01.

11.5 Format 4

For algorithms like BOS and BET, formats 1 and 2 of the encryption key are not suitable since the number of iterations to form a cycle in each round is very small. For the maximum number of iterations in each round of these algorithms, just 5 bits will be enough. It should also be kept in mind that the size of the blocks should be multiples of 8, otherwise it would be difficult to implement the cipher in an Intel 8085 based system. Hence, other formats suggesting the block-size to be non-multiple of 8 have been avoided. Format 4 has been proposed specially for BOS and BET, although it may well fit into other proposed algorithms.

In this format, 5 bits are used to represent the number of iterations in a particular round. For a particular chunk of 512 bits, where there are 7 rounds, another 7 bits are used to represent the status of the iterations. Each bit in this 7 bit field denotes a particular round starting from the MSB. A '1' for a particular round means there are iterations to be performed in this round whose number is given by the 5-bit field. On the other hand, a '0' means no iterations, i.e. that particular round is executed only once. For example, if the 7-bit field is 1011101, then round 1, round 3, round 4, round 5 and round 7 will have to be iterated whose number will be given by the 5-bit fields, whereas round 2 and round 6 will have to be executed only once, whatever may be the values in the 5-bit fields. A total of $7 \times 5 = 35$ bits will be needed for the iterations in the 7 rounds. Hence, for a chunk of 512 bits, the part of the key would be as follows:

Status	Round 1 iterations	Round 2 iterations	Round 3 iterations	Round 4 iterations	Round 5 iterations	Round 6 iterations	Round 7 iterations
7 bits	5 bits	5 bits	5 bits	5 bits	5 bits	5 bits	5 bits

This part of the key will be 42 bits long. Three such strings with different bit patterns are concatenated to form the final key of $3 \times 42 = 126$ bits. During encryption,

the first chunk of 512 bits is read from the source file and it is encrypted with the first 42 bits of the key. Then the second and third chunks are encrypted using the second and third 42 bits of the key respectively. Since all three parts of the key are exhausted for encrypting three chunks of the source file, the fourth chunk of 512 bits from the source file is encrypted with the first part of the key. In this fashion, the three parts of the key are used again and again to encrypt the whole source file. Encryption may end with any part of the key depending on the size of the source file. It is worth mentioning here that the minimum size of a file to be encrypted using this key format will be 3×512 bits, i.e. 1536 bits, which is equal to 192 bytes. Practical experiences have shown that even a small text file containing an e-mail is about 4 KB. Hence, there should not be any problem in using this format of the encryption key.

11.6 Conclusion

There are so many other existing key formats that can be used for the proposed algorithms. Many researchers have suggested key formats involving variable segments of a file and the corresponding rank of those segments. Although this type of format is very useful in enhancing the security of the proposed algorithms, they can be employed for the high-level implementations only. The main issue at this instance is the 8085 microprocessor-based implementations. Hence, the formats suggested in this chapter are made as simple as possible, keeping in mind the word length of the Intel 8085 microprocessor.

For each of the combinations like OMAT+BET and MMAT+DSPB, two keys will be necessary, thereby enhancing the security imparted by these algorithms. Two keys together will form a very large key space. The formats of the encryption keys for OMAT and BET have been considered separately due to the dissimilar natures of these algorithms. Hence, two separate keys are must for the OMAT+BET combination. Since MMAT and DSPB may share the same key format, only one key may be used repeatedly for the MMAT+DSPB combination.

Since the keys in the proposed formats are long enough, they may be considered computationally secure. Even then, more vigorous testing is required before it can be taken for granted that the proposed schemes are not vulnerable to any

kind of attack. Table 11.4 lists the efforts required to break the proposed keys using the brute force (exhaustive search) method.

Table 11.4: Brute force attack on the proposed key formats

Proposed Key	Key length (bits)	Number of alternate keys	Time required at 1 encryption/ μ s	Time required at 10^6 encryption/ μ s
Format 1	112	5.1923×10^{33}	8.23233×10^{19} years	8.23233×10^{13} years
Format 2	104	2.0282×10^{31}	3.21575×10^{17} years	3.21575×10^{11} years
Format 3	126	8.5071×10^{37}	1.34879×10^{24} years	1.34879×10^{18} years
Format 4	126	8.5071×10^{37}	1.34879×10^{24} years	1.34879×10^{18} years

The figures from table 11.4 indicate that the proposed keys are not vulnerable to brute force attack, which in turn makes the proposed algorithms computationally secure. Even then, it does not necessarily mean that the key should be of excessively long size, because this, in turn, may be regarded as an overhead. Hence the proposed keys have been designed while keeping a proper balance between these two issues. Last, but not the least, the proposed algorithms and the corresponding keys are feasible for the intended targets, more precisely microprocessor-based systems.

Concluding Discussions

12.1 Introduction

In this chapter, the proposed algorithms have been compared to each other in various aspects to draw the conclusions. The test results of all the proposed algorithms have already been analysed individually in the respective chapters and each has been compared with Triple DES, which has been used as a benchmark since it is one of the most popular standard algorithms. Some algorithms have shown very good results in comparison to Triple DES. Moreover, some algorithms were good in one test and some in the other. The use of the same set of files to test all the algorithms has made the comparative study quite easy.

The character frequencies are considered first to begin with. The character frequencies produced by the proposed ciphers have been compared numerically rather than graphically. Different categories of files have different trends of character frequencies. For example, the text files (.txt format) do not have non-printable characters and hence the characters are generally clustered in the region of printable characters. Similarly, the characters in the .dll files are clustered within some particular regions. A good cipher has to distribute these characters evenly over the character space, i.e. all the 0 to 255 ASCII characters should have more or less same frequency. To study this aspect of a proposed algorithm, the arithmetic mean of the frequencies is computed and the average distance of the frequencies from this mean is measured. Lower the average distance more even is the character distribution. Another way to check the evenness of the frequencies is to compute the more accepted statistical value, the *standard deviation*. Lesser the standard deviation better may be the cipher considered. Tables 12.1 to 12.4 list these values for all the algorithms being considered for each category of files. Similarly, to compare the heterogeneity between the source and the encrypted files, the χ^2 values are compared. Tables 12.5 to 12.8 give comparative views of the χ^2 values for the different categories of files. Last, but not the least, to look into the efficiency aspect of the proposed algorithms for the intended targets, their encryption times (in secs.) have been compared in Table 12.9. The conclusions have been drawn in section 12.5.

Table 12.1: Comparison of character frequencies for .dll file

Particulars	LSPB	RSPB	DSPB	CSPB	BET	SPOB	MAT	OMAT	MMAT	BOS	DEPS	MMAT + DSPB	OMAT + BET	Triple DES
Minimum frequency	162	153	148	176	18	10	256	257	259	0	52	264	265	256
Maximum Frequency	12786	12719	13120	12264	17428	19711	10999	10827	10595	21843	13422	10633	10830	1691
Arithmetic Mean (AM)	352.25	352.25	352.25	352.25	352.25	352.25	352.25	352.25	352.25	352.25	352.25	352.25	352.25	352.25
Average difference from AM	157.39	155.93	159.26	150.24	259.80	359.69	85.19	82.22	80.17	492.04	250.66	80.50	82.22	81.04
Standard Deviation	790.87	786.76	811.84	757.30	1099.58	1289.14	668.51	657.54	642.97	1526.86	896.17	645.34	657.71	232.38

Table 12.2: Comparison of character frequencies for .exe file

Particulars	LSPB	RSPB	DSPB	CSPB	BET	SPOB	MAT	OMAT	MMAT	BOS	DEPS	MMAT + DSPB	OMAT + BET	Triple DES
Minimum frequency	107	105	104	104	26	7	179	180	183	0	33	187	184	122
Maximum Frequency	9780	9762	9849	9567	10786	12316	8481	8333	8154	8182	10647	8189	8346	7182
Arithmetic Mean (AM)	256.25	256.25	256.25	256.25	256.25	256.25	256.25	256.25	256.25	256.25	256.25	256.25	256.25	256.25
Average difference from AM	118.75	119.41	120.05	117.81	178.78	248.45	68.11	63.87	62.25	408.70	181.64	62.16	63.79	130.12
Standard Deviation	605.02	603.90	609.43	591.76	681.66	804.78	516.65	507.09	495.78	910.22	678.74	498.00	507.88	501.22

Table 12.3: Comparison of character frequencies for .jpg file

Particulars	LSPB	RSPB	DSPB	CSPB	BET	SPOB	MAT	OMAT	MMAT	BOS	DEPS	MMAT + DSPB	OMAT + BET	Triple DES
Minimum frequency	333	330	314	339	238	199	362	360	353	0	237	346	353	361
Maximum Frequency	731	697	736	620	1140	1902	466	471	496	9022	1151	486	485	475
Arithmetic Mean (AM)	412.50	412.50	412.50	412.50	412.50	412.50	412.50	412.50	412.50	412.50	412.50	412.50	412.50	412.50
Average difference from AM	30.55	31.88	33.43	30.13	52.82	98.03	16.58	17.14	14.86	635.75	67.07	16.58	16.93	15.42
Standard Deviation	45.55	44.08	48.18	40.06	80.65	148.98	20.62	21.15	19.55	1385.10	98.55	21.65	21.18	19.32

Table 12.4: Comparison of character frequencies for .txt file

Particulars	LSPB	RSPB	DSPB	CSPB	BET	SPOB	MAT	OMAT	MMAT	BOS	DEPS	MMAT + DSPB	OMAT + BET	Triple DES
Minimum frequency	77	51	38	44	0	0	142	191	197	0	0	191	187	177
Maximum Frequency	919	1116	936	749	5744	10820	391	380	275	28441	4073	269	310	302
Arithmetic Mean (AM)	229.25	229.25	229.25	229.25	229.25	229.25	229.25	229.25	229.25	229.25	229.25	229.25	229.25	229.25
Average difference from AM	92.48	95.95	96.31	86.43	280.48	366.36	33.21	16.35	11.93	373.79	257.82	10.81	13.92	15.07
Standard Deviation	127.79	131.60	130.31	114.52	511.08	867.47	41.51	22.40	14.84	1826.86	442.83	13.76	17.89	19.38

Table 12.5: Comparison of χ^2 values for .dll files

Particulars	LSPB	RSPB	DSPB	CSPB	BET	SPOB	MAT	OMAT	MMAT	BOS	DEPS	OMAT + BET	MMAT + DSPB	Triple DES
1.dll	4833	4816	4651	5136	5471	4912	6658	7808	8539	13177	7012	7827	8610	29790
2.dll	16449	16408	15799	16421	17601	20959	19750	20495	21592	46062	22285	21078	21600	43835
3.dll	40879	40957	40379	41168	40798	48891	41532	42195	43890	311218	40456	43415	44073	66128
4.dll	111298	111767	109164	116385	1920600	1358795	136269	14224	149143	415291	126714	142618	1490116	1211289
5.dll	1014978	1047282	936791	1358258	7376322	2897554	534082	142224	620375	1569635	1578332	563669	2618682	2416524

Table 12.6: Comparison of χ^2 values for .exe files

Particulars	LSPB	RSPB	DSPB	CSPB	BET	SPOB	MAT	OMAT	MMAT	BOS	DEPS	OMAT + BET	MMAT + DSPB	Triple DES
1.exe	7009	6905	6763	7004	8107	10382	7077	7521	7633	30676	7564	7518	7690	8772
2.exe	17212	17387	16802	17018	19221	23702	21113	22526	23766	46235	22863	22893	23639	43426
3.exe	929886	930209	927133	937597	899616	858925	978811	985682	993854	2321449	900984	986123	992862	986693
4.exe	154446	155177	147562	168235	131095	534803	238430	250534	280797	1958490	151307	256526	279552	475893
5.exe	2274398	2280662	2196194	2342468	2217990	14429014	2601366	2690803	2748185	2004973	2119639	2696344	2801766	1847377

Table 12.7: Comparison of χ^2 values for .jpg files

Particulars	LSPB	RSPB	DSPB	CSPB	BET	SPOB	MAT	OMAT	MMAT	BOS	DEPS	OMAT + BET	MMAT + DSPB	Triple DES
1.jpg	3820	3842	3801	3996	4196	7414	4179	4258	4313	41118	4053	4489	4357	4331
2.jpg	2521	2450	2466	2421	3604	3716	2671	2696	2892	101874	2963	2847	2926	2916
3.jpg	4016	4006	4004	4191	4724	6196	5087	5198	5184	514189	4439	5032	5242	5227
4.jpg	8707	8510	8845	8423	14853	9001	21061	21824	22284	531483	23053	22045	22212	22314
5.jpg	11394	11132	11512	11325	20862	12719	28771	29028	29845	3198284	31100	29190	30035	29824

Table 12.8: Comparison of χ^2 values for .txt files

Particulars	LSPB	RSPB	DSPB	CSPB	BET	SPOB	MAT	OMAT	MMAT	BOS	DEPS	OMAT + BET	MMAT + DSPB	Triple DES
t1.txt	8014	8062	8049	7927	10028	6213	10269	10225	10580	5158	7895	10423	10669	10629
t2.txt	32154	33473	33345	32433	32962	43081	31443	31685	32556	42571	28461	32829	32763	32638
t3.txt	80779	83007	83061	81405	81943	102298	78276	79093	81919	105767	70257	81725	81525	82101
t4.txt	165273	170273	170692	166288	175305	215996	161141	161812	165899	409852	154803	166308	165616	170557
t5.txt	407160	427406	427033	409902	406733	706203	407030	415275	165753	7499376	400590	434533	496903	430338

Table 12.9: Comparative encryption time (in secs.)

Particulars	LSPB	RSPB	DSPB	CSPB	BET	SPOB	MAT	OMAT	MMAT	BOS	DEPS	OMAT + BET	MMAT + DSPB	Triple DES
1.dll	0.054945	0.054945	0.054945	0.109890	0.054945	0.109890	0.054945	0.054945	0.054945	0.109890	0.989011	0.10989	0.10989	6.00
2.dll	0.109890	0.109890	0.164835	0.219780	0.164835	0.274725	0.109890	0.164835	0.164835	0.219780	3.021978	0.32967	0.32967	16.00
3.dll	0.219780	0.219780	0.274725	0.384615	0.274725	0.494505	0.329670	0.274725	0.329670	0.329670	5.164835	0.54945	0.604395	26.00
4.dll	0.329670	0.274725	0.439560	0.604396	0.384615	0.604396	0.494505	0.384615	0.384615	0.384615	6.593406	0.76923	0.824175	34.00
5.dll	0.549451	0.549451	0.714286	0.934066	0.604396	1.098901	0.714286	0.659341	0.714286	0.714286	11.648351	1.263737	1.428572	69.00
1.exe	0.054945	0.054945	0.054945	0.109890	0.054945	0.109890	0.054945	0.054945	0.054945	0.109890	1.428571	0.10989	0.10989	12.00
2.exe	0.109890	0.054945	0.164835	0.274725	0.164835	0.274725	0.164835	0.164835	0.164835	0.164835	2.967033	0.32967	0.32967	15.00
3.exe	0.329670	0.329670	0.439560	0.604396	0.384615	0.659341	0.384615	0.384615	0.329670	0.384615	6.593406	0.76923	0.76923	29.00
4.exe	0.439560	0.439560	0.604396	0.824176	0.549451	0.934066	0.549451	0.494505	0.659341	0.494505	10.164835	1.043956	1.263737	49.00
5.exe	0.549451	0.549451	0.659341	0.934066	0.604396	0.989011	0.714286	0.549451	0.604396	0.604396	16.593407	1.153847	1.263737	58.00
1.jpg	0.054945	0.054945	0.109890	0.164835	0.054945	0.164835	0.054945	0.054945	0.054945	0.109890	1.703297	0.10989	0.164835	8.00
2.jpg	0.219780	0.109890	0.274725	0.329670	0.219780	0.329670	0.219780	0.219780	0.274725	0.219780	4.285714	0.43956	0.54945	21.00
3.jpg	0.274725	0.329670	0.329670	0.494505	0.329670	0.549451	0.329670	0.329670	0.384615	0.384615	6.318681	0.65934	0.714285	31.00
4.jpg	0.439560	0.439560	0.604396	0.714286	0.439560	0.824176	0.494505	0.494505	0.604396	0.494505	9.725274	0.934065	1.208792	47.00
5.jpg	0.604396	0.604396	0.769231	0.989011	0.659341	1.153846	0.714286	0.659341	0.769231	0.714286	13.296702	1.318682	1.538462	63.00
t1.txt	0.054945	0.054945	0.054945	0.054945	0.054945	0.054945	0.000000	0.000000	0.054945	0.054945	0.439560	0.054946	0.10989	2.00
t2.txt	0.109890	0.109890	0.109890	0.109890	0.109890	0.109890	0.109890	0.054945	0.109890	0.109890	1.428571	0.164835	0.21978	7.00
t3.txt	0.164835	0.164835	0.219780	0.274725	0.219780	0.329670	0.219780	0.164835	0.219780	0.164835	3.516483	0.384615	0.43956	17.00
t4.txt	0.329670	0.329670	0.439560	0.604396	0.384615	0.604396	0.329670	0.384615	0.439560	0.329670	7.142857	0.76923	0.87912	35.00
t5.txt	0.494505	0.494505	0.659341	0.934066	0.549451	0.989011	0.659341	0.604396	0.659341	0.549451	11.538461	1.153847	1.318682	55.00
Average	0.27473	0.26648	0.35714	0.48352	0.31319	0.53297	0.33516	0.30769	0.35165	0.33242	6.22802	0.62088	0.70879	30

12.2 Comparison of character frequencies

As discussed in the previous section, data have been collected from the character frequency tables for the proposed algorithms. Each category of files has been treated separately. Comparisons have been made according to the 'average difference from the arithmetic mean' and the 'standard deviation' obtained from the tables.

In case of .dll files, it is clear from table 12.1 that, among the proposed algorithms, MMAT has the lowest average difference of 80.17 from the arithmetic mean, which means that most of the characters have frequencies close to the average frequency. This value is almost equal to that of Triple DES. MMAT also has the lowest standard deviation of 642.97 among the proposed ciphers, but it is quite higher than that of Triple DES. The second best performance for .dll files has been shown by MMAT+DSPB.

For .exe files, MMAT+DSPB has the lowest average difference of 62.16 from the arithmetic mean. On the other hand, the standard deviation 495.78 of MMAT is the lowest among the proposed ciphers. These values are quite close to that of Triple DES.

MMAT emerges the best among all in the case of .jpg files also. The average difference from the arithmetic mean in MMAT for .jpg files is 14.86, which is lower than Triple DES, and the standard deviation is 19.55, almost equal to Triples DES. In this case also, the second best performance has been shown by MMAT+DSPB.

For .txt files, MMAT+DSPB combination surpasses all the proposed algorithms. The results are even better than Triple DES. The average difference from the arithmetic mean is 10.81 and the standard deviation is 13.76, both of which are lower than Triple DES.

Considering all the four categories of files at the same time for the character frequencies generated by the proposed algorithms, MMAT goes beyond all. Hence

with regard to character frequencies, MMAT may be considered the best among the proposed algorithms, which is closely followed by MMAT+DSPB.

12.3 Comparison of χ^2 values

The comparison of χ^2 values for each file will be too long to accommodate here. Hence the average in each category has been computed. A high average will prove a cipher to be a good one. For .dll files, the highest average of the χ^2 values is 1872158 for BET. Similarly, SPOB has the highest average χ^2 value of 3171365 for .exe files. The same for .jpg file is 877390 given by BOS, and for .txt file, it is 161252 also given by BOS. On the whole, considering all the twenty files simultaneously, BOS, with the average χ^2 value of 1034248, has shown the best performance closely followed by SPOB and BET. Although all other transposition ciphers have χ^2 values close to that of BOS, the average performance of MMAT is also not so poor in comparison to BOS.

12.4 Comparison of encryption time

As already mentioned in section 12.1, the encryption time taken by the proposed algorithms for all the twenty files have been listed in table 12.9. The encryption time has been measured for the current implementation using a standard Pentium IV machine. It is obvious that the transposition ciphers will take less time than substitution ciphers since the mathematics involved in these algorithms are not as complex as in substitution ciphers. Comparing the encryption time for all the twenty files at a time, RSPB, with an average encryption time of 0.26648 seconds, is the fastest of all the proposed algorithms. The average encryption time of Triple DES, which has been used as a benchmark, is 30 seconds, much higher than the same for any of the proposed ciphers.

12.5 Conclusions

Although the transposition ciphers have very small encryption time, the same for the substitution ciphers are also very close, the average encryption time for all the proposed algorithms being below 1 second, except for DEPS which has an average

encryption time of 6.22802 seconds. Therefore, considering all the aspects of comparison, namely the character frequencies, the χ^2 values, and the encryption time, it may be concluded that MMAT and MMAT+DSPB have shown the best performances among all the proposed algorithms.

Cryptography is a crucial part of many good security systems. Even then, it can also make systems weaker when used in inappropriate ways. In many situations it provides only a feeling of security, but not actual security. This is often that is required, because that is what most customers want. They want to feel secure, but they don't want the hassle that comes with complex systems. There are no complex systems that are fully secure. Therefore, care has been taken to make the implementations of proposed algorithms as simple as possible.

A security system is only as strong as its weakest link. Strictly speaking, strengthening anything other than the weakest link is useless. In a cryptosystem, the weakest link may be the cipher itself, or the keys, or any other part. Anyway, each one of the proposed algorithms has been analysed in its weakest form so that the cipher itself and the associated keys may not be vulnerable to the attackers.

Sincere efforts have been made to design, analyse and implement the proposed algorithms. Each proposed algorithm has its own merits and demerits. Unfortunately, no algorithm is 100% proof and sustainable against the prevailing threats. However, the algorithms have been designed and proposed keeping in mind the intended targets, to be precise, the microprocessor-based systems with less powerful processors and very low capacity memory units.

C Source Codes

1. "encrypt.h" - C header file: used by all the programs for encryption and decryption.

```

#include<stdio.h>
#include<math.h>
#include<conio.h>
int ch;
int i,j,k,m,b=0,x[8],loop;
unsigned long size;
char *file2;
int array_512[513];
int array_ptr=1;
int pad;
int pad_info_1[8],pad_info_2[8];
int pad_info[16];
unsigned PrimeBuff[513];
/*-----*/
long filesize(FILE *);
void binary(int);
int ascii(int);
int ascii_pad(int[]);
void FindPrime();
/*-----*/
int bits_addition(int block_end,int array_ptr, int block_size);
int bits_addition_two(int block_end,int array_ptr, int block_size);
/*-----*/
int bits_substraction(int block_end,int array_ptr, int block_size);
int bits_substraction_two(int block_end,int array_ptr, int block_size);
/*-----*/
void padding_information (int);
void file_padding(long out_loop,long remender,char *sf);
void delete_padding(char*,char*);
/*-----*/
int Rotate_Right_CY(int start_position,int block_length, int cary);
int Rotate_Left_CY(int start_position,int block_length, int cary);
void INR(int start_position,int block_length);
/*-----*/
int Rotate_Right_CY_B(int*,int start_position,int block_length, int cary);
int Rotate_Left_CY_B(int*,int start_position,int block_length, int cary);
void INR_B(int*,int start_position,int block_length);
void DCR_B(int*,int start_position,int block_length);
/*-----*/
int bits_substraction(int block_end,int array_ptr, int block_size)

```

```

{
int carry=0,i;
for(i=0;i<=block_end;i++)
{
array_512[array_ptr+block_size]=+array_512[array_ptr+block_size]-array_512[array_ptr]-carry;
if(array_512[array_ptr+block_size]==0)
carry=0;
else if(array_512[array_ptr+block_size]==-1)
{
array_512[array_ptr+block_size]=1;
carry=1;
}
else if(array_512[array_ptr+block_size]==-2)
{
array_512[array_ptr+block_size]=0;
carry=1;
}
array_ptr--;
}
return (array_ptr);
}
/*-----*/
int bits_substraction_two(int block_end,int array_ptr, int block_size)
{
int carry=0,i;
for(i=0;i<=block_end;i++)
{
array_512[array_ptr]=-array_512[array_ptr+block_size]+array_512[array_ptr]-carry;
if(array_512[array_ptr]==0)
carry=0;
else if(array_512[array_ptr]==-1)
{
array_512[array_ptr]=1;
carry=1;
}
else if(array_512[array_ptr]==-2)
{
array_512[array_ptr]=0;
carry=1;
}
array_ptr--;
}
return (array_ptr);
}
/*-----*/
int bits_addition_two(int block_end,int array_ptr, int block_size)
{
int carry=0,i;
for(i=0;i<=block_end;i++)
{

```

```

array_512[array_ptr]= array_512[array_ptr+block_size]+ array_512[array_ptr]+carry;
if(array_512[array_ptr]==0 || array_512[array_ptr]==1)
carry=0;
else if(array_512[array_ptr]==2)
{
array_512[array_ptr]=0;
carry=1;
}
else if(array_512[array_ptr]==3)
{
array_512[array_ptr]=1;
carry=1;
}
array_ptr--;
}
return (array_ptr);
}
/*-----*/
int bits_addition(int block_end,int array_ptr, int block_size)
{
int carry=0,i;
for(i=0;i<=block_end;i++)
{
array_512[array_ptr+block_size]= array_512[array_ptr+block_size]+
array_512[array_ptr]+carry;
if(array_512[array_ptr+block_size]==0 || array_512[array_ptr+block_size]==1)
carry=0;
else if(array_512[array_ptr+block_size]==2)
{
array_512[array_ptr+block_size]=0;
carry=1;
}
else if(array_512[array_ptr+block_size]==3)
{
array_512[array_ptr+block_size]=1;
carry=1;
}
array_ptr--;
}
return (array_ptr);
}
/*-----*/
long filesize(FILE *stream)
{
long curpos, length;
curpos = ftell(stream);
fseek(stream, 0L, SEEK_END);
length = ftell(stream);
fseek(stream, curpos, SEEK_SET);
return length;
}

```

```

}
/*-----*/
void binary(int c)
{
int m,i,k;
for(i=7;i>=0;i--) //binary conversion
{
m=1<<i;
k=c&m;
x[7-i]=(k==0)?0:1;
}
}
/*-----*/
int ascii(int j)
{
int b=0,i;
for( i=0;i<8;i++) //ascii conversion
b=b+ (array_512[j++]*pow(2,(7-i)));
return(b);
}
/*-----*/
int ascii_pad ( int pad[])
{
int b=0,i;
for( i=0;i<8;i++) //ascii conversion
b=b+ (pad[i]*pow(2,(7-i)));
return(b);
}
/*-----*/
void FindPrime()
{
int a=3, b, i;
for(i=1; i<=512; i++)
PrimeBuff[i]=0;
for(a=3, i=1, PrimeBuff[2]=1; a<=512; a+=2)
{
for(b=3; b*b<=a; b+=2)
if (!(a%b))
break;
if (a<b*b)
PrimeBuff[a]=1;
}
}
/*-----*/
void delete_padding(char *sf,char *file2)
{
FILE *fs,*ft;
fs=fopen(sf,"rb");
fseek(fs,-2,SEEK_END);

```

```

ch=getc(fs);
binary(ch);
j=0;
for(i=0;i<8;i++)
pad_info[j++]=x[i];
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
pad_info[j++]=x[i];
printf("\n\n");
for(i=0;i<16;i++)
printf(" %d",pad_info[i]);
b=0;k=0;
for( i=0;i<16;i++) //ascii conversion
b=b+ (pad_info[k++]*pow(2,(15-i)));
printf(" %d",b);
b=(b/8)+2;
printf(" %d",b);
size=size-b;
printf(" \nsize=%ld",size);
printf(" \nfile2=%s",file2);
fclose(fs);
fs=fopen(file2,"rb");
ft=fopen("tt","wb");
while(size>0)
{
ch=getc(fs);
putc(ch,ft);
size--;
}
fclose(ft);
fclose(fs);
remove(file2);
rename("tt",file2);
}
/*_____*/
void file_padding(long out_loop,long remender,char *sf)
{
long curpos_ss;
FILE *fs;
fs=fopen(sf,"rb");
curpos_ss= out_loop*64;
fseek(fs, curpos_ss, SEEK_SET);
array_ptr=1;
pad=512-(remender*8);
printf("\npad=%d",pad);
while(remender!=0)
{
ch=getc(fs);
binary(ch);

```

```

for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
remender--;
}
for( i=0;i<pad;i++)
array_512[array_ptr++]=0;
}
/*_____*/
void padding_information (int pad)
{
for(i=15;i>=0;i--) //binary conversion
{
m=1<<i;
k=pad&m;
pad_info[15-i]=(k==0)?0:1;
}
printf("pad _info from pad_inform:\n");
/*_____*/
for(i=0;i<16;i++)
printf(" %d",pad_info[i]);
/*_____*/
j=0;
for(i=0;i<8;i++)
pad_info_1[i]=pad_info[j++];
for(i=0;i<8;i++)
pad_info_2[i]=pad_info[j++];
/*_____*/
for(i=0;i<8;i++)
printf(" %d",pad_info_1[i]);
/*_____*/
printf("\npad _info22222 from pad_inform:\n");
/*_____*/
for(i=0;i<8;i++)
printf(" %d",pad_info_2[i]);
}
/*_____*/
void INR_B(int *B,int start_position,int block_length)
{
int last_position=start_position+(block_length-1);
int cary=1,i;
for(i=0;i<block_length;i++)
{
*(B+last_position)=*(B+last_position)+cary;
if(*(B+last_position)==0 || *(B+last_position)==1)
cary=0;
else if(*(B+last_position)==2)
{
*(B+last_position)=0;
cary=1;
}
}
}

```

```

last_position--;
}
}
/*_____*/
void DCR_B(int *B,int start_position,int block_length)
{
int last_position=start_position+(block_length-1), cary=1,i;
for(i=0;i<block_length;i++)
{
*(B+last_position)=*(B+last_position)-cary;
if(*(B+last_position)==0)
cary=0;
else if(*(B+last_position)==1)
{
cary=0;
}
else if(*(B+last_position)==-1)
{
*(B+last_position)=1;
cary=1;
}
last_position--;
}
}
/*_____*/
void INR(int start_position,int block_length)
{
int last_position=start_position+(block_length-1);
int cary=1,i;
for(i=0;i<block_length;i++)
{
array_512[last_position]=array_512[last_position]+cary;
if(array_512[last_position]==0 || array_512[last_position]==1)
cary=0;
else if(array_512[last_position]==2)
{
array_512[last_position]=0;
cary=1;
}
last_position--;
}
}
/*_____*/
int Rotate_Right_CY_B(int *B,int start_position,int block_length, int cary)
{
int start_position_temp=start_position;
int cary_temp= cary;
for(int i=start_position;i<=block_length;i++)
printf(" %d ",*(B+i));
printf("\n carry=%d ",cary);
}

```

```

cary_temp=*(B+start_position+(block_length-1));
printf("\n New _ carry=%d ",cary_temp);
for(i=1;i<block_length;i++)
{
*(B+start_position+(block_length-i))=*(B+start_position+((block_length-i)-1));
}
*(B+start_position)=cary;
cary=cary_temp;
printf("\n");
for(i=start_position_temp ;i<=block_length;i++)
printf(" %d ",*(B+i));
return(cary);
}
/*_____*/
int Rotate_Right_CY(int start_position,int block_length, int cary)
{
int cary_temp= cary;
cary_temp=array_512[start_position+(block_length-1)];
for(int i=1;i<block_length;i++)
{
array_512[start_position+(block_length-i)]=array_512[start_position+((block_length-i)-1)];
}
array_512[start_position]=cary;
cary=cary_temp;
return(cary);
}
/*_____*/
int Rotate_Right(int start_position,int block_length, int cary)
{
int start_position_temp=start_position;
for(int i=start_position;i<=block_length;i++)
printf(" %d ",array_512[i]);
printf("\n carry=%d ",cary);
cary=array_512[start_position+(block_length-1)];
printf("\n carry=%d ",cary);
for(i=1;i<block_length;i++)
{
array_512[start_position+(block_length-i)]=array_512[start_position+((block_length-i)-1)];
}
array_512[start_position]=cary;
printf("\n");
for(i=start_position_temp ;i<=block_length;i++)
printf(" %d ",array_512[i]);
return(cary);
}
/*_____*/
int Rotate_Left_CY(int start_position,int block_length, int cary)
{
int start_position_temp=start_position, cary_temp=cary;
for(int i= start_position;i<=block_length;i++)

```

```

printf(" %d ",array_512[i]);
printf("\n carry=%d ",cary);
cary=array_512[start_position];
printf("\n carry=%d ",cary);
for(i=1;i<block_length;i++)
{
array_512[start_position]=array_512[start_position+1];
start_position++;
}
printf("\n carry_temp=%d ",cary_temp);
array_512[start_position_temp+(block_length-1)]=cary_temp;
/*_____*/
printf("\n");
for(i=start_position_temp;i<=block_length;i++)
printf(" %d ",array_512[i]);
return(cary);
}
/*_____*/
int Rotate_Left_CY_B(int *B,int start_position,int block_length, int cary)
{
int start_position_temp=start_position, cary_temp=cary;
cary=*(B+start_position);
for(int i=1;i<block_length;i++)
{
*(B+start_position)=*(B+start_position+1);
start_position++;
}
*(B+start_position_temp+(block_length-1))=cary_temp;
/*_____*/
return(cary);
}
int Rotate_Left(int start_position,int block_length, int cary)
{
int start_position_temp=start_position;
for(int i= start_position;i<=block_length;i++)
printf(" %d ",array_512[i]);
printf("\n carry=%d ",cary);
cary=array_512[start_position];
printf("\n carry=%d ",cary);
for(i=1;i<block_length;i++)
{
array_512[start_position]=array_512[start_position+1];
start_position++;
}
array_512[start_position]=cary;
printf("\n");
for(i=start_position_temp;i<=block_length;i++)
printf(" %d ",array_512[i]);
return(cary);
}

```

2. Source code for LSPB encryption/decryption

```

#include<stdio.h>
#include<malloc.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
void LSPB_main(char *sf,char *tf);
void CreateLSPB(int Block_Size);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
FindPrime();
start=clock();
LSPB_main(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}
void CreateLSPB(int Block_Size)
{
int i, j, p_count, np_count, CP[513], CNP[513];
for(i=1; i<=512;)
{
for(j=1, p_count=np_count=0; j<=Block_Size; j++)
{
if (PrimeBuff[j]==1) /* if (PrimeBuff[i+j]==1) */
CP[++p_count]=array_512[i+j-1];
else
CNP[++np_count]=array_512[i+j-1];
}
for(j=1; j<=p_count; j++)
array_512[i++]=CP[j];
for(j=1; j<=np_count; j++)
array_512[i++]=CNP[j];
}
}
void LSPB_main(char *sf,char *tf)
{
long out_loop,remender;
unsigned long out_counter;
int fill_array, Block_Size=8;
FILE *fs,*ft;

```

```

fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
for(Block_Size=8;Block_Size<=512; Block_Size*=2)
CreateLSPB(Block_Size);
j=1; // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
if(remender!=0)
{
file_padding(out_loop,remender,sf);//Fill array_512 with data and pad
for(Block_Size=8;Block_Size<=512; Block_Size*=2)
CreateLSPB(Block_Size);
j=1; // write to Destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
padding_information(pad);//Fill array info_pad_1[8] and info_pad_2[8]
b=ascii_pad(pad_info_1);//convert info_pad_1[8] to equivalent ascii
printf(" \npad_info_1=%d",b);
putc(b,ft);// write to Destination file
b=ascii_pad(pad_info_2);//convert info_pad_2[8] to equivalent ascii
printf(" \npad_info_2=%d",b);
putc(b,ft);// write to destination file
}
fclose(ft);
fclose(fs);
}

```

3. Source code for RSPB encryption/decryption

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
void RSPB_main(char *sf,char *tf);
void CreateRSPB(int Block_Size);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
FindPrime();
start=clock();
RSPB_main(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}
/*-----*/
void CreateRSPB(int Block_Size)
{
int i, j, p_count, np_count, CP[513], CNP[513];
for(i=1; i<=512;)
{
for(j=1, p_count=np_count=0; j<=Block_Size; j++)
{
if (PrimeBuff[j]==1) /* if (PrimeBuff[i+j]==1) */
CP[++p_count]=array_512[i+j-1];
else
CNP[++np_count]=array_512[i+j-1];
}
for(j=1; j<=np_count; j++)
array_512[i++]=CNP[j];
for(j=1; j<=p_count; j++)
array_512[i++]=CP[j];
}
}
/*-----*/
void RSPB_main(char *sf,char *tf)
{
long out_loop,remender;
unsigned long out_counter;
int fill_array, Block_Size;

```

```

FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
for(Block_Size=8;Block_Size<=512; Block_Size*=2)
CreateRSPB(Block_Size);
j=1;          // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
if(remender!=0)
{
file_padding(out_loop,remender,sf);//Fill array_512 with data and pad
for(Block_Size=8;Block_Size<=512; Block_Size*=2)
CreateRSPB(Block_Size);
j=1;          // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
padding_information(pad);//Fill array info_pad_1[8] and info_pad_2[8]
b=ascii_pad(pad_info_1);//convert info_pad_1[8] to equivalent ascii
printf(" \npad_info_1=%d",b);
putc(b,ft);// write on Destination file
b=ascii_pad(pad_info_2);//convert info_pad_2[8] to equivalent ascii
printf(" \npad_info_2=%d",b);
putc(b,ft);// write to destination file
}
fclose(ft);
fclose(fs);
}

```

4. Source code for DSPB encryption/decryption

```

#include<stdio.h>
#include<malloc.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
void DSPB_main(char *sf,char *tf);
void CreateDSPB(int Block_Size);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
FindPrime();
start=clock();
DSPB_main(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}
void CreateDSPB(int Block_Size)
{
int i, j, p_count, np_count, CP[512], CNP[512];
for(i=1; i<=512;)
{
for(j=1, p_count=np_count=0; j<=Block_Size; j++)
{
if (PrimeBuff[j]==1) /* if (PrimeBuff[i+j]==1) */
CP[++p_count]=array_512[i+j-1];
else
CNP[++np_count]=array_512[i+j-1];
}
for(j=1; j<=p_count/2; j++)
array_512[i++]=CP[j];
for(j=1; j<=np_count; j++)
array_512[i++]=CNP[j];
for(j=p_count/2+1; j<=p_count; j++)
array_512[i++]=CP[j];
} }
void DSPB_main(char *sf,char *tf)
{
long out_loop,remender;
unsigned long out_counter;
int fill_array, Block_Size;

```

```

FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
for(Block_Size=8;Block_Size<=512; Block_Size*=2)
CreateDSPB(Block_Size);
j=1;          // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
if(remender!=0)
{
file_padding(out_loop,remender,sf);//Fill array_512 with data and pad
for(Block_Size=8;Block_Size<=512; Block_Size*=2)
CreateDSPB(Block_Size);
j=1;          // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
padding_information(pad);//Fill array info_pad_1[8] and info_pad_2[8]
b=ascii_pad(pad_info_1);//convert info_pad_1[8] to equivalent ascii
printf(" \npad_info_1=%d",b);
putc(b,ft);// write to destination file
b=ascii_pad(pad_info_2);//convert info_pad_2[8] to equivalent ascii
printf(" \npad_info_2=%d",b);
putc(b,ft);// write to destination file
}
fclose(ft);
fclose(fs);
}

```

5. Source code for CSPB encryption/decryption

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include "encrypt.h"
void DSPB_main(char *sf,char *tf);
void CreateRSPB(int);
void CreateLSPB(int);
void CreateDSPB(int);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
FindPrime();
start=clock();
DSPB_main(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}
void CreateLSPB(int Block_Size)
{
int i,j,p_count,np_count,CP[512],CNP[512];
for(i=1;i<=512;)
{
for(j=1,p_count=np_count=0;j<=Block_Size;j++)
{
if (PrimeBuff[j]==1) /* if (PrimeBuff[i+j]==1) */
CP[++p_count]=array_512[i+j-1];
else
CNP[++np_count]=array_512[i+j-1];
}
for(j=1;j<=p_count;j++)
array_512[i++]=CP[j];
for(j=1;j<=np_count;j++)
array_512[i++]=CNP[j];
}
}
void CreateRSPB(int Block_Size)
{
int i,j,p_count,np_count,CP[512],CNP[512];
for(i=1;i<=512;)

```

```

{
for(j=1, p_count=np_count=0; j<=Block_Size; j++)
{
if (PrimeBuff[j]==1) /* if (PrimeBuff[i+j]==1) */
CP[+p_count]=array_512[i+j-1];
else
CNP[+np_count]=array_512[i+j-1];
}
for(j=1; j<=np_count; j++)
array_512[i++]=CNP[j];
for(j=1; j<=p_count; j++)
array_512[i++]=CP[j];
}
}
void CreateDSPB(int Block_Size)
{
int i, j, p_count, np_count;
int CP[512], CNP[512];
for(i=1; i<=512;)
{
for(j=1, p_count=np_count=0; j<=Block_Size; j++)
{
if (PrimeBuff[j]==1) /* if (PrimeBuff[i+j]==1) */
CP[+p_count]=array_512[i+j-1];
else
CNP[+np_count]=array_512[i+j-1];
}
for(j=1; j<=p_count/2; j++)
array_512[i++]=CP[j];
for(j=1; j<=np_count; j++)
array_512[i++]=CNP[j];
for(j=p_count/2+1; j<=p_count; j++)
array_512[i++]=CP[j];
}
}
void DSPB_main(char *sf,char *tf)
{
long out_loop,remender;
unsigned long out_counter;
int fill_array, Block_Size;
FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;

```

```

fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
for(Block_Size=8;Block_Size<=512; Block_Size*=2)
{
CreateLSPB(Block_Size);
CreateRSPB(Block_Size);
CreateDSPB(Block_Size);
}
j=1;          // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
if(remender!=0)
{
file_padding(out_loop,remender,sf);//Fill array_512 with data and pad
for(Block_Size=8;Block_Size<=512; Block_Size*=2)
{
CreateLSPB(Block_Size);
CreateRSPB(Block_Size);
CreateDSPB(Block_Size);
}
j=1;          // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
padding_information(pad);//Fill array info_pad_1[8] and info_pad_2[8]
b=ascii_pad(pad_info_1);//convert info_pad_1[8] to equivalent ascii
printf(" \npad_info_1=%d",b);
putc(b,ft);// write on Destination file
b=ascii_pad(pad_info_2);//convert info_pad_2[8] to equivalent ascii
printf(" \npad_info_2=%d",b);
putc(b,ft);// write to destination file
}
fclose(ft);
fclose(fs);
}

```

6. Source code for BET encryption/decryption

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
void encrypt_512();
void bet_encryption(char *sf,char *tf);
int array_temp[513];
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
clrscr();
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
start=clock();
bet_encryption(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}

/*-----*/
void encrypt_512()
{
int block_size=2;
int block_end,loop_end;
int loop;
iteration:
block_size=block_size*2;
loop_end=512/block_size;
int n=block_size;
int enop=n/4;
int aptr=1,atemp=1,k;
for(loop=0;loop<loop_end;loop++)
{
k=aptr+enop;
for(int j=aptr;j<k;j++)
{
array_temp[atemp]=array_512[aptr];
aptr++;
atemp++;
}
aptr=aptr+enop;
}
}

```

```

k=aptr+enop;
for( j=aptr;j<k;j++)
array_temp[atemp++]=array_512[aptr++];
aptr=aptr-(2*enop);
k=aptr+enop;
for( j=aptr;j<k;j++)
array_temp[atemp++]=array_512[aptr++];
aptr=aptr+enop;
k=aptr+enop;
for( j=aptr;j<k;j++)
array_temp[atemp++]=array_512[aptr++];
}
for(i=1;i<513;i++)
array_512[i]=array_temp[i];
if (block_size<257)
goto iteration;
}

/*_____*/
void bet_encryption(char *sf,char *tf)
{
long out_loop,remender;
unsigned long out_counter;
int fill_array;
FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
encrypt_512();
j=1; // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}

```

```

}
if(remender!=0)
{
file_padding(out_loop,remender,sf);//Fill array_512 with data and pad
/*_____*/
encrypt_512();
/*_____*/
j=1;          // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
/*_____*/
padding_information (pad);//Fill array info_pad_1[8] and info_pad_2[8]
/*_____*/
b=ascii_pad(pad_info_1);//convert info_pad_1[8] to equivalent ascii
printf(" \npad_info_1=%d",b);
putc(b,ft);// write to destination file
/*_____*/
b=ascii_pad(pad_info_2);//convert info_pad_2[8] to equivalent ascii
printf(" \npad_info_2=%d",b);
putc(b,ft);// write to destination file
}
fclose(ft);
fclose(fs);
}

```

7. Source code for SPOB encryption/decryption

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
void encrypt_512(int);
void swapbit_encryption(char *sf,char *tf);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
start=clock();
swapbit_encryption(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}
void encrypt_512(int gap)
{
int i, j, t;
for(i=1; i<=512; i=j+gap+1)
{
for(j=i; j<i+8-(gap+1); j++)
{
t=array_512[j];
array_512[j]=array_512[j+gap+1];
array_512[j+gap+1]=t;
}
}
}
void swapbit_encryption(char *sf,char *tf)
{
long out_loop,remainder;
unsigned long out_counter;
int fill_array,i, gap
FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remainder=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)

```

```

{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
for(i=8;i<=512; i*=2)
{
for(gap=1; gap<=i-2; gap++)
encrypt_512(gap);
}
j=1;          // write on Destination file
for(loop=0;loop<64;loop++)
{
b=ascii(j);
j=j+8;
putc(b,ft);
}
}
if(remender!=0)
{
file_padding(out_loop,remender,sf);//Fill array_512 with data and pad
for(i=8;i<=512; i*=2)
{
for(gap=1; gap<=i-2; gap++)
encrypt_512(gap);
}
j=1;          // write on Destination file
for(loop=0;loop<64;loop++)
{
b=ascii(j);
j=j+8;
putc(b,ft);
}
padding_information(pad);//Fill array info_pad_1[8] and info_pad_2[8]
b=ascii_pad(pad_info_1);//conver info_pad_1[8] to eque ascii
printf(" \npad_info_1=%d",b);
putc(b,ft);// write on Destination file
b=ascii_pad(pad_info_2);//conver info_pad_2[8] to eque ascii
printf(" \npad_info_2=%d",b);
putc(b,ft);// write on Destination file
}
fclose(ft);
fclose(fs);
}

```

8. Source code for MAT encryption

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
void encrypt_512();
void mat_encryption(char *sf,char *tf);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
start=clock();
mat_encryption(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}
/*_____*/
void encrypt_512()
{
int block_size=4,block_end,loop_end, loop;
iteration:
block_size=block_size*2;
array_ptr=block_size;
block_end=(block_size-1);
loop_end=512/(block_size*2);
for(loop=0;loop<loop_end;loop++)
{
array_ptr=bits_addition(block_end,array_ptr,block_size);
array_ptr=array_ptr+(block_size*3);
}
if (block_size<257)
goto iteration;
}
/*_____*/
void mat_encryption(char *sf,char *tf)
{
long out_loop,remender;
unsigned long out_counter;
int fill_array;
FILE *fs,*ft;
fs=fopen(sf,"rb");

```

```

ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
printf("\nthe out_loop=%ld",out_loop);
printf("\nthe rem=%ld",remender);
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
encrypt_512();
j=1;
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
if(remender!=0)
{
file_padding(out_loop,remender,sf);
encrypt_512();
j=1; // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
padding_information(pad);//Fill array info_pad_1[8] and info_pad_2[8]
b=ascii_pad(pad_info_1);//convert info_pad_1[8] to equivalent ascii
printf(" \npad_info_1=%d",b);
putc(b,ft);// write to destination file
b=ascii_pad(pad_info_2);//convert info_pad_2[8] to equivalent ascii
printf(" \npad_info_2=%d",b);
putc(b,ft);// write to destination file
}
fclose(ft);
fclose(fs);
}

```

9. Source code for MAT decryption

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
void dencrypt_512();
void mat_dencryption(char *,char *);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
clrscr();
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
file2=tfile;
start=clock();
mat_dencryption(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}

/*-----*/
void mat_dencryption(char *sf,char *tf)
{
long out_loop,remainder;
unsigned long out_counter;
int fill_array;
FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remainder=size%64;
printf("\nthe out_loop=%ld",out_loop);
printf("\nthe rem=%ld",remainder);
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);

```

```

binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
dencrypt_512();
j=1;          // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
fclose(ft);
fclose(fs);
if(remender!=0)
{
delete_padding(sf,tf);
}
}

/*-----*/
void dencrypt_512()
{
int block_size=512;
int carry,block_end,loop_end;
iteration:
block_size=block_size/2;
array_ptr=block_size;
block_end=(block_size-1);
loop_end=512/(block_size*2);
for(loop=0;loop<loop_end;loop++)
{
array_ptr=bits_substraction(block_end,array_ptr,block_size);
array_ptr=array_ptr+(block_size*3);
}
if (block_size>15)
goto iteration;
}

```

10. Source code for OMAT encryption

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
void encrypt_512();
void omat_encryption(char *sf,char *tf);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
clrscr();
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
start=clock();
omat_encryption(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}
/*_____*/
void encrypt_512()
{
int block_size=4;
iteration:
block_size=block_size*2;
array_ptr=block_size;
int block_end=(block_size-1);
int loop_end=(512/(block_size))-1;

for(int loop=0;loop<loop_end;loop++)
{
array_ptr=bits_addition(block_end,array_ptr,block_size);
array_ptr=array_ptr+(block_size*2);
}
if(block_size<257)
goto iteration;
}
/*_____*/
void omat_encryption(char *sf,char *tf)
{
long out_loop,remender;
unsigned long out_counter;
int fill_array;

```

```

FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
int fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
encrypt_512();
j=1; // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
if(remender!=0)
{
file_padding(out_loop,remender,sf);//Fill array_512 with data and pad
encrypt_512();
j=1; // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
padding_information(pad);//Fill array info_pad_1[8] and info_pad_2[8]
b=ascii_pad(pad_info_1);//convert info_pad_1[8] to equivalent ascii
printf(" \npad_info_1=%d",b);
putc(b,ft);// write on Destination file
b=ascii_pad(pad_info_2);//convert info_pad_2[8] to equivalent ascii
printf(" \npad_info_2=%d",b);
putc(b,ft);// write to destination file
}
fclose(ft);
fclose(fs);
}

```

11. Source code for OMAT decryption

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
void dencrypt_512();
void omat_dencryption(char *sf,char *tf);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
clrscr();
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
file2=tfile;
start=clock();
omat_dencryption(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}

/*-----*/
void omat_dencryption(char *sf,char *tf)
{
long out_loop,remender;
unsigned long out_counter;
int fill_array;
FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)

```

```

array_512[array_ptr++]=x[i];
fill_array--;
}
dencrypt_512();
j=1; // write on Destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
fclose(ft);
fclose(fs);
if(remainder!=0)
delete_padding(sf,tf);
}

/*-----*/
void dencrypt_512()
{
int block_size=512;
iteration:
int carry;
block_size=block_size/2;
array_ptr=512-block_size;
int block_end=(block_size-1);
int loop_end=(512/block_size)-1;
for(int loop=0;loop<loop_end;loop++)
{
array_ptr=bits_substraction(block_end,array_ptr,block_size);
}
if (block_size>15)
goto iteration;
}

```

12. Source code for MMAT encryption

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
void encrypt_512_two();
void mat_encryption_two(char *sf,char *tf);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
clrscr();
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
start=clock();
mat_encryption_two(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}
void encrypt_512_two()
{
int block_size=4,block_end,loop_end, loop;
iteration:
block_size=block_size*2;
array_ptr=block_size;
block_end=(block_size-1);
loop_end=512/(block_size*2);
for(loop=0;loop<loop_end;loop++)
{
array_ptr=bits_addition(block_end,array_ptr,block_size);
array_ptr=array_ptr+(block_size*3);
}
array_ptr=block_size;
for(loop=0;loop<loop_end;loop++)
{
array_ptr=bits_addition_two(block_end,array_ptr,block_size);
array_ptr=array_ptr+(block_size*3);
}
if (block_size<257)
goto iteration;
}
void mat_encryption_two(char *sf,char *tf)
{
long out_loop,remender;

```

```

unsigned long out_counter;
int fill_array;
FILE *fs, *ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
encrypt_512_two();
j=1;          // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
if(remender!=0)
{
file_padding(out_loop,remender,sf);//Fill array_512 with data and pad
encrypt_512_two();
j=1;          // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
padding_information(pad);//Fill array info_pad_1[8] and info_pad_2[8]
b=ascii_pad(pad_info_1);//convert info_pad_1[8] to equivalent ascii
printf(" \npad_info_1=%d",b);
putc(b,ft);// write on Destination file
b=ascii_pad(pad_info_2);//convert info_pad_2[8] to equivalent ascii
printf(" \npad_info_2=%d",b);
putc(b,ft);// write to destination file
}
fclose(ft);
fclose(fs);
}

```

13. Source code for MMAT decryption

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
void dencrypt_512_two();
void mat_dencryption_two(char *,char *);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
clrscr();
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
file2=tfile;
start=clock();
mat_dencryption_two(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}
/*_____*/
void mat_dencryption_two(char *sf,char *tf)
{
long out_loop,remender;
unsigned long out_counter;
int fill_array;
FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
}
}
}

```

```

fill_array--;
}
dencrypt_512_two();
j=1; // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
fclose(ft);
fclose(fs);
if(remender!=0)
delete_padding(sf,tf);
}
/*-----*/
void dencrypt_512_two()
{
int block_size=512;
int carry,block_end,loop_end;
iteration:
block_size=block_size/2;
array_ptr=block_size;
block_end=(block_size-1);
loop_end=512/(block_size*2);
for(loop=0;loop<loop_end;loop++)
{
array_ptr=bits_substraction_two(block_end,array_ptr,block_size);
array_ptr=array_ptr+(block_size*3);
}
array_ptr=block_size;
for(loop=0;loop<loop_end;loop++)
{
array_ptr=bits_substraction(block_end,array_ptr,block_size);
array_ptr=array_ptr+(block_size*3);
}
if (block_size>15)
goto iteration;
}

```

14. Source code for BOS encryption/decryption

```

#include<stdio.h>
#include<conio.h>
#include<time.h>
#include"encrypt.h"
int array[513];
void encrypt_512(int block_size);
void bos_encryption(char *sf,char *tf);

void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
clrscr();
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
start=clock();
bos_encryption(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}

/*-----*/
void encrypt_512(int block_size)
{
int ptr=1,i,j,k;
int pos1,pos2;
int out_loop=512/block_size;

for(i=0;i<out_loop;i++)
{
for(j=0;j<(block_size/8);j++)
{
pos1=(ptr+1)/2;
pos2=((ptr+1)/2)+(block_size/2);
/*-----*/
if(array_512[ptr]==array_512[ptr+1])
array[pos1]=1;
else
array[pos1]=0;
/*-----*/
if(array_512[ptr]==0)
array[pos2]=0;
}
}
}

```

```

if(array_512[ptr]==1)
array[pos2]=1;
ptr=ptr+2;
}
}
for(i=1;i<513;i++)
array_512[i]=array[i];
}

/*-----*/
void bos_encryption(char *sf,char *tf)
{
int gap,block_size=8;
long out_loop,remender;
unsigned long out_counter;
int fill_array,i;
FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
for(i=block_size;i<=512; i*=2)
{
encrypt_512(i);
}
j=1; // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
if(remender!=0)
{
file_padding(out_loop,remender,sf);//Fill array_512 with data and pad

```

```
for(i=block_size;i<=512;i*=2)
{
encrypt_512(i);
}
j=1;          // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
padding_information(pad);//Fill array info_pad_1[8] and info_pad_2[8]
b=ascii_pad(pad_info_1);// convert info_pad_2[8] to equivalent ascii
printf(" \npad_info_1=%d",b);
putc(b,ft);// write on Destination file
b=ascii_pad(pad_info_2);//convert info_pad_2[8] to equivalent ascii
printf(" \npad_info_2=%d",b);
putc(b,ft);// write to destination file
}
fclose(ft);
fclose(fs);
}
```

15. Source code for DEPS encryption

```

#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<malloc.h>
#include<time.h>
#include"encrypt.h"

/*-----*/
void deps_encryption(char *sf,char *tf);
void encrypt_512(int block_size);
/*-----*/

void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
clrscr();
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
start=clock();
deps_encryption(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}

/*-----*/

void deps_encryption(char *sf,char *tf)
{
long out_loop,remainder;
unsigned long out_counter;
int fill_array;
FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remainder=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;

```

```

fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
for(int Block_Size=8;Block_Size<=512; Block_Size*=2)
encrypt_512(Block_Size);
j=1; // write to destination file
for(int lop=0;lop<64;lop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
}
if(remender!=0)
{
file_padding(out_loop,remender,sf);//Fill array_512 with data and pad
/*_____*/
for(int Block_Size=8;Block_Size<=512; Block_Size*=2)
encrypt_512(Block_Size);
/*_____*/
j=1; // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);
}
/*_____*/
padding_information (pad);//Fill array info_pad_1[8] and info_pad_2[8]
/*_____*/
b=ascii_pad(pad_info_1);//convert info_pad_1[8] to equivalent ascii
printf(" \npad_info_1=%d",b);
putc(b,ft);// write to destination file
/*_____*/
b=ascii_pad(pad_info_2);//convert info_pad_2[8] to equivalent ascii
printf(" \npad_info_2=%d",b);
putc(b,ft);// write to destination file
}
fclose(ft);
fclose(fs);
}

/*_____*/
void encrypt_512(int block_size)
{
int array_ptr=1;

```

```
int D=block_size;
int carry=0;
int *B;
int out_loop,loop_end=512/(block_size);
for(out_loop=0;out_loop<loop_end;out_loop++)
{
B=(int *)malloc(sizeof(int)*(block_size+1));
for(i=0;i<=block_size;i++)
*(B+i)=0;
for(int lop=0;lop<D;lop++)
{
carry=0;
carry=Rotate_Right_CY(array_ptr,block_size,carry);
if(carry==0) {goto lb;}
INR(array_ptr,block_size);
lb:
carry=Rotate_Left_CY_B(B,array_ptr,block_size,carry);
}
for(int i=array_ptr;i<array_ptr+block_size;i++)
array_512[i]=*(B+i);
free(B);
array_ptr=array_ptr+block_size;
}
}
```

16. Source code for DEPS decryption

```

#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<malloc.h>
#include<time.h>
#include"encrypt.h"
void deps_decryption(char *sf,char *tf);
void dencrypt_512(int block_size);
void main()
{
char *sfile,*tfile;
clock_t start,end;
float time;
printf("Enter source file name :");
scanf("%s",sfile);
printf("\nEnter target file name :");
scanf("%s",tfile);
file2=tfile;
start=clock();
deps_decryption(sfile,tfile);
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("\n Time = %f",time);
getch();
}
void dencrypt_512(int block_size)
{
int array_ptr=1, D=block_size, carry=0, *B;
int out_loop,loop_end=512/(block_size);

for(out_loop=0;out_loop<loop_end;out_loop++)
{
B=(int *)malloc(sizeof(int)*(block_size+1));
for(i=0;i<=block_size;i++)
*(B+i)=0;
*(B+block_size)=1;
for( loop=0;loop<D;loop++)
{
carry=0;
carry=Rotate_Right_CY(array_ptr,block_size,carry);
if(carry==0)
{carry=1;} //complement carry
else
{carry=0;}
carry=Rotate_Left_CY_B(B,array_ptr,block_size,carry);
DCR_B(B,array_ptr,block_size);
}
}

```

```

for(int i=array_ptr;i<array_ptr+block_size;i++)
array_512[i]=*(B+i);
free(B);
array_ptr=array_ptr+(block_size);
}
}
void deps_decryption(char *sf,char *tf)
{
long out_loop,remender;
unsigned long out_counter;
int fill_array;
FILE *fs,*ft;
fs=fopen(sf,"rb");
ft=fopen(tf,"wb");
size=filesize(fs);
printf("\n the size=%ld",size);
out_loop=size/64;
remender=size%64;
for(out_counter=0;out_counter<out_loop;out_counter++)
{
array_ptr=1;
fill_array=64;
while(fill_array>0)
{
ch=getc(fs);
binary(ch);
for(i=0;i<8;i++)
array_512[array_ptr++]=x[i];
fill_array--;
}
decrypt_512(256);
decrypt_512(128);
decrypt_512(64);
decrypt_512(32);
decrypt_512(16);
decrypt_512(8);
j=1; // write to destination file
for(loop=0;loop<64;loop++)
{ b=ascii(j);
j=j+8;
putc(b,ft);}
}
fclose(ft);
fclose(fs);
if(remender!=0)
{
delete_padding(sf,tf);
}
}
}

```

17. Source code for Triple DES encryption

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include<conio.h>
#include<time.h>
void getkey(unsigned char [],unsigned char [],unsigned char []);
/* generates 3 keys from key file */
void binkey(unsigned char key[],int dec);
/* convert int 'dec' to its 8 bit binary equivalent */
void encrypt(unsigned char *,unsigned char [16][49]);
void decrypt(unsigned char *,unsigned char [16][49]);
void convtbit(unsigned char [],unsigned char []);
void convtopc1(unsigned char [],int [8][7],unsigned char [],unsigned char []);
void shiftbits(unsigned char [],int);
void getpc2(unsigned char [],int [6][8],unsigned char []); // pc2 table
void getexpbits(unsigned char [],int [8][6],unsigned char []); // E table
void getXOR(unsigned char[],unsigned char[],int); //result will be in second parameter
int dec(long);
void bin(int,unsigned char []);
void getsbskey(unsigned char xorbits[],unsigned char subskey[]);
void getPerm(unsigned char [],int [4][8],unsigned char []);
void permutation(unsigned char [],int [8][8]);
void convchar(unsigned char [],unsigned char *); //2nd param. holds result
void funF(unsigned char [33],unsigned char [49],unsigned char [49]);
//3rd param. holds result

void main()
{
unsigned char *tarr;
char source[40],dest[40];
int i,l,count,tempchar,num;
time_t first,second;
unsigned char key1[65];
unsigned char key2[65];
unsigned char key3[65];
unsigned char ckey[29], dkey[29], keypc1[57], keypc21[16][49],
keypc22[16][49], keypc23[16][49];
int pc1[8][7] = { 57,49,41,33,25,17,09,
                 01,58,50,42,34,26,18,
                 10,02,59,51,43,35,27,
                 19,11,03,60,52,44,36,
                 63,55,47,39,31,23,15,
                 07,62,54,46,38,30,22,
                 14,06,61,53,45,37,29,
                 21,13,05,28,20,12,04
                 };

```

```

int pc2[6][8] = { 14,17,11,24,01,05,03,28,
                 15,06,21,10,23,19,12,04,
                 26,08,16,07,27,20,13,02,
                 41,52,31,37,47,55,30,40,
                 51,45,33,48,44,49,39,56,
                 34,53,46,42,50,36,29,32    };
int lshift[16]={1,1,2,2,2,2,2,2,1,2,2,2,2,2,1};
FILE *fp, *fc; /* for file */
int iter;
unsigned char ch;
if((tarr=(unsigned char *)malloc(8))==NULL)
{
printf("\n Not enough memory(ptrtemp)");
getch();
exit(0);
}
num=0; /*to make constraint number of times you can enter file names in case of mistakes */
start_en:
num++;
printf("\n\n Enter The Source File : ");
scanf("%s",source);
printf("\n Enter The Destination File : ");
scanf("%s",dest);
fp=fopen(source,"rb");
if(fp==NULL)
{
printf("\n cannot open file %s",source);
getch();
if(num==3)
exit(0);
else
goto start_en;
}
fc=fopen(dest,"wb");
if(fc==NULL)
{
printf("\n cannot open file %s",dest);
getch();
if(num==3)
exit(0);
else
goto start_en;
}
/* stop processing when source file don't contain any data */
fseek(fp,0,2);
if(ftell(fp)==0)
{
printf("\n Error: %s file must contain some data\n",source);
getch();
exit(0); }

```

```

rewind(fp);
/* key generation */
getkey(key1,key2,key3); // as result key1,key2,key3 will contain data
/* convert key1 to 56 bits dividing ckey and dkey by permuted choice-1 */
convtopc1(key1,pc1,ckey,dkey);
for(i=0;i<16;i++)
{
shiftbits(ckey,lshift[i]);
shiftbits(dkey,lshift[i]);
strcpy(keypc1,ckey);
strcat(keypc1,dkey);
getpc2(keypc1,pc2,keypc21[i]); /*calling to get permuted choice-2*/
}
/* convert key2 to 56 bits dividing ckey and dkey by permuted choice-1 */
convtopc1(key2,pc1,ckey,dkey);
for(i=0;i<16;i++)
{
shiftbits(ckey,lshift[i]);
shiftbits(dkey,lshift[i]);
strcpy(keypc1,ckey);
strcat(keypc1,dkey);
getpc2(keypc1,pc2,keypc22[i]); /*calling to get permuted choice-2*/
}
/* convert key3 to 56 bits dividing ckey and dkey by permuted choice-1 */
convtopc1(key3,pc1,ckey,dkey);
for(i=0;i<16;i++)
{
shiftbits(ckey,lshift[i]);
shiftbits(dkey,lshift[i]);
strcpy(keypc1,ckey);
strcat(keypc1,dkey);
/*calling to get permuted choice-2*/
getpc2(keypc1,pc2,keypc23[i]);
}
/* key generation terminates */
first=time(NULL);
count=0;
do
{
ch=fgetc(fp);
if(!feof(fp))
{
if(count<8 && count>=0)
{
tempchar=0;
for(l=count;l<8;l++)
{
tarr[l]=48+l;
tempchar++;
}
}
}
}

```

```

tarr[7]=48+tempchar;
tarr[8]='\0';
encrypt(tarr,keypc21); // tarr will contain the result
decrypt(tarr,keypc22);
encrypt(tarr,keypc23);
for(iter=0;iter<8;iter++)
{
ch=tarr[iter];
fputc(ch,fc);
}
}
break;
}
else
{
tarr[count]=ch;
count++;
if(count==8)
{
tarr[8]='\0';
encrypt(tarr,keypc21); // tarr will contain the result
decrypt(tarr,keypc22);
encrypt(tarr,keypc23);
for(iter=0;iter<8;iter++)
{
ch=tarr[iter];
fputc(ch,fc);
}
count=0;
}
}
}while(!feof(fp));
fclose(fp);
fclose(fc);
second=time(NULL);
printf("\n Encryption Time =%f seconds ",difftime(second,first));
getch();
}

void getkey(unsigned char key1[],unsigned char key2[],unsigned char key3[])
{
FILE *fp;
unsigned char ch;
char keyfile[40];
int i;
strcpy(key1,"");
strcpy(key2,"");
strcpy(key3,"");
printf("\n Enter the Key file (it should contain 192 characters) : ");
scanf("%s",keyfile);

```

```

fp=fopen(keyfile,"rb");
if(fp==NULL)
{
printf("\n Error opening file\n");
getch();
exit(1);
}
i=1;
while(!feof(fp) && i<=8)
{
ch=fgetc(fp);
binkey(key1,ch);
i++;
}
while(i<=8)
{
binkey(key1, ' ');
i++;
}
i=1;
while(!feof(fp) && i<=8)
{
ch=fgetc(fp);
binkey(key2,ch);
i++;
}
while(i<=8)
{
binkey(key2, ' ');
i++;
}
i=1;
while(!feof(fp) && i<=8)
{
ch=fgetc(fp);
binkey(key3,ch);
i++;
}
while(i<=8)
{
binkey(key3, ' ');
i++;
}
fclose(fp);
}

void binkey(unsigned char key[],int dec)
{
unsigned char tempbin[9],bin[9],tbin[2];
int j,k,num;

```

```

strcpy(bin,"");
for(j=0;j<8;j++)
tempbin[j]=0;
num=0;
num=dec;
j=7;
do
{
tempbin[j]=num%2;
num/=2;
j--;
} while(num!=0);
for(k=0;k<8;k++)
{
itoa(tempbin[k],tbin,10); //converting integer to character
if(k==0)
strcpy(bin,tbin);
else
strcat(bin,tbin);
}
bin[8]='\0';
if(strlen(key)==0)
strcpy(key,bin);
else
strcat(key,bin);
}

void encrypt(unsigned char *tarr,unsigned char keypc2[16][49])
{
unsigned char pbits[65]="0",tempbits[65];
unsigned char leftbits[33],rightbits[33],templeft[49],tempright[49]="0";
int i;
int ip[8][8] = { 58,50,42,34,26,18,10,02,
                 60,52,44,36,28,20,12,04,
                 62,54,46,38,30,22,14,06,
                 64,56,48,40,32,24,16,08,
                 57,49,41,33,25,17,09,01,
                 59,51,43,35,27,19,11,03,
                 61,53,45,37,29,21,13,05,
                 63,55,47,39,31,23,15,07           };
int ipinv[8][8]={ 40,08,48,16,56,24,64,32,
                  39,07,47,15,55,23,63,31,
                  38,06,46,14,54,22,62,30,
                  37,05,45,13,53,21,61,29,
                  36,04,44,12,52,20,60,28,
                  35,03,43,11,51,19,59,27,
                  34,02,42,10,50,18,58,26,
                  33,01,41,09,49,17,57,25           };
convtbit(tarr,pbits); // converts string tarr to ascii code in pbits
permutation(pbits,ip); /* performing initial permutation*/

```

```

for(i=0;i<32;i++)
{
leftbits[i]=pbits[i];
rightbits[i]=pbits[32+i];
}
leftbits[32]='\0';
rightbits[32]='\0';
/* Starting rounds */
for(i=0;i<16;i++)
{
strcpy(tempright,"");
/* implement function F with rightbits */
funF(rightbits,keypc2[i],tempright); //result is in tempright
/*for next iteration left and right bits are */
strcpy(templeft,leftbits);
strcpy(leftbits,rightbits); //L(i)=R(i-1)
getXOR(templeft,tempright,32); //result in tempright
strcpy(rightbits,tempright); //R(i)=L(i-1) XOR F
}
/* Performing 32-bit swapping */
strcpy(tempbits,rightbits);
strcat(tempbits,leftbits);
/* Perform inverse initial permutation */
permutation(tempbits,ipinv);
convchar(tempbits,tarr);
}

void decrypt(unsigned char *tarr,unsigned char keypc2[16][49])
{
unsigned char cbits[65],tempbits[65];
unsigned char leftbits[33],rightbits[33],templeft[49],tempright[49];
int i;
int ip[8][8]={ 58,50,42,34,26,18,10,02,
60,52,44,36,28,20,12,04,
62,54,46,38,30,22,14,06,
64,56,48,40,32,24,16,08,
57,49,41,33,25,17,09,01,
59,51,43,35,27,19,11,03,
61,53,45,37,29,21,13,05,
63,55,47,39,31,23,15,07
};
int ipinv[8][8]={ 40,08,48,16,56,24,64,32,
39,07,47,15,55,23,63,31,
38,06,46,14,54,22,62,30,
37,05,45,13,53,21,61,29,
36,04,44,12,52,20,60,28,
35,03,43,11,51,19,59,27,
34,02,42,10,50,18,58,26,
33,01,41,09,49,17,57,25
};
convtbit(tarr,cbits); /* decrypting the cyphertext */
permutation(cbits,ip); /* Performing initial permutation */

```

```

for(i=0;i<32;i++)
{
leftbits[i]=cbits[i];
rightbits[i]=cbits[32+i];
}
leftbits[32]='\0';
rightbits[32]='\0';
/* Starting rounds */
for(i=15;i>=0;i--)
{
funF(rightbits,keypc2[i],tempright); //result is in tempright
/*for next iteration left and right bits are */
strcpy(templeft,leftbits);
strcpy(leftbits,rightbits); //L(i)=R(i-1)
getXOR(templeft,tempright,32); //result in tempright
strcpy(rightbits,tempright); //R(i)=L(i-1) XOR F
}
/* Performing 32-bit swapping */
strcpy(tempbits,rightbits);
strcat(tempbits,leftbits);
permutation(tempbits,ipinv); /* Perform inverse initial permutation */
convchar(tempbits,tarr);
}

```

```

void convtbit(unsigned char tarr[],unsigned char pbits[])
{
int tempbin[9];
unsigned char bin[9],tbin[2];
int i,j,k,num;
strcpy(pbits,"");
for(i=0;i<8;i++)
{
strcpy(bin,"");
for(j=0;j<8;j++)
tempbin[j]=0;
num=0;
num=tarr[i];
j=7;
do
{
tempbin[j]=num%2;
num/=2;
j--;
} while(num!=0);
for(k=0;k<8;k++)
{
itoa(tempbin[k],tbin,10); //converting integer to character
if(k==0)
strcpy(bin,tbin);
else

```

```

strcat(bin,tbin);
}
bin[8]='\0';
if(i==0)
strcpy(pbits,bin);
else
strcat(pbits,bin);
}
}

```

```

void convtopc1(unsigned char key[65],int pc1[8][7],unsigned char ckey[],unsigned char dkey[])
/* convert 64 bit key to 56 bit key by permuted choice-1 */
{
int i,j,k=0,l=0;
for(i=0;i<8;i++)
{
for(j=0;j<7;j++)
{
if(i<4)
{
ckey[k]=key[(pc1[i][j] - 1)];
k++;
}
else
{
dkey[l]=key[(pc1[i][j]-1)];
l++;
}
}
}
ckey[28]='\0';
dkey[28]='\0';
}

```

```

void shiftbits(unsigned char tkeys[],int n)
{
int i,j;
unsigned char temp;
for(i=1;i<=n;i++)
{
temp=tkeys[0];
for(j=1;j<28;j++)
{
tkeys[j-1]=tkeys[j];
}
tkeys[27]=temp;
}
}

```

```

void getpc2(unsigned char keypc1[],int pc2[6][8],unsigned char keypc2[])
{
int i,j,k;
k=0;
for(i=0;i<6;i++)
{
for(j=0;j<8;j++)
{
keypc2[k]=keypc1[(pc2[i][j]-1)];
k++;
}
}
keypc2[48]='\0';
}

```

```

void getexpbits(unsigned char rightbits[],int exp[8][6],unsigned char tempright[])
{
int i,j,k;
strcpy(tempright,"");
k=0;
for(i=0;i<8;i++)
{
for(j=0;j<6;j++)
{
tempright[k]=rightbits[(exp[i][j]-1)];
k++;
}
}
tempright[48]='\0';
}

```

```

void getXOR(unsigned char keypc2[],unsigned char tempright[],int n)
{
int bit1,bit2,xor,i;
unsigned char ch1[2],ch2[2],temp[49];
for(i=0;i<n;i++)
{
ch1[0]=keypc2[i];
ch2[0]=tempright[i];
ch1[1]='\0';
ch2[1]='\0';
bit1=atoi(ch1);
bit2=atoi(ch2);
xor=bit1+bit2;
if(xor==2)
temp[i]='\0';
else if(xor==0)
temp[i]='\0';
else
temp[i]='1';
}
}

```

```

temp[n]='\0';
strcpy(tempright,temp);
}
int dec(long bin)
{
int i=0,q,dcml=0;
do
{
q=bin%10;
dcml+=q*(int)pow(2,i);
bin/=10;
i++;
}while(bin!=0);
return dcml;
}

```

```

void bin(int dec,unsigned char binstr[])
{
static int tempbin[4];
unsigned char tbin[2]="0";
int i,j,k,flag=0,num;
strcpy(binstr,"");
num=dec;
for(i=0;i<4;i++)
tempbin[i]=0;
j=3;
do
{
tempbin[j]=num%2;
num/=2;
j--;
}while(num!=0);
for(k=0;k<4;k++)
{
itoa(tempbin[k],tbin,10); //converting integer to string(character)
if(flag==0) //for first bit
{
strcpy(binstr,tbin);
flag=1;
}
else
strcat(binstr,tbin);
}
}
}

```

```

void getsubskey(unsigned char xorbits[],unsigned char subskey[])
{
int row,col,temp;
unsigned char tempstr1[3],tempstr2[5],tempright[49];

```

```

int sbox1[4][16]={ 14,04,13,01,02,15,11,08,03,10,06,12,05,09,00,07,
00,15,07,04,14,02,13,01,10,06,12,11,09,05,03,08,
04,01,14,08,13,06,02,11,15,12,09,07,03,10,05,00,
15,12,08,02,04,09,01,07,05,11,03,14,10,00,06,13    };
int sbox2[4][16]={ 15,01,08,14,06,11,03,04,09,07,02,13,12,00,05,10,
03,13,04,07,15,02,08,14,12,00,01,10,06,09,11,05,
00,14,07,11,10,04,13,01,05,08,12,06,09,03,02,15,
13,08,10,01,03,15,04,02,11,06,07,12,00,05,14,09    };
int sbox3[4][16]={ 10,00,09,14,06,03,15,05,01,13,12,07,11,04,02,08,
13,07,00,09,03,04,06,10,02,08,05,14,12,11,15,01,
13,06,04,09,08,15,03,00,11,01,02,12,05,10,14,07,
01,10,13,00,06,09,08,07,04,15,14,03,11,05,02,12    };
int sbox4[4][16]={ 07,13,14,03,00,06,09,10,01,02,08,05,11,12,04,15,
13,08,11,05,06,15,00,03,04,07,02,12,01,10,14,09,
10,06,09,00,12,11,07,13,15,01,03,14,05,02,08,04,
03,15,00,06,10,01,13,08,09,04,05,11,12,07,02,14    };
int sbox5[4][16]={ 02,12,04,01,07,10,11,06,08,05,03,15,13,00,14,09,
14,11,02,12,04,07,13,01,05,00,15,10,03,09,08,06,
04,02,01,11,10,13,07,08,15,09,12,05,06,03,00,14,
11,08,12,07,01,14,02,13,06,15,00,09,10,04,05,03    };
int sbox6[4][16]={ 12,01,10,15,09,02,06,08,00,13,03,04,14,07,05,11,
10,15,04,02,07,12,09,05,06,01,13,14,00,11,03,08,
09,14,15,05,02,08,12,03,07,00,04,10,01,13,11,06,
04,03,02,12,09,05,15,10,11,14,01,07,06,00,08,13    };
int sbox7[4][16]={ 04,11,02,14,15,00,08,13,03,12,09,07,05,10,06,01,
13,00,11,07,04,09,01,10,14,03,05,12,02,15,08,06,
01,04,11,13,12,03,07,14,10,15,06,08,00,05,09,02,
06,11,13,08,01,04,10,07,09,05,00,15,14,02,03,12    };
int sbox8[4][16]={ 13,02,08,04,06,15,11,01,10,09,03,14,05,00,12,07,
01,15,13,08,10,03,07,04,12,05,06,11,00,14,09,02,
07,11,04,01,09,12,14,02,00,06,10,13,15,03,05,08,
02,01,14,07,04,10,08,13,15,12,09,00,03,05,06,11    };

strcpy(temptright,xorbits);/* use s-box1 */
tempstr1[0]=temptright[0];
tempstr1[1]=temptright[5];
tempstr1[2]='\0';
tempstr2[0]=temptright[1];
tempstr2[1]=temptright[2];
tempstr2[2]=temptright[3];
tempstr2[3]=temptright[4];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox1[row][col];
bin(temp,tempstr2); //tempstr2 will hold binary value
strcpy(subskey,tempstr2);/* use s-box2 */
tempstr1[0]=temptright[6];
tempstr1[1]=temptright[11];
tempstr1[2]='\0';
tempstr2[0]=temptright[7];

```

```

tempstr2[1]=tempright[8];
tempstr2[2]=tempright[9];
tempstr2[3]=tempright[10];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox2[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);/* use s-box3 */
tempstr1[0]=tempright[12];
tempstr1[1]=tempright[17];
tempstr1[2]='\0';
tempstr2[0]=tempright[13];
tempstr2[1]=tempright[14];
tempstr2[2]=tempright[15];
tempstr2[3]=tempright[16];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox3[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);/* use s-box4 */
tempstr1[0]=tempright[18];
tempstr1[1]=tempright[23];
tempstr1[2]='\0';
tempstr2[0]=tempright[19];
tempstr2[1]=tempright[20];
tempstr2[2]=tempright[21];
tempstr2[3]=tempright[22];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox4[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);/* use s-box5 */
tempstr1[0]=tempright[24];
tempstr1[1]=tempright[29];
tempstr1[2]='\0';
tempstr2[0]=tempright[25];
tempstr2[1]=tempright[26];
tempstr2[2]=tempright[27];
tempstr2[3]=tempright[28];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox5[row][col];
strcpy(tempstr2,"");

```

```

bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);/* use s-box6 */
tempstr1[0]=tempright[30];
tempstr1[1]=tempright[35];
tempstr1[2]='\0';
tempstr2[0]=tempright[31];
tempstr2[1]=tempright[32];
tempstr2[2]=tempright[33];
tempstr2[3]=tempright[34];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox6[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);/* use s-box7 */
tempstr1[0]=tempright[36];
tempstr1[1]=tempright[41];
tempstr1[2]='\0';
tempstr2[0]=tempright[37];
tempstr2[1]=tempright[38];
tempstr2[2]=tempright[39];
tempstr2[3]=tempright[40];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox7[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);/* use s-box8 */
tempstr1[0]=tempright[42];
tempstr1[1]=tempright[47];
tempstr1[2]='\0';
tempstr2[0]=tempright[43];
tempstr2[1]=tempright[44];
tempstr2[2]=tempright[45];
tempstr2[3]=tempright[46];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox8[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);
}

```

```

void getPerm(unsigned char subskey[],int permF[4][8],unsigned char tempright[])
{
/* performing permutation*/
int i,j,k=0;

```

```

strcpy(tempright,"");
for(i=0;i<4;i++)
{
for(j=0;j<8;j++)
{
tempright[k]=subskey[(permF[i][j]-1)];
k++;
}
}
tempright[32]='\0';
}

```

```

void permutation(unsigned char tbits[],int permbits[8][8])
{
unsigned char tempbits[65];
int i,j,k=0;
for(i=0;i<8;i++)
{
for(j=0;j<8;j++)
{
tempbits[k]=tbits[(permbits[i][j]-1)];
k++;
}
}
tempbits[64]='\0';
strcpy(tbits,tempbits);
}

```

```

void convchar(unsigned char tempbits[],unsigned char *tarr)
{
int j,k=0,num;
unsigned char temp[9],ch;
while(k<8)
{
for(j=0;j<8;j++)
{
temp[j]=tempbits[8*k+j];
}
temp[8]='\0'; //holds 8 bit for a character
num=dec(atol(temp)); //convert binary to decimal value
ch=num;
tarr[k]=ch;
k++;
}
tarr[8]='\0';
}

```

```

void funF(unsigned char rightbits[33],unsigned char keypc2[49],unsigned char tempright[])
/*1st param. holds result*/
{

```

```

unsigned char subskey[33];
int exp[8][6]={      32,01,02,03,04,05, // E table
                    04,05,06,07,08,09,
                    08,09,10,11,12,13,
                    12,13,14,15,16,17,
                    16,17,18,19,20,21,
                    20,21,22,23,24,25,
                    24,25,26,27,28,29,
                    28,29,30,31,32,01    };

int permF[4][8]={    16,07,20,21,29,12,28,17, //P table
                    01,15,23,26,05,18,31,10,
                    02,08,24,14,32,27,03,09,
                    19,13,30,06,22,11,04,25    };

strcpy(tempright,"");

/* expand right 32 bits to make 48 bits */
getexpbits(rightbits,exp,tempright); //result stored in tempright[]

/* XOR beging here */      //result is in tempright[]
getXOR(keypc2,tempright,48);

/* substitution using S-boxes */
getsubskey(tempright,subskey); // result is in subskey 32 bit

/* permutation of the substituted key */
strcpy(tempright,"0");
getPerm(subskey,permF,tempright); //result will be in tempright (32 bit)
}

```

18. Source code for Triple DES decryption

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include<conio.h>
#include<time.h>
void getkey(unsigned char [],unsigned char [],unsigned char []);// generates 3 keys from key file
void binkey(unsigned char key[],int dec); // convert int 'dec' to its 8 bit binary equiv
void encrypt(unsigned char *,unsigned char [16][49]);
void decrypt(unsigned char *,unsigned char [16][49]);
void convtbit(unsigned char [],unsigned char []);
void convtopc1(unsigned char [],int [8][7],unsigned char [],unsigned char []);
void shiftbits(unsigned char [],int);
void getpc2(unsigned char [],int [6][8],unsigned char []); // pc2 table
void getexpbits(unsigned char [],int [8][6],unsigned char []); // E table
void getXOR(unsigned char[],unsigned char[],int); //result will be in second parameter
int dec(long);
void bin(int,unsigned char []);
void getsubskey(unsigned char xorbits[],unsigned char subskey[]);
void getPerm(unsigned char [],int [4][8],unsigned char []);
void permutation(unsigned char [],int [8][8]);
void convchar(unsigned char [],unsigned char *); //2nd param. holds result
void funF(unsigned char [33],unsigned char [49],unsigned char [49]); //3rd param. holds result

void main()
{
unsigned char *tarr;
char source[40],dest[40];
int i,l,count,tempchar,num;
time_t first,second;
unsigned char key1[65],key2[65],key3[65];
unsigned char ckey[29],dkey[29],keypc1[57],keypc21[16][49],keypc22[16][49],keypc23[16][49];
int pc1[8][7] = { 57,49,41,33,25,17,09,
                 01,58,50,42,34,26,18,
                 10,02,59,51,43,35,27,
                 19,11,03,60,52,44,36,
                 63,55,47,39,31,23,15,
                 07,62,54,46,38,30,22,
                 14,06,61,53,45,37,29,
                 21,13,05,28,20,12,04 };
int pc2[6][8] = { 14,17,11,24,01,05,03,28,
                 15,06,21,10,23,19,12,04,
                 26,08,16,07,27,20,13,02,
                 41,52,31,37,47,55,30,40,
                 51,45,33,48,44,49,39,56,
                 34,53,46,42,50,36,29,32 };
int lshift[16]={1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};

```

```

FILE *fc, *fap,*ft; /* for file */
int iter;
unsigned char ch2,ch;
long pos;
if((tarr=(unsigned char *)malloc(8))==NULL)
{
printf("\n Not enough memory(ptrtemp)");
getch();
exit(0);
}
/* Decryption */
num=0; /*to make constraint number of times you can enter
file names in case of mistakes */
start_dc:
num++;
printf("\n\n Enter The Source File : ");
scanf("%s",source);
printf("\n Enter The Destination File : ");
scanf("%s",dest);
fc=fopen(source,"rb");
if(fc==NULL)
{
printf("\n cannot open file %s",source);
getch();
if(num==3)
exit(0);
else
goto start_dc;
}
fap=fopen(dest,"wb");
if(fap==NULL)
{
printf("\n cannot open file %s sanjoy",dest);
getch();
if(num==3)
exit(0);
else
goto start_dc;
}
/* stop processing when source file don't contain any data */
fseek(fc,0,2);
if(ftell(fc)==0)
{
printf("\n Error: %s file must contain some data\n",source);
getch();
exit(0); /* go out of the program */
}
rewind(fc);
/* key generation starts*/
getKey(key1,key2,key3); // as result key1,key2,key3 will contain data

```

```

/* convert key1 to 56 bits dividing ckey and dkey by permuted choice-1 */
convtopc1(key1,pc1,ckey,dkey);
for(i=0;i<16;i++)
{
shiftbits(ckey,lshift[i]);
shiftbits(dkey,lshift[i]);
strcpy(keypc1,ckey);
strcat(keypc1,dkey);
getpc2(keypc1,pc2,keypc21[i]); /*calling to get permuted choice-2*/
}
/* convert key2 to 56 bits dividing ckey and dkey by permuted choice-1 */
convtopc1(key2,pc1,ckey,dkey);
for(i=0;i<16;i++)
{
shiftbits(ckey,lshift[i]);
shiftbits(dkey,lshift[i]);
strcpy(keypc1,ckey);
strcat(keypc1,dkey);
getpc2(keypc1,pc2,keypc22[i]); /*calling to get permuted choice-2*/
}
/* convert key3 to 56 bits dividing ckey and dkey by permuted choice-1 */
convtopc1(key3,pc1,ckey,dkey);
for(i=0;i<16;i++)
{
shiftbits(ckey,lshift[i]);
shiftbits(dkey,lshift[i]);
strcpy(keypc1,ckey);
strcat(keypc1,dkey);
getpc2(keypc1,pc2,keypc23[i]); /*calling to get permuted choice-2*/
} /* key generation terminates */
first=time(NULL);
count=0;
do
{
ch2=fgetc(fc);
if(!feof(fc))
{
if(count<8 && count>0)
{
tempchar=0;
for(l=count;l<8;l++)
{
tarr[l]=48+l;
tempchar++;
}
tarr[7]=48+tempchar;
tarr[8]='\0';
decrypt(tarr,keypc23); //tarr will contain the result
encrypt(tarr,keypc22);
decrypt(tarr,keypc21);
}
}
}

```

```

for(iter=0;iter<8;iter++)
{
ch=tarr[iter];
fputc(ch,fap);
}
}
break;
}
else
{
tarr[count]=ch2;
count++;
if(count==8)
{
tarr[8]='\0';
decrypt(tarr,keypc23); //tarr will contain the result
encrypt(tarr,keypc22);
decrypt(tarr,keypc21);
for(iter=0;iter<8;iter++)
{
ch=tarr[iter];
fputc(ch,fap);
}
count=0;
}
}
}
while(!feof(fc));
fclose(fap);
fclose(fc);
/* ft is for temporary file to remove the padded characters */
ft=fopen("temp.dat","wb");
if(ft==NULL)
{
printf("\n cannot open file temp.dat ");
getch();
exit(0);
}
fap=fopen(dest,"rb");
if(fap==NULL)
{
printf("\n cannot open file %s ",dest);
getch();
exit(0);
}
fseek(fap,-1,2); /*go 1 byte before the end to get how many characters has been padded*/
ch=fgetc(fap);
tempchar=ch-48; // covert to integer
fseek(fap,-tempchar,2);
pos=ftell(fap); // to get the position where the file actually ends

```

```

/* extract the actual file using a temporary file */
rewind(fap); // go to the beginning of the file
while(!feof(fap))
{
ch=fgetc(fap);
fputc(ch,ft);
if(pos==ftell(fap))
break;
}
fclose(fap);
fclose(ft);
remove(dest);
rename("temp.dat",dest);
second=time(NULL);
printf("\n Decryption Time =%f seconds ",difftime(second,first));
getch();
}

void getkey(unsigned char key1[],unsigned char key2[],unsigned char key3[])
// generates 3 keys from key file
{
FILE *fp;
unsigned char ch;
char keyfile[40];
int i;
strcpy(key1,"");
strcpy(key2,"");
strcpy(key3,"");
printf("\n Enter the Key file (it should contain 192 characters) : ");
scanf("%s",keyfile);
fp=fopen(keyfile,"rb");
if(fp==NULL)
{
printf("\n Error opening file\n");
getch();
exit(1);
}
i=1;
while(!feof(fp) && i<=8)
{
ch=fgetc(fp);
binkey(key1,ch);
i++;
}
while(i<=8)
{
binkey(key1,' ');
i++;
}
i=1;

```

```

while(!feof(fp) && i<=8)
{
ch=fgetc(fp);
binkey(key2,ch);
i++;
}
while(i<=8)
{
binkey(key2,' ');
i++;
}
i=1;
while(!feof(fp) && i<=8)
{
ch=fgetc(fp);
binkey(key3,ch);
i++;
}
while(i<=8)
{
binkey(key3,' ');
i++;
}
fclose(fp);
}

```

```

void binkey(unsigned char key[],int dec)
{
unsigned char tempbin[9],bin[9],tbin[2];
int j,k,num;
strcpy(bin,"");
for(j=0;j<8;j++)
tempbin[j]=0;
num=0;
num=dec;
j=7;
do
{
tempbin[j]=num%2;
num/=2;
j--;
}
while(num!=0);
for(k=0;k<8;k++)
{
itoa(tempbin[k],tbin,10); //converting integer to character
if(k==0)
strcpy(bin,tbin);
else
strcat(bin,tbin); }
}

```

```

bin[8]='\0';
if(strlen(key)==0)
strcpy(key,bin);
else
strcat(key,bin);
}

void encrypt(unsigned char *tarr,unsigned char keypc2[16][49])
{
unsigned char pbits[65]="0",tempbits[65];
unsigned char leftbits[33],rightbits[33],templeft[49],tempright[49]="0";
int i;
int ip[8][8]={ 58,50,42,34,26,18,10,02,
               60,52,44,36,28,20,12,04,
               62,54,46,38,30,22,14,06,
               64,56,48,40,32,24,16,08,
               57,49,41,33,25,17,09,01,
               59,51,43,35,27,19,11,03,
               61,53,45,37,29,21,13,05,
               63,55,47,39,31,23,15,07           };
int ipinv[8][8]={ 40,08,48,16,56,24,64,32,
                 39,07,47,15,55,23,63,31,
                 38,06,46,14,54,22,62,30,
                 37,05,45,13,53,21,61,29,
                 36,04,44,12,52,20,60,28,
                 35,03,43,11,51,19,59,27,
                 34,02,42,10,50,18,58,26,
                 33,01,41,09,49,17,57,25           };
convtbit(tarr,pbits); // converts string tarr to ascii code in pbits
permutation(pbits,ip); /* performing initial permutation*/
for(i=0;i<32;i++)
{
leftbits[i]=pbits[i];
rightbits[i]=pbits[32+i];
}
leftbits[32]='\0';
rightbits[32]='\0';
/* Starting rounds */
for(i=0;i<16;i++)
{
strcpy(tempright,"");
funF(rightbits,keypc2[i],tempright); /*implement function F with rightbits, result is in tempright*/
/*for next iteration left and right bits are */
strcpy(templeft,leftbits);
strcpy(leftbits,rightbits); //L(i)=R(i-1)
getXOR(templeft,tempright,32); //result in tempright
strcpy(rightbits,tempright); //R(i)=L(i-1) XOR F
}
/* Performing 32-bit swapping */
strcpy(tempbits,rightbits);

```

```

strcat(tempbits,leftbits);
/* Perform inverse initial permutation */
permutation(tempbits,ipinv);
convchar(tempbits,tarr);
}

void decrypt(unsigned char *tarr,unsigned char keypc2[16][49])
{
unsigned char cbits[65],tempbits[65],leftbits[33],rightbits[33],templeft[49],tempright[49];
int i;
int ip[8][8]={ 58,50,42,34,26,18,10,02,
                60,52,44,36,28,20,12,04,
                62,54,46,38,30,22,14,06,
                64,56,48,40,32,24,16,08,
                57,49,41,33,25,17,09,01,
                59,51,43,35,27,19,11,03,
                61,53,45,37,29,21,13,05,
                63,55,47,39,31,23,15,07        };
int ipinv[8][8]={ 40,08,48,16,56,24,64,32,
                 39,07,47,15,55,23,63,31,
                 38,06,46,14,54,22,62,30,
                 37,05,45,13,53,21,61,29,
                 36,04,44,12,52,20,60,28,
                 35,03,43,11,51,19,59,27,
                 34,02,42,10,50,18,58,26,
                 33,01,41,09,49,17,57,25        };

/* decrypting the cyphertext */
convtbit(tarr,cbits);
permutation(cbits,ip); /* Performing initial permutation */
for(i=0;i<32;i++)
{
leftbits[i]=cbits[i];
rightbits[i]=cbits[32+i];
}
leftbits[32]='\0';
rightbits[32]='\0';
/* Starting rounds */
for(i=15;i>=0;i--)
{
funF(rightbits,keypc2[i],tempright); //result is in tempright
/*for next iteration left and right bits are */
strcpy(templeft,leftbits);
strcpy(leftbits,rightbits); //L(i)=R(i-1)
getXOR(templeft,tempright,32); //result in tempright
strcpy(rightbits,tempright); //R(i)=L(i-1) XOR F
}
/* Performing 32-bit swapping */
strcpy(tempbits,rightbits);
strcat(tempbits,leftbits);
permutation(tempbits,ipinv); /* Perform inverse initial permutation */

```

```
convchar(tempbits,tarr);
}
```

```
void convtbit(unsigned char tarr[],unsigned char pbits[])
{
int tempbin[9],i,j,k,num;
unsigned char bin[9],tbin[2];
strcpy(pbits,"");
for(i=0;i<8;i++)
{
strcpy(bin,"");
for(j=0;j<8;j++)
tempbin[j]=0;
num=0;
num=tarr[i];
j=7;
do
{
tempbin[j]=num%2;
num/=2;
j--;
} while(num!=0);
for(k=0;k<8;k++)
{
itoa(tempbin[k],tbin,10); //converting integer to character
if(k==0)
strcpy(bin,tbin);
else
strcat(bin,tbin);
}
bin[8]='\0';
if(i==0)
strcpy(pbits,bin);
else
strcat(pbits,bin);
}
}
```

```
void convtopc1(unsigned char key[65],int pc1[8][7],unsigned char ckey[],unsigned char dkey[])
/* convert 64 bit key to 56 bit key by permuted choice-1 */
{
int i,j,k,l=0;
k=0;
for(i=0;i<8;i++)
{
for(j=0;j<7;j++)
{
if(i<4)
{
ckey[k]=key[(pc1[i][j] - 1)];
```

```

k++;
}
else
{
dkey[l]=key[(pc1[l][j]-1)];
l++;
}
}
}
}
ckey[28]='\0';
dkey[28]='\0';
}

```

```

void shiftbits(unsigned char tkeys[],int n)
{
int i,j;
unsigned char temp;
for(i=1;i<=n;i++)
{
temp=tkeys[0];
for(j=1;j<28;j++)
{
tkeys[j-1]=tkeys[j];
}
tkeys[27]=temp;
}
}

```

```

void getpc2(unsigned char keypc1[],int pc2[6][8],unsigned char keypc2[])
{
int i,j,k;
k=0;
for(i=0;i<6;i++)
{
for(j=0;j<8;j++)
{
keypc2[k]=keypc1[(pc2[i][j]-1)];
k++;
}
}
keypc2[48]='\0';
}

```

```

void getexpbits(unsigned char rightbits[],int exp[8][6],unsigned char tempright[])
{
int i,j,k;
strcpy(tempright,"");
k=0;

```

```

for(i=0;i<8;i++)
{
for(j=0;j<6;j++)
{
tempright[k]=rightbits[(exp[i][j]-1)];
k++;
}
}
tempright[48]='\0';
}

```

```

void getXOR(unsigned char keypc2[],unsigned char tempright[],int n)
{
int bit1,bit2,xor,i;
unsigned char ch1[2],ch2[2],temp[49];
for(i=0;i<n;i++)
{
ch1[0]=keypc2[i];
ch2[0]=tempright[i];
ch1[1]='\0';
ch2[1]='\0';
bit1=atoi(ch1);
bit2=atoi(ch2);
xor=bit1+bit2;
if(xor==2)
temp[i]='0';
else if(xor==0)
temp[i]='0';
else
temp[i]='1';
}
temp[n]='\0';
strcpy(tempright,temp);
}

```

```

int dec(long bin)
{
int i=0,q,dcml=0;
do
{
q=bin%10;
dcml+=q*(int)pow(2,i);
bin/=10;
i++;
}while(bin!=0);
return dcml;
}

```

```

void bin(int dec,unsigned char binstr[])
{
static int tempbin[4];
unsigned char tbin[2]="0";
int i,j,k,flag=0,num;
strcpy(binstr,"");
num=dec;
for(i=0;i<4;i++)
tempbin[i]=0;
j=3;
do
{
tempbin[j]=num%2;
num/=2;
j--;
} while(num!=0);
for(k=0;k<4;k++)
{
itoa(tempbin[k],tbin,10); //converting integer to string(character)
if(flag==0) //for first bit
{
strcpy(binstr,tbin);
flag=1;
}
else
strcat(binstr,tbin);
}
}

void getsubkey(unsigned char xorbites[],unsigned char subskey[])
{
int row,col,temp;
unsigned char tempstr1[3],tempstr2[5],tempright[49];
int sbox1[4][16]={
14,04,13,01,02,15,11,08,03,10,06,12,05,09,00,07,
00,15,07,04,14,02,13,01,10,06,12,11,09,05,03,08,
04,01,14,08,13,06,02,11,15,12,09,07,03,10,05,00,
15,12,08,02,04,09,01,07,05,11,03,14,10,00,06,13
};
int sbox2[4][16]={
15,01,08,14,06,11,03,04,09,07,02,13,12,00,05,10,
03,13,04,07,15,02,08,14,12,00,01,10,06,09,11,05,
00,14,07,11,10,04,13,01,05,08,12,06,09,03,02,15,
13,08,10,01,03,15,04,02,11,06,07,12,00,05,14,09
};
int sbox3[4][16]={
10,00,09,14,06,03,15,05,01,13,12,07,11,04,02,08,
13,07,00,09,03,04,06,10,02,08,05,14,12,11,15,01,
13,06,04,09,08,15,03,00,11,01,02,12,05,10,14,07,
01,10,13,00,06,09,08,07,04,15,14,03,11,05,02,12
};
int sbox4[4][16]={
07,13,14,03,00,06,09,10,01,02,08,05,11,12,04,15,
13,08,11,05,06,15,00,03,04,07,02,12,01,10,14,09,
10,06,09,00,12,11,07,13,15,01,03,14,05,02,08,04,
03,15,00,06,10,01,13,08,09,04,05,11,12,07,02,14
};
}

```

```

int sbox5[4][16]={    02,12,04,01,07,10,11,06,08,05,03,15,13,00,14,09,
                    14,11,02,12,04,07,13,01,05,00,15,10,03,09,08,06,
                    04,02,01,11,10,13,07,08,15,09,12,05,06,03,00,14,
                    11,08,12,07,01,14,02,13,06,15,00,09,10,04,05,03    };
int sbox6[4][16]={    12,01,10,15,09,02,06,08,00,13,03,04,14,07,05,11,
                    10,15,04,02,07,12,09,05,06,01,13,14,00,11,03,08,
                    09,14,15,05,02,08,12,03,07,00,04,10,01,13,11,06,
                    04,03,02,12,09,05,15,10,11,14,01,07,06,00,08,13    };
int sbox7[4][16]={    04,11,02,14,15,00,08,13,03,12,09,07,05,10,06,01,
                    13,00,11,07,04,09,01,10,14,03,05,12,02,15,08,06,
                    01,04,11,13,12,03,07,14,10,15,06,08,00,05,09,02,
                    06,11,13,08,01,04,10,07,09,05,00,15,14,02,03,12    };
int sbox8[4][16]={    13,02,08,04,06,15,11,01,10,09,03,14,05,00,12,07,
                    01,15,13,08,10,03,07,04,12,05,06,11,00,14,09,02,
                    07,11,04,01,09,12,14,02,00,06,10,13,15,03,05,08,
                    02,00,14,07,04,10,08,13,15,12,09,00,03,05,06,11    };

strcpy(temptright,xorbits);
/* use s-box1 */
tempstr1[0]=temptright[0];
tempstr1[1]=temptright[5];
tempstr1[2]='\0';
tempstr2[0]=temptright[1];
tempstr2[1]=temptright[2];
tempstr2[2]=temptright[3];
tempstr2[3]=temptright[4];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox1[row][col];
bin(temp,tempstr2); //tempstr2 will hold binary value
strcpy(subskey,tempstr2);
/* use s-box2 */
tempstr1[0]=temptright[6];
tempstr1[1]=temptright[11];
tempstr1[2]='\0';
tempstr2[0]=temptright[7];
tempstr2[1]=temptright[8];
tempstr2[2]=temptright[9];
tempstr2[3]=temptright[10];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox2[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);
/* use s-box3 */
tempstr1[0]=temptright[12];
tempstr1[1]=temptright[17];
tempstr1[2]='\0';

```

```

tempstr2[0]=tempright[13];
tempstr2[1]=tempright[14];
tempstr2[2]=tempright[15];
tempstr2[3]=tempright[16];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox3[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);
/* use s-box4 */
tempstr1[0]=tempright[18];
tempstr1[1]=tempright[23];
tempstr1[2]='\0';
tempstr2[0]=tempright[19];
tempstr2[1]=tempright[20];
tempstr2[2]=tempright[21];
tempstr2[3]=tempright[22];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox4[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);
/* use s-box5 */
tempstr1[0]=tempright[24];
tempstr1[1]=tempright[29];
tempstr1[2]='\0';
tempstr2[0]=tempright[25];
tempstr2[1]=tempright[26];
tempstr2[2]=tempright[27];
tempstr2[3]=tempright[28];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox5[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);
/* use s-box6 */
tempstr1[0]=tempright[30];
tempstr1[1]=tempright[35];
tempstr1[2]='\0';
tempstr2[0]=tempright[31];
tempstr2[1]=tempright[32];
tempstr2[2]=tempright[33];
tempstr2[3]=tempright[34];
tempstr2[4]='\0';

```

```

row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox6[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);
/* use s-box7 */
tempstr1[0]=tempright[36];
tempstr1[1]=tempright[41];
tempstr1[2]='\0';
tempstr2[0]=tempright[37];
tempstr2[1]=tempright[38];
tempstr2[2]=tempright[39];
tempstr2[3]=tempright[40];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox7[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);
/* use s-box8 */
tempstr1[0]=tempright[42];
tempstr1[1]=tempright[47];
tempstr1[2]='\0';
tempstr2[0]=tempright[43];
tempstr2[1]=tempright[44];
tempstr2[2]=tempright[45];
tempstr2[3]=tempright[46];
tempstr2[4]='\0';
row=dec(atoi(tempstr1));
col=dec(atoi(tempstr2));
temp=sbox8[row][col];
strcpy(tempstr2,"");
bin(temp,tempstr2); //tempstr2 will hold binary value
strcat(subskey,tempstr2);
}

```

```

void getPerm(unsigned char subskey[],int permF[4][8],unsigned char tempright[])
{
int i,j,k=0;
strcpy(tempright,"");
for(i=0;i<4;i++)
{
for(j=0;j<8;j++)
{
tempright[k]=subskey[(permF[i][j]-1)];
k++;
}
}
}

```

```
tempright[32]='\0';
}
```

```
void permutation(unsigned char tbits[],int permbits[8][8])
{
unsigned char tempbits[65];
int i,j,k=0;
for(i=0;i<8;i++)
{
for(j=0;j<8;j++)
{
tempbits[k]=tbits[(permbits[i][j]-1)];
k++;
}
}
tempbits[64]='\0';
strcpy(tbits,tempbits);
}
```

```
void convchar(unsigned char tempbits[],unsigned char *tarr)
{
int j,k=0,num;
unsigned char temp[9],ch;
while(k<8)
{
for(j=0;j<8;j++)
{
temp[j]=tempbits[8*k+j];
}
temp[8]='\0'; //holds 8 bit for a character
num=dec(atol(temp)); //convert binary to decimal value
ch=num;
tarr[k]=ch;
k++;
}
tarr[8]='\0';
}
```

```
void funF(unsigned char rightbits[33],unsigned char keypc2[49],unsigned char tempright[])
//1st parameter holds result
{
unsigned char subskey[33];
int exp[8][6]={
32,01,02,03,04,05, // E table
04,05,06,07,08,09,
08,09,10,11,12,13,
12,13,14,15,16,17,
16,17,18,19,20,21,
20,21,22,23,24,25,
24,25,26,27,28,29,
28,29,30,31,32,01
};
```

```
int permF[4][8]={    16,07,20,21,29,12,28,17, //P table
                    01,15,23,26,05,18,31,10,
                    02,08,24,14,32,27,03,09,
                    19,13,30,06,22,11,04,25    };
```

```
strcpy(tempright,"");
```

```
/* expand right 32 bits to make 48 bits */
getexbits(rightbits,exp,tempright);
//result stored in tempright[]
```

```
/* XOR begins here */
getXOR(keypc2,tempright,48);
//result is in tempright[]
```

```
/* substitution using S-boxes */
getsubskey(tempright,subskey);
// result is in subskey 32 bit
```

```
/* permutation of the substituted key */
strcpy(tempright,"0");
getPerm(subskey,permF,tempright);
//result will be in tempright (32 bit)
}
```

19. Source code for computing character frequencies

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>

int main(void)
{
    int i,count,l,c;
    char srcfile[30];
    char destfile[30];
    long int temp1[256];
    long int temp2[256];
    FILE *fenc;
    FILE *fp;
    printf("\n Enter the file name : ");
    gets(srcfile);
    printf("\n Enter the file name for storing frequencies : ");
    gets(destfile);
    fenc = fopen(srcfile,"rb");
    fp = fopen(destfile,"w");
    if(fenc == NULL || fp == NULL)
    {
        printf("File Opening Error! \n");
        getch();
        exit(1);
    }
    else
    {
        for(i=0;i<256;i++)
        {
            temp1[i] = 0;
            temp2[i] = 0;
        }
        while((c = fgetc(fenc))!=EOF)
        {
            temp1[c]++;
        }
        fclose(fenc);
        for(l=0;l<256;l++)
        {
            count=temp1[l];
            fprintf(fp,"%d\t%d\n",l,count);
        }
        fclose(fp);
    }
    return 0;
}
```

20. Source code for computing the χ^2 value between files

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>

void main()
{
FILE *fp,*fc;
char fsrc[40],fdest[40];
static unsigned long int count[256],calc_tab[3][257],expect[3][256];
unsigned long int charint,tot=0,totsrc=0,totdest=0,i,df,flag,merge,j,k;
unsigned char ch;
long double chisq=0.0,denom;
for(i=0;i<256;i++)
count[i]=0;
printf("\n Enter the source file name : ");
gets(fsrc);
fp=fopen(fsrc,"rb");
if(fp==NULL)
{
printf("\n Cannot open file ");
exit(0);
}
ch=fgetc(fp);
while(!feof(fp))
{
charint=ch; // convert to integer
count[charint]=count[charint] + 1;
tot++;
totsrc++;
ch=fgetc(fp); // for next iteration
}
fclose(fp);
for(i=0;i<256;i++)
{
calc_tab[0][i]=count[i];
}
calc_tab[0][256]=totsrc;
for(i=0;i<256;i++)
count[i]=0;
printf("\n Enter the Encrypted file name : ");
gets(fdest);
fc=fopen(fdest,"rb");
if(fc==NULL)
{
printf("\n Cannot open file ");
exit(0);
}
}

```

```

ch=fgetc(fc);
while(!feof(fc))
{
if((totsrc==totdest) // to handle Triple DES algorithm where padding increases enc filesize
break;
charint=ch; // convert to integer
count[charint]=count[charint]+1;
tot++;
totdest++;
ch=fgetc(fc); // for next iteration
}
fclose(fc);
for(i=0;i<256;i++)
{
calc_tab[1][i]=count[i];
}
calc_tab[1][256]=totdest;
for(i=0;i<257;i++)
{
calc_tab[2][i]=calc_tab[0][i]+calc_tab[1][i];
}
for(i=0;i<256;i++)
{
expect[0][i]=(calc_tab[0][256]*calc_tab[2][i])/tot;
expect[1][i]=(calc_tab[1][256]*calc_tab[2][i])/tot;
}
do{
flag=1;
for(i=0;i<256;i++)
{
if(calc_tab[2][i]==0) // if there are no characters in that column
continue;
if((expect[0][i]<10 && expect[0][i]>=0) || (expect[1][i]<10 && expect[1][i]>=0))
{
flag=0;
merge=0;
for(j=i-1;j>=0;j--)
{
if(calc_tab[2][j]!=0)
{
calc_tab[0][j]=calc_tab[0][j]+calc_tab[0][i];
calc_tab[1][j]=calc_tab[1][j]+calc_tab[1][i];
calc_tab[2][j]=calc_tab[0][j]+calc_tab[1][j];
calc_tab[0][i]=0;
calc_tab[1][i]=0;
calc_tab[2][i]=0;
merge=1;
break;
}
}
}
}
}

```

```

if(merge==0) // if merging has not been done then search from last to first
{
for(j=i+1;j<256;j++)
{
if(calc_tab[2][j]!=0)
{
calc_tab[0][i]=calc_tab[0][i]+calc_tab[0][j];
calc_tab[1][i]=calc_tab[1][i]+calc_tab[1][j];
calc_tab[2][i]=calc_tab[0][i]+calc_tab[1][i];
calc_tab[0][j]=0;
calc_tab[1][j]=0;
calc_tab[2][j]=0;
merge=1;
break;
}
}
}
if(merge==0)
{
if(calc_tab[2][i]<10 || calc_tab[1][i]<10)
{
printf("\n Too few characters in the files\n");
getch();
exit(0);
}
}
for(k=0;k<256;k++)
{
expect[0][k]=(calc_tab[0][256]*calc_tab[2][k])/tot;
expect[1][k]=(calc_tab[1][256]*calc_tab[2][k])/tot;
}
}
}
while(flag==0);
df=0;
for(i=0;i<256;i++)
{
if(calc_tab[2][i]!=0)
{
df++;
denom=(calc_tab[0][256]*calc_tab[2][i])/(float)tot;
if(denom != 0.0)
chisq+=(float)(pow(calc_tab[0][i],2)+pow(calc_tab[1][i],2))/denom;
}
}
chisq=tot;
printf("\n\n Calculated Chi Square value %.4lf ",chisq);
printf("\n Degree Of Freedom=%lu\n",df);
}

```

8085 Assembly Language Programs

1. 8-bit RSPB encryption/decryption

```

                                ORG 0F800H
                                LXI H,0F900H
                                LXI D,0FA00H
                                MVI A,40H

LOOP:                          STA 202H
                                MOV A,M
                                PUSH H
                                PUSH D
                                CALL MAP
                                POP D
                                POP H
                                MOV A,M
                                ANI 80H
                                ADD B
                                STAX D
                                INX H
                                INX D
                                LDA 202H
                                DCR A
                                JNZ LOOP
                                HLT
                                END

MAP:                            LXI H,0FB00H
                                LXI D,0FD00H
                                MVI B,00H
                                STA 200H
                                MVI A,07H

LOOP1:                          STA 201H
                                MOV C,M
                                INR H
                                MOV A,M
                                RRC
                                JNC LP
                                LDA 200H
                                XCHG
                                ANA M
                                XCHG
```

LP11: RRC
 DCR C
 JNZ LP11
 JMP LPOO

LP: LDA 200H
 XCHG
 ANA M
 XCHG

LP22: RLC
 DCR C
 JNZ LP22

LPOO: ADD B
 MOV B,A
 DCR H
 INX H
 INX D
 LDA 201H
 DCR A
 JNZ LOOP1
 RET

ORG 0FB00H
DB 04H, 01H, 01H, 02H, 02H, 01H, 03H

ORG 0FC00H
DB 00H, 01H, 01H, 00H, 01H, 00H, 01H

ORG 0FD00H
DB 01H, 02H, 04H, 08H, 10H, 20H, 40H

END

2. 16-bit (and higher) RSPB encryption/decryption

```

                ORG 0F800H
                MVI D,02H
                LXI B,0F900H

MN:            LXI SP,0FFA0H
                PUSH B
                PUSH D
                PUSH H
                PUSH PSW
                CALL OPPR
                LXI H,0FC00H
                MOV A,M
                INR A
                INR A
                MOV M,A
                POP PSW
                POP H
                POP D
                POP B
                INX B
                INX B
                DCR D
                JNZ MN
                HLT
                END

OPPR:         MVI L,00H
                MVI D,02H

L2:           MVI E,08H
                MVI H,01H

L3:           LDAX B
                ANA H
                CNZ BTST
                INR L
                MOV A,H
                RLC
                MOV H,A
                DCR E
                JNZ L3
                INX B
                DCR D
                JNZ L2
                RET

```

BTST: PUSH PSW
 PUSH B
 PUSH H
 LDA 0FC00H
 MOV B,A
 MVI H,2BH
 MOV A,M

BB: SUI 08H
 JC AA
 INR B
 JMP BB

AA: ADI 08H
 MOV C,A
 MVI H,0FAH
 MOV L,B
 MVI A,01H
 MOV A,C
 CPI 00H
 JZ YY
 MVI A,01H

XX: RLC
 DCR C
 JNZ XX

3. 8-bit LSPB encryption/decryption

```
                ORG 0F800H
                LXI H,0F900H
                LXI D,0FA00H
                MVI A,40H

LOOP:           STA 202H
                MOV A,M
                PUSH H
                PUSH D
                CALL MAP
                POP D
                POP H
                MOV A,M
                ANI 80H
                ADD B
                RRC
                RRC
                RRC
                RRC
                STAX D
                INX H
                INX D
                LDA 202H
                DCR A
                JNZ LOOP
                HLT
                END

MAP:           LXI H,0FB00H
                LXI D,0FD00H
                MVI B,00H
                STA 200H
                MVI A,07H

LOOP1:         STA 201H
                MOV C,M
                INR H
                MOV A,M
                RRC
                JNC LP
                LDA 200H
                XCHG
                ANA M
                XCHG
```

LP11: RRC
 DCR C
 JNZ LP11
 JMP LPOO

LP: LDA 200H
 XCHG
 ANA M
 XCHG

LP22: RLC
 DCR C
 JNZ LP22

LPOO: ADD B
 MOV B,A
 DCR H
 INX H
 INX D
 LDA 201H
 DCR A
 JNZ LOOP1
 RET

ORG 0FB00H
DB 04H, 01H, 01H, 02H, 02H, 01H, 03H

ORG 0FC00H
DB 00H, 01H, 01H, 00H, 01H, 00H, 01H

ORG 0FD00H
DB 01H, 02H, 04H, 08H, 10H, 20H, 40H

END

4. 16-bit (and higher) LSPB encryption/decryption

```

                ORG 0F800H
                MVI D,02H
                LXI B,0F900H

MN:            LXI SP,0FFA0H
                PUSH B
                PUSH D
                PUSH H
                PUSH PSW
                CALL OPPR
                LXI H,0FC00H
                MOV A,M
                INR A
                INR A
                MOV M,A
                POP PSW
                POP H
                POP D
                POP B
                INX B
                INX B
                DCR D
                JNZ MN
                HLT
                END

OPPR:         MVI L,00H
                MVI D,02H

L2:           MVI E,08H
                MVI H,01H

L3:           LDAX B
                ANA H
                CNZ BTST
                INR L
                MOV A,H
                RLC
                MOV H,A
                DCR E
                JNZ L3
                INX B
                DCR D
                JNZ L2
                RET

```

BTST: PUSH PSW
 PUSH B
 PUSH H
 LDA 0FC00H
 MOV B,A
 MVI H,2BH
 MOV A,M

BB: SUI 08H
 JC AA
 INR B
 JMP BB

AA: ADI 08H
 MOV C,A
 MVI H,0FAH
 MOV L,B
 MVI A,01H
 MOV A,C
 CPI 00H
 JZ YY
 MVI A,01H

XX: RLC
 DCR C
 JNZ XX
 JMP PP

YY: MVI A,01H

PP: ORA M
 MOV M,A
 POP H
 POP B
 POP PSW
 RET

ORG 0FB00H
DB 00H, 0AH, 0BH, 01H, 0CH, 02H, 0DH, 03H,
04H, 05H, 0EH, 06H, 0FH, 07H, 08H, 09H,

ORG 0F900H
DB 5AH, 9CH, 23H, 65H

ORG 2500H
DB 00H

END

5. 8-bit DSPB encryption/decryption

```
ORG 0F800H
LXI H,0F900H
LXI D,0FA00H
MVI A,40H
```

```
LOOP:  STA 202H
        MOV A,M
        PUSH H
        PUSH D
        CALL MAP
        POP D
        POP H
        MOV A,M
        ANI 80H
        ADD B
        ANI 0FH
        RRC
        RRC
        MOV C,A
        MOV A,B
        ANI 03H
        ADD C
        MOV C,A
        MOV A,B
        ANI 0CH
        RLC
        RLC
        RLC
        RLC
        ADC C
        STAX D
        INX H
        INX D
        LDA 202H
        DCR A
        JNZ LOOP
        HLT
        END
```

```
MAP:   LXI H,0FB00H
        LXI D,0FD00H
        MVI B,00H
        STA 200H
        MVI A,07H
```

LOOP1: STA 201H
MOV C,M
INR H
MOV A,M
RRC
JNC LP
LDA 200H
XCHG
ANA M
XCHG

LP11: RRC
DCR C
JNZ LP11
JMP LPOO

LP: LDA 200H
XCHG
ANA M
XCHG

LP22: RLC
DCR C
JNZ LP22

LPOO: ADD B
MOV B,A
DCR H
INX H
INX D
LDA 201H
DCR A
JNZ LOOP1
RET

ORG FB00H
DB 04H, 01H, 01H, 02H, 02H, 01H, 03H

ORG FC00H
DB 00H, 01H, 01H, 00H, 01H, 00H, 01H

ORG FD00H
DB 01H, 02H, 04H, 08H, 10H, 20H, 40H

END

6. 16-bit (and higher) DSPB encryption/decryption

```

                ORG 0F800H
                MVI B,10H
                MVI D,02H
                LXI B,0F900H

MN:            LXI SP,0FFA0H
                PUSH B
                PUSH D
                PUSH H
                PUSH PSW
                CALL OPPR
                LXI H,0FC00H
                MOV A,M
                INR A
                INR A
                MOV M,A
                POP PSW
                POP H
                POP D
                POP B
                INX B
                INX B
                DCR D
                JNZ MN
                HLT
                END

OPPR:         MVI L,00H
                MVI D,02H

L2:           MVI E,08H
                MVI H,01H

L3:           LDAX B
                ANA H
                CNZ BTST
                INR L
                MOV A,H
                RLC
                MOV H,A
                DCR E
                JNZ L3
                INX B
                DCR D
                JNZ L2
                RET

```

BTST: PUSH PSW
 PUSH B
 PUSH H
 LDA 0FC00H
 MOV B,A
 MVI H,2BH
 MOV A,M

BB: SUI 08H
 JC AA
 INR B
 JMP BB

AA: ADI 08H
 MOV C,A
 MVI H,0FAH
 MOV L,B
 MVI A,01H
 MOV A,C
 CPI 00H
 JZ YY
 MVI A,01H

XX: RLC
 DCR C
 JNZ XX
 JMP PP

YY: MVI A,01H

PP: ORA M
 MOV M,A
 POP H
 POP B
 POP PSW
 RET

ORG FB00H
DB 03H, 00H, 01H, 04H, 02H, 05H, 0DH, 06H,
07H, 08H, 0EH, 09H, 0FH, 0AH, 0BH, 0CH,

ORG F900H
DB 5AH, 9CH, 23H, 65H

ORG 2500H
DB 00H

END

7. 8-bit BET encryption/decryption

```
                ORG 0F800H
                LXI H,0F900H
                LXI D,0FA00H
                MVI A,40H

LOOP:           STA 202H
                MOV A,M
                PUSH H
                PUSH D
                CALL BET8
                POP D
                POP H
                MOV A,M
                ANI 10H
                ADD B
                ANI 0F0H
                RRC
                RRC
                MOV C,A
                MOV A,B
                ANI 01H
                ADD C
                MOV C,A
                MOV A,B
                ANI 0CH
                RLC
                RLC
                RLC
                RLC
                ADC C
                STAX D
                INX H
                INX D
                LDA 202H
                DCR A
                JNZ LOOP
                HLT
                END

BET8:          LXI H,0FB00H
                LXI D,0FD00H
                MVI B,00H
                STA 200H
                MVI A,07H
```

```
LOOP1:   STA 201H
         MOV C,M
         INR H
         MOV A,M
         RRC
         JNC LP
         LDA 200H
         XCHG
         ANA M
         XCHG

LP11:    RRC
         DCR C
         JNZ LP11
         JMP LPOO

LP:      LDA 200H
         XCHG
         ANA M
         XCHG

LP22:    RLC
         DCR C
         JNZ LP22

LPOO:    ADD B
         MOV B,A
         DCR H
         INX H
         INX D
         LDA 201H
         DCR A
         JNZ LOOP1
         RET
```

```
ORG 0FB00H
DB 03H, 01H, 02H, 02H, 01H, 03H
```

```
ORG 0FC00H
DB 00H, 01H, 00H, 01H, 00H, 01H
```

```
ORG 0FD00H
DB 02H, 04H, 08H, 10H, 20H, 40H
```

```
END
```

8. 16-bit (and higher) BET encryption/decryption

```

                ORG 0F800H
                MVI B,40H
                LXI H,0FA00H
                MVI A,00H

LPLP:          MOV M,A
                INX H
                DCR B
                JNZ LPLP
                MVI D,02H
                LXI B,0F900H

MN:           LXI SP,0FFA0H
                PUSH B
                PUSH D
                PUSH H
                PUSH PSW
                CALL OPPR
                LXI H,0FC00H
                MOV A,M
                INR A
                INR A
                MOV M,A
                POP PSW
                POP H
                POP D
                POP B
                INX B
                INX B
                DCR D
                JNZ MN
                HLT
                END

OPPR:         MVI L,00H
                MVI D,02H

L2:           MVI E,08H
                MVI H,01H

L3:           LDAX B
                ANA H
                CNZ BSTR
                INR L
                MOV A,H
                RLC
                MOV H,A
                DCR E
                JNZ L3

```

```

        INX B
        DCR D
        JNZ L2
        RET

BSTR:   PUSH PSW
        PUSH B
        PUSH H
        LDA 0FC00H
        MOV B,A
        MVI H,2BH
        MOV A,M

BB:     SUI 08H
        JC AA
        INR B
        JMP BB

AA:     ADI 08H
        MOV C,A
        MVI H,0FAH
        MOV L,B
        MOV A,M
        CPI 00H
        JZ YY
        MVI A,01H

XX:     RLC
        DCR C
        JNZ XX
        JMP PP

YY:     MVI A,01H

PP:     ORA M
        MOV M,A
        POP H
        POP B
        POP PSW
        RET

        ORG 0FB00H
        DB 00H, 08H, 01H, 09H, 02H, 0AH, 03H, 0BH,
        04H, 0CH, 05H, 0DH, 06H, 0EH, 07H, 0FH,

        ORG 0F900H
        DB 5AH, 9CH, 23H, 65H

        ORG 2500H
        DB 00H

        END

```

9. 8-bit SPOB encryption/decryption

```
                ORG 0F800H
                LXI H,0F900H
                LXI D,0FA00H
                MVI A,40H

LOOP:           STA 202H
                MOV A,M
                PUSH H
                PUSH D
                CALL SPOB
                POP D
                POP H
                MOV A,M
                ANI 01H
                ADD B
                MOV B,A
                MOV A,M
                ANI 80H
                ADD B
                STAX D
                INX H
                INX D
                LDA 202H
                DCR A
                JNZ LOOP
                HLT
                END

SPOB:          LXI H,0FB00H
                LXI D,0FD00H
                MVI B,00H
                STA 200H
                MVI A,06H

LOOP1:         STA 201H
                MOV C,M
                INR H
                MOV A,M
                RRC
                JNC LP
                LDA 200H
                XCHG
                ANA M
                XCHG
```

LP11: RRC
 DCR C
 JNZ LP11
 JMP LPOO

LP: LDA 200H
 XCHG
 ANA M
 XCHG

LP22: RLC
 DCR C
 JNZ LP22

LPOO: ADD B
 MOV B,A
 DCR H
 INX H
 INX D
 LDA 201H
 DCR A
 JNZ LOOP1
 RET

ORG 0FB00H
DB 01H, 01H, 03H, 01H, 04H, 02H

ORG 0FC00H
DB 01H, 00H, 01H, 00H, 01H, 01H

ORG 0FD00H
DB 01H, 04H, 08H, 20H, 40H, 80H

END

10. 16-bit (and higher) SPOB encryption/decryption

```

                ORG 0F800H
                MVI B,40H
                LXI H,0FA00H
                MVI A,00H

LPLP:          MOV M,A
                INX H
                DCR B
                JNZ LPLP
                MVI D,02H
                LXI B,0F900H

MN:           LXI SP,0FFA0H
                PUSH B
                PUSH D
                PUSH H
                PUSH PSW
                CALL OPPR
                LXI H,0FC00H
                MOV A,M
                INR A
                INR A
                MOV M,A
                POP PSW
                POP H
                POP D
                POP B
                INX B
                INX B
                DCR D
                JNZ MN
                HLT
                END

OPPR:         MVI L,00H
                MVI D,02H

L2:           MVI E,08H
                MVI H,01H

L3:           LDAX B
                ANA H
                CNZ BSTR
                INR L
                MOV A,H
                RLC
                MOV H,A
                DCR E
                JNZ L3

```

```
INX B
DCR D
JNZ L2
RET

BSTR:  PUSH PSW
        PUSH B
        PUSH H
        LDA 0FC00H
        MOV B,A
        MVI H,2BH
        MOV A,M

BB:    SUI 08H
        JC AA
        INR B
        JMP BB

AA:    ADI 08H
        MOV C,A
        MVI H,0FAH
        MOV L,B
        MOV A,M
        CPI 00H
        JZ YY
        MVI A,01H

XX:    RLC
        DCR C
        JNZ XX
        JMP PP

YY:    MVI A,01H

PP:    ORA M
        MOV M,A
        POP H
        POP B
        POP PSW
        RET

ORG 0FB00H
DB 06H, 09H, 05H, 0EH, 00H, 07H, 0BH, 0DH,
0FH, 0CH, 01H, 03H, 08H, 0AH, 02H, 04H,

ORG 0F900H
DB 5AH, 9CH, 23H, 65H

ORG 2500H
DB 00H

END
```

11. 8-bit MAT encryption

```

                ORG 0F800H
                MVI C,20H
                LXI H,0FA00H
                LXI D,0F900H

LOOP:          MOV A,M
                STAX D
                INX H
                INX D
                ADD A,M
                STAX D
                INX H
                INX D
                DCR C
                JNZ LOOP
                HLT
                END

```

12. 8-bit MAT decryption

```

                ORG 0F800H
                MVI C,20H
                LXI H,0FA00H
                LXI D,0F900H

LOOP:          MOV A,M
                STAX D
                INX H
                INX D
                CMP M
                JNC LOOP1
                SUB A,M
                JMP LOOP2

LOOP1          MOV B,A
                MOV A,M
                SUB A

LOOP2          STAX D
                INX H
                INX D
                DCR C
                JNZ LOOP
                HLT
                END

```

13. 16-bit (and higher) MAT encryption

```
ORG 0F800H
MVI C,10H
LXI H,0F901H
LXI D,0F903H

LOOP:    XRA A
         MVI B,02H

LOOP1:   LDAX D
         ADC M
         STAX D
         DCX H
         DCX D
         DCR B
         JNZ LOOP1
         MVI B,06H

LOOP2:   INX H
         INX D
         DCR B
         JNZ LOOP2
         DCR C
         JNZ LOOP
         HLT
         END
```

For block-sizes higher than 16 bits, few modifications according to table 5.1 of chapter 5 are required to be made in this program.

14. 16-bit (and higher) MAT decryption

```
ORG 0F800H
MVI C,10H
LXI H,0F901H
LXI D,0F903H

LOOP:    XRA A
         MVI B,02H

LOOP1:   LDAX D
         SBB M
         STAX D
         DCX H
         DCX D
         DCR B
         JNZ LOOP1
         MVI B,06H

LOOP2:   INX H
         INX D
         DCR B
         JNZ LOOP2
         DCR C
         JNZ LOOP
         HLT
         END
```

For block-sizes higher than 16 bits, few modifications according to table 5.1 of chapter 5 are required to be made in this program.

15. 8-bit OMAT encryption

```
                ORG 0F800H
                MVI C,3EH
                LXI H,0F900H
                MOV A,M

LOOP:           INX H
                ADD M
                MOV M,A
                DCR C
                JNZ LOOP
                HLT
                END
```

16. 8-bit OMAT decryption

```
                ORG 0F800H
                MVI C,3EH
                LXI H,0F93FH
                MOV A,M

LOOP:           DCX H
                SUB M
                MOV M,A
                DCR C
                JNZ LOOP
                HLT
                END
```

17. 16-bit (and higher) OMAT encryption

```
                ORG 0F800H
                MVI C,20H
                LXI H,0F903H
                LXI D,0F901H

LOOP3:          XRA A
                MVI B,02H

LOOP1:          LDAX D
                ADC M
                MOV M,A
                DCX H
                DCX D
                DCR B
                JNZ LOOP1
                MVI B,02H

LOOP2:          INX H
                INX D
                DCR B
                JNZ LOOP2
                DCR C
                JNZ LOOP3
                HLT
                END
```

For block-sizes higher than 16 bits, few modifications according to table 6.1 of chapter 6 are required to be made in this program.

18. 16-bit (and higher) OMAT decryption

```
                ORG 0F800H
                MVI C,20H
                LXI H,0F903H
                LXI D,0F901H

LOOP3:          XRA A
                MVI B,02H

LOOP1:          LDAX D
                SBB M
                MOV M,A
                DCX H
                DCX D
                DCR B
                JNZ LOOP1
                MVI B,02H

LOOP2:          INX H
                INX D
                DCR B
                JNZ LOOP2
                DCR C
                JNZ LOOP3
                HLT
                END
```

For block-sizes higher than 16 bits, few modifications according to table 6.1 of chapter 6 are required to be made in this program.

19. 8-bit MMAT encryption

```

                ORG 0F800H
                MVI C,20H
                LXI H,0F900H
                LXI D,0F901H

LOOP2:         LDAX D
                ADD M
                MOV M,A
                LDAX D
                ADD M
                STAX D
                MVI B,02H

LOOP1:         INX H
                INX D
                DCR B
                JNZ LOOP1
                DCR C
                JNZ LOOP2
                HLT
                END

```

20. 8-bit MMAT decryption

```

                ORG 0F800H
                MVI C,20H
                LXI H,0F900H
                LXI D,0F901H

LOOP2:         LDAX D
                SUB M
                STAX D
                MOV B,A
                MOV A,M
                SUB B
                MOV M,A
                MVI B,02H

LOOP1:         INX H
                INX D
                DCR B
                JNZ LOOP1
                DCR C
                JNZ LOOP2
                HLT
                END

```

21. 16-bit (and higher) MMAT encryption

```

                ORG 0F800H
                MVI C,10H
                LXI H,0F901H
                LXI D,0F903H

LOOP5:         XRA A
                MVI B,02H

LOOP1:         LDAX D
                ADC M
                STAX D
                DCX H
                DCX D
                DCR B
                JNZ LOOP1
                MVI B,02H

LOOP2:         INX H
                INX D
                DCR B
                JNZ LOOP2
                XRA A
                MVI B,02H

LOOP3:         LDAX D
                ADC M
                MOV M,A
                DCX H
                DCX D
                DCR B
                JNZ LOOP3
                MVI B,06H

LOOP4:         INX H
                INX D
                DCR B
                JNZ LOOP4
                DCR C
                JNZ LOOP5
                HLT
                END

```

For block-sizes higher than 16 bits, few modifications according to table 7.1 of chapter 7 are required to be made in this program.

22. 16-bit (and higher) MMAT decryption

```

                ORG 0F800H
                MVI C,10H
                LXI H,0F901H
                LXI D,0F903H

LOOP5:         XRA A
                MVI B,02H

LOOP1:         LDAX D
                SBB M
                STAX D
                DCX H
                DCX D
                DCR B
                JNZ LOOP1
                MVI B,02H

LOOP2:         INX H
                INX D
                DCR B
                JNZ LOOP2
                XRA A
                MVI B,02H

LOOP3:         LDAX D
                SBB M
                MOV M,A
                DCX H
                DCX D
                DCR B
                JNZ LOOP3
                MVI B,06H

LOOP4:         INX H
                INX D
                DCR B
                JNZ LOOP4
                DCR C
                JNZ LOOP5
                HLT
                END

```

For block-sizes higher than 16 bits, few modifications according to table 7.1 of chapter 7 are required to be made in this program.

23. 8-bit BOS encryption/decryption

```

                ORG 0F800H
                LXI SP,0FB00H
                LXI H,0F900H
                PUSH H
                LXI H,0FA00H
                MVI A,04H
                STA 0FC00H

LOOP3:         MVI E,04H

LOOP2:         MVI B,00H
                MVI C,00H
                MOV A,M
                RLC
                MOV M,A
                MOV A,B
                RAL
                MOV B,A
                MOV A,M
                RLC
                MOV M,A
                MOV C,A
                XRA B
                RAL
                MVI D,04H

LOOP1:         RLC
                DCR D
                JNZ LOOP1
                ORA B
                MOV D,A
                XTHL
                MOV A,M
                RLC
                ORA D
                MOV M,A
                XTHL
                DCR E
                JNZ LOOP2
                INX H
                XTHL
                INXH
                XTHL
                LDA 0FC00H
                DCR A
                STA 0FC00H
                JNZ LOOP3
                HLT
                END

```

24. 16-bit (and higher) BOS encryption/decryption

```

                                ORG 0F800H
                                MVI A,20H
                                STA 0FC00H
                                MVI A,01H
                                STA 0FC01H
                                LXI SP,0FB00H
                                LXI H,0F900H
                                PUSH H
                                LXI H,0FA00H

LOOP6:                          MVI A,01H
                                STA 0FC02H

LOOP4:                          MVI E,04H

LOOP3:                          MVI B,00H
                                MVI C,00H
                                MOV A,M
                                RLC
                                MOV M,A
                                MOV A,B
                                RAL
                                MOV B,A
                                MOV A,M
                                RLC
                                MOV M,A
                                MOV C,A
                                RAL
                                XRA B
                                MOV D,A
                                XTHL
                                MOV A,M
                                STC
                                CMC
                                RAL
                                ORA D
                                MOV M,A
                                LDA 0FC01H

LOOP1:                          INX H
                                DCR A
                                JNZ LOOP1
                                MOV A,M
                                STC
                                CMC
                                RAL
                                ORA B
                                LDA 0FC01H

```

```
LOOP2:   DCX H
         DCR A
         JNZ LOOP2
         XTHL
         DCR E
         JNZ LOOP3
         LDA 0FC02H
         INX H
         DCR A
         STA 0FC02H
         JNZ LOOP4
         LDA 0FC01H
         ADD A
         XTHL
```

```
LOOP5:   INXH
         DCR A
         JNZ LOOP5

         XTHL
         LDA 0FC00H
         DCR A
         STA 0FC00H
         JNZ LOOP6
         HLT
         END
```

For block-sizes higher than 16 bits, few modifications according to table 8.4 of chapter 8 are required to be made in this program.

25. 8-bit DEPS encryption

```
                ORG 0F800H
                LXI H,0F900H
                MVI E,40H

LOOP3:          MVI C,00H
                MVI D,08H

LOOP2:          STC
                CMC
                MOV A,M
                RAR
                JNC LOOP1
                INR A

LOOP1:          MOV M,A
                MOV A,C
                RAL
                MOV C,A
                DCR D
                JNZ LOOP2
                MOV A,M
                INX H
                DCR E
                JNZ LOOP3
                HLT
                END
```

26. 8-bit DEPS decryption

```
                ORG 0F800H
                LXI H,0F900H
                MVI E,40H

LOOP4:          MVI B,01H
                MVI D,08H
                MOV A,M

LOOP3:          STC
                CMC
                RAR
                MOV C,A
                MOV A,B
                JNC LOOP1
                CMC
                RAL
                DCR A
                JMP LOOP2

LOOP1:          RAL

LOOP2:          MOV B,A
                MOV A,C
                DCR D
                JNZ LOOP3
                MOV M,B
                INX H
                DCR E
                JNZ LOOP4
                HLT
                END
```

27. 16-bit (and higher) DEPS encryption

```

MVI A, 02H ; 04H/08H/10H/20H/40H
STA 0FE00H
MVI A, 20H ; 10H/08H/04H/02H/01H
STA 0FE01H
MVI A, 04H ; 08H/10H/20H/40H/80H
STA 0FE02H
LXI SP, 0FD00H
LXI H, 0F901H ; 0F903/0F907/0F90F/0F91F/0F93F
PUSH H
LXI H, 0FA00H

LOOP6: SHLD 0FC00H
LDA 0FE00H
MOVE E, A

LOOP3: MVI C, 00H
MVI D, 08H

LOOP2: MVI B, 02H ; 04H/08H/10H/20H/40H
LHLD 0FC00H
STC
CMC

LOOP: MOV A, M
RAR
MOV M, A
INX H
DCR B
JNZ LOOP
DCX H
JNC LOOP1
INR A

LOOP1: MOV M, A
MOV A, C
RAL
MOV C, A
DCR D
JNZ LOOP2
XTHL
MOV M, C

```

```
DCX H
XTHL
DCR E
JNZ LOOP3
LHLD 0FC00H
LDA 0FE00H

LOOP4:    INX H
          DCR A
          JNZ LOOP4
          LDA 0FE02H
          XTHL

LOOP5:    INX H
          DCR A
          JNZ LOOP5
          XTHL
          LDA 0FE01H
          DCR A
          STA 0FE01H
          JNZ LOOP6
          HLT
          END
```

Modifications required for 32/64/128/256/512-bit block-sizes, respectively, are given as comments with some instructions (to be modified) of this program.

28. 16-bit (and higher) DEPS decryption

```

MVI A, 02H ; 04H/08H/10H/20H/40H
STA 0FE00H
MVI A, 20H ; 10H/08H/04H/02H/01H
STA 0FE01H
MVI A, 04H ; 08H/10H/20H/40H/80H
STA 0FE02H
LXI SP, 0FD00H
LXI H, 0F901H ; 0F903/0F907/0F90F/0F91F/0F93F
PUSH H
LXI H, 0FA00H

LOOP6: SHLD 0FC00H
LDA 0FE00H
MOVE E, A
MVI D, 10H

LOOP2: MVI B, 02H ; 04H/08H/10H/20H/40H
LHLD 0FC00H
STC
CMC

LOOP: MOV A, M
RAR
MOV M, A
INX H
DCR B
JNZ LOOP
XTHL
JNC CY0
CMC
LDA 0FE00H
MOV B, A
MOV A, M

LOOPX: RAL
MOV M, A
DCX H
DCRB
JNZ LOOPX
LDA 0FE00H
MOV B, A

LOOPY: INX H
DCR B
JNZ LOOPY
LDA 0FE00H
MOV B, A

```

```
STC
CMC

LOOPZ:  MOV A, M
        SBI 01H
        MOV M, A
        DCX H
        DCR B
        JNZ LOOPZ
        JMP XYZ

CY0:    LDA 0FE00H
        MOV B, A
        MOV A, M

LOOPW:  RAL
        MOV M, A
        DCX H
        DCRB
        JNZ LOOPW

XYZ:    LDA 0FE00H

LOOP4:  INX H
        DCR A
        JNZ LOOP 4
        XTHL
        DCR D
        JNZ LOOP2
        LDA 0FE00H
        XTHL

LOOP3:  INX H
        DCR A
        JNZ LOOP3
        XTHL
        LDA 0FE01H
        DCR A
        STA 0FE01H
        JNZ LOOP6
        HLT
        END
```

Modifications required for 32/64/128/256/512-bit block-sizes, respectively, are given as comments with some instructions (to be modified) of this program.

Figures and Tables

Figures

<u>No.</u>	<u>Description</u>	<u>Page</u>
1.1	A structure of a typical cryptosystem	3
1.2	ECB mode of operation	17
1.3	CBC mode of operation	18
1.4	CFB mode of operation	19
1.5	OFB mode of operation	21
1.6	A microprocessor- based implementation	31
2.1	Trace of bit movements in PPOB	34
2.2	LSPB iterations on 8-bit block	35
2.3	RSPB iterations on 8-bit block	36
2.4	DSPB iterations on 8-bit block	37
2.5	CSPB iterations on 8-bit block	38
2.6	Comparison of character-frequencies in the source and encrypted .dll files	48
2.7	Comparison of character-frequencies in the source and encrypted .exe files	19
2.8	Comparison of character-frequencies in the source and encrypted .jpg files	50
2.9	Comparison of character-frequencies in the source and encrypted .txt files	51
2.10	Proposed algorithms vs. Triple DES in χ^2 -test of .dll files	52
2.11	Proposed algorithms vs. Triple DES in χ^2 -test of .exe files	53
2.12	Proposed algorithms vs. Triple DES in χ^2 -test of .jpg files	54
2.13	Proposed algorithms vs. Triple DES in χ^2 -test of .txt files	55
3.1	Character-frequencies in the source and encrypted .dll files	68
3.2	Character-frequencies in the source and encrypted .exe files	68
3.3	Character-frequencies in the source and encrypted .jpg files	69
3.4	Character-frequencies in the source and encrypted .txt files	69
3.5	BET vs. Triple DES in χ^2 -test of .dll files	71
3.6	BET vs. Triple DES in χ^2 -test of .exe files	72
3.7	BET vs. Triple DES in χ^2 -test of .jpg files	72
3.8	BET vs. Triple DES in χ^2 -test of .txt files	73
4.1	Transposition of bits in <i>Round 1</i> of SPOB	77
4.2	Character-frequencies in the source and encrypted .dll files	80
4.3	Character-frequencies in the source and encrypted .exe files	80
4.4	Character-frequencies in the source and encrypted .jpg files	81
4.5	Character-frequencies in the source and encrypted .txt files	81
4.6	SPOB vs. Triple DES in χ^2 -test of .dll files	83

4.7	SPOB vs. Triple DES in χ^2 -test of .exe files	83
4.8	SPOB vs. Triple DES in χ^2 -test of .jpg files	84
4.9	SPOB vs. Triple DES in χ^2 -test of .txt files	85
5.1	Character-frequencies in the source and encrypted .dll files	94
5.2	Character-frequencies in the source and encrypted .exe files	94
5.3	Character-frequencies in the source and encrypted .jpg files	95
5.4	Character-frequencies in the source and encrypted .txt files	95
5.5	MAT vs. Triple DES in χ^2 -test of .dll files	97
5.6	MAT vs. Triple DES in χ^2 -test of .exe files	97
5.7	MAT vs. Triple DES in χ^2 -test of .jpg files	98
5.8	MAT vs. Triple DES in χ^2 -test of .txt files	99
6.1	Character-frequencies in the source and encrypted .dll files	106
6.2	Character-frequencies in the source and encrypted .exe files	106
6.3	Character-frequencies in the source and encrypted .jpg files	107
6.4	Character-frequencies in the source and encrypted .txt files	107
6.5	OMAT vs. Triple DES in χ^2 -test of .dll files	109
6.6	OMAT vs. Triple DES in χ^2 -test of .exe files	109
6.7	OMAT vs. Triple DES in χ^2 -test of .jpg files	110
6.8	OMAT vs. Triple DES in χ^2 -test of .txt files	111
7.1	Character-frequencies in the source and encrypted .dll files	119
7.2	Character-frequencies in the source and encrypted .exe files	119
7.3	Character-frequencies in the source and encrypted .jpg files	120
7.4	Character-frequencies in the source and encrypted .txt files	120
7.5	MMAT vs. Triple DES in χ^2 -test of .dll files	122
7.6	MMAT vs. Triple DES in χ^2 -test of .exe files	123
7.7	MMAT vs. Triple DES in χ^2 -test of .jpg files	123
7.8	MMAT vs. Triple DES in χ^2 -test of .txt files	124
8.1	Computation of the Front and Rear bits in the first iteration	128
8.2	Character-frequencies in the source and encrypted .dll files	133
8.3	Character-frequencies in the source and encrypted .exe files	133
8.4	Character-frequencies in the source and encrypted .jpg files	134
8.5	Character-frequencies in the source and encrypted .txt files	134
8.6	BOS vs. Triple DES in χ^2 -test of .dll files	136
8.7	BOS vs. Triple DES in χ^2 -test of .exe files	136
8.8	BOS vs. Triple DES in χ^2 -test of .jpg files	137
8.9	BOS vs. Triple DES in χ^2 -test of .txt files	138
9.1	Cipher-text generation using DEPS on 8-bit data	141
9.2	Character-frequencies in the source and encrypted .dll files	148
9.3	Character-frequencies in the source and encrypted .exe files	148

9.4	Character-frequencies in the source and encrypted .jpg files	149
9.5	Character-frequencies in the source and encrypted .txt files	149
9.6	BOS vs. Triple DES in χ^2 -test of .dll files	151
9.7	BOS vs. Triple DES in χ^2 -test of .exe files	152
9.8	BOS vs. Triple DES in χ^2 -test of .jpg files	152
9.9	BOS vs. Triple DES in χ^2 -test of .txt files	153
10.1	Character-frequencies in the source and encrypted .dll files	157
10.2	Character-frequencies in the source and encrypted .exe files	157
10.3	Character-frequencies in the source and encrypted .jpg files	158
10.4	Character-frequencies in the source and encrypted .txt files	158
10.5	OMAT+BET vs. Triple DES in χ^2 -test of .dll files	159
10.6	OMAT+BET vs. Triple DES in χ^2 -test of .exe files	160
10.7	OMAT+BET vs. Triple DES in χ^2 -test of .jpg files	161
10.8	OMAT+BET vs. Triple DES in χ^2 -test of .txt files	161
10.9	Character-frequencies in the source and encrypted .dll files	164
10.10	Character-frequencies in the source and encrypted .exe files	164
10.11	Character-frequencies in the source and encrypted .jpg files	165
10.12	Character-frequencies in the source and encrypted .txt files	165
10.13	MMAT+DSPB vs. Triple DES in χ^2 -test of .dll files	166
10.14	MMAT+DSPB vs. Triple DES in χ^2 -test of .exe files	167
10.15	MMAT+DSPB vs. Triple DES in χ^2 -test of .jpg files	167
10.16	MMAT+DSPB vs. Triple DES in χ^2 -test of .txt files	168

Tables

<u>No.</u>	<u>Description</u>	<u>Page</u>
1.1	Vigenère Tableau	11
1.2	Examples of embedded systems.	23
2.1	Encryption/Decryption process of 8-bit block using LSPB	36
2.2	Encryption/Decryption process of 8-bit block using RSPB	37
2.3	Encryption/Decryption process of 8-bit block using DSPB	38
2.4	Encryption/Decryption process of 8-bit block using CSPB	39
2.5	Number of iterations for a complete cycle	39
2.6	χ^2 -test with .dll files	52
2.7	Encryption time with .dll files	52
2.8	χ^2 -test with .exe files	53
2.9	Encryption time with .exe files	53
2.10	χ^2 -test with .jpg files	54
2.11	Encryption time with .jpg files	54
2.12	χ^2 -test with .txt files	55
2.13	Encryption time with .txt files	55

2.14	Avalanche and runs in LSPB	56
2.15	Avalanche and runs in RSPB	57
2.16	Avalanche and runs in DSPB	58
2.17	Avalanche and runs in CSPB	59
3.1	Number of iterations for a complete cycle	62
3.2	χ^2 -test for BET with .dll files	71
3.3	χ^2 -test for BET with .exe files	71
3.4	χ^2 -test for BET with .jpg files	72
3.5	χ^2 -test for BET with .txt files	73
3.6	Avalanche and runs in BET	74
4.1	Regeneration of the original block	78
4.2	Number of iterations for a complete cycle	78
4.3	χ^2 -test for SPOB with .dll files	82
4.4	χ^2 -test for SPOB with .exe files	83
4.5	χ^2 -test for SPOB with .jpg files	84
4.6	χ^2 -test for SPOB with .txt files	84
4.7	Avalanche and runs in SPOB	86
5.1	Amendments for MAT encryption with higher block-sizes	92
5.2	χ^2 -test for MAT with .dll files	96
5.3	χ^2 -test for MAT with .exe files	97
5.4	χ^2 -test for MAT with .jpg files	98
5.5	χ^2 -test for MAT with .txt files	98
5.6	Avalanche and runs in MAT	100
6.1	Amendments for OMAT encryption with higher block-sizes	104
6.2	χ^2 -test for OMAT with .dll files	108
6.3	χ^2 -test for OMAT with .exe files	109
6.4	χ^2 -test for OMAT with .jpg files	110
6.5	χ^2 -test for OMAT with .txt files	110
6.6	Avalanche and runs in OMAT	112
7.1	Amendments for MMAT with higher block-sizes	118
7.2	χ^2 -test for MMAT with .dll files	122
7.3	χ^2 -test for MMAT with .exe files	122
7.4	χ^2 -test for MMAT with .jpg files	123
7.5	χ^2 -test for MMAT with .txt files	124
7.6	Avalanche and runs in MMAT	125
8.1	Computation fo front and rear bits	128
8.2	Iterations for an 8-bit block	129
8.3	Number of iterations needed to form a cycle	129
8.4	Amendments for BOS with higher block-sizes	132
8.5	χ^2 -test for BOS with .dll files	135

8.6	χ^2 -test for BOS with .exe files	136
8.7	χ^2 -test for BOS with .jpg files	137
8.8	χ^2 -test for BOS with .txt files	137
8.9	Avalanche and runs in BOS	139
9.1	χ^2 -test for DEPS with .dll files	150
9.2	χ^2 -test for DEPS with .exe files	151
9.3	χ^2 -test for DEPS with .jpg files	152
9.4	χ^2 -test for DEPS with .txt files	153
9.5	Avalanche and runs in DEPS	154
10.1	χ^2 -test for OMAT+BET with .dll files	159
10.2	χ^2 -test for OMAT+BET with .exe files	160
10.3	χ^2 -test for OMAT+BET with .jpg files	160
10.4	χ^2 -test for OMAT+BET with .txt files	161
10.5	Avalanche and runs in OMAT+BET	162
10.6	χ^2 -test for OMAT+BET with .dll files	166
10.7	χ^2 -test for OMAT+BET with .exe files	166
10.8	χ^2 -test for OMAT+BET with .jpg files	167
10.9	χ^2 -test for OMAT+BET with .txt files	168
10.10	Avalanche and runs in OMAT+BET	169
11.1	Average time required for exhaustive key search	170
11.2	Formation of key in format 1	172
11.3	Formation of key in format 2	173
11.4	Brute force attack on the proposed key formats	178
12.1	Comparison of character frequencies for .dll file	180
12.2	Comparison of character frequencies for .exe file	180
12.3	Comparison of character frequencies for .jpg file	181
12.4	Comparison of character frequencies for .txt file	181
12.5	Comparison of χ^2 values for .dll files	182
12.6	Comparison of χ^2 values for .exe files	182
12.7	Comparison of χ^2 values for .jpg files	183
12.8	Comparison of χ^2 values for .txt files	183
12.9	Comparison of encryption time (in secs.)	184

References

1. Schneier, B., *Applied Cryptography Second Edition: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, USA, 1996.
2. Schneier, B. and Ferguson, N., *Practical Cryptography*, Wiley Dreamtech India, New Delhi, 2003.
3. Stallings, W., *Cryptography and Network Security: Principles and Practices*, Pearson Education, Delhi, 2003.
4. Stallings, W., *Network Security Essentials: Applications and Standards*, Pearson Education, Delhi, 2003.
5. Kahate, A., *Cryptography and Network Security*, Tata McGraw-Hill, New Delhi, 2003.
6. Bishop, M., *Computer Security: Art and Science*, Pearson Education, Delhi, 2003.
7. Pfleeger, C. P. and Pfleeger, S. L., *Security in Computing*, Pearson Education, Delhi, 2003.
8. Smith, R. E., *Internet Cryptography*, Pearson Education, Delhi, 2000.
9. Hook, D., *Beginning Cryptography with Java*, Wiley Dreamtech India, New Delhi, 2005.
10. Rhee, M. Y., *Cryptography and Secure Communications*, McGraw-Hill, Singapore, 1994.
11. Mal, S., Mandal, J. K. and Dutta, S., *A microprocessor-Based Encoder for Secured Transmission*, Proceedings of the National Conference on Intelligent Computing on VLSI, Kalyani Govt. Engg. College, 16-17 February, 2001, pp 164-169.
12. Mal, S. and Mandal, J. K., *A Cascaded Technique of Encryption Realized Using a Microprocessor-Based System*, AMSE Journal, Vol. 7, No. 2, 2002, pp 25-35.
13. Mal, S., Mandal, J. K. and Dutta, S., *A Microprocessor-Based Cascaded Technique of Encryption*, Proceedings of XXXVIth Annual Convention of CSI-2001, Kolkata, 20-24 November, 2001, pp C269-275.
14. Mandal, J. K. and Dutta, S., *A Universal Encryption Technique*, Proceedings of National Conference of Networking of Machines, Microprocessors, IT and HRD – Need of the Nation in the Next Millennium, Kalyani, November 25-26, 1999, pp. B114-B120.

15. Mandal, J. K. and Dutta, S., *A Universal Bit-Level Encryption Technique*, Proceedings of VIIth State Science and Technology Congress, Jadavpur University, February 2000.
16. Mandal, J. K. and Mal, S., *A Cascaded Technique of Encryption Realized using a Microprocessor-Based System*, AMSE, France (accepted for publication).
17. Brickell, E. F., *Survey of Hardware Implementation of RSA*, Proceedings of Advances in Cryptography – CRYPTO '89, Springer Verlag, 1990, pp 368-370.
18. Kamal, R., *Embedded Systems: Architecture, Programming and Design*, Tata McGraw-Hill, New Delhi, 2003.
19. Lewis, D. W., *Fundamentals of Embedded Software: Where C and Assembly Meet*, PHI, New Delhi, 2002.
20. Srinath, N. K., *8085 Microprocessor: Programming and Interfacing*, PHI, New Delhi, 2005.
21. Rafiquzzaman, M., *Microprocessors: Theory and Applications - Intel and Motorola*, PHI, New Delhi, 1999.
22. Mathur, A. P., *Introduction to Microprocessors, 3rd Edition*, Tata McGraw-Hill, New Delhi, 1989.
23. Irvine, K. R., *Assembly Language for Intel-Based Computer*, Pearson Education, Delhi, 2005.
24. R. S. Gaonkar, *Microprocessor Architecture: Programming, and Applications with the 8085, 4th Edition*, Penram (India), 2000.
25. Leventhal, L., *Introduction to Microprocessors: Software, Hardware, Programming*, PHI, New Delhi, 1988.
26. Pal, A., *Microprocessors: Principles and Applications*, Tata McGraw-Hill, New Delhi, 1990.
27. Abel, P., *IBM PC Assembly Language and Programming*, PHI, New Delhi, 2000.
28. Ram, B., *Advanced Microprocessors and Interfacing*, Tata McGraw-Hill, New Delhi, 2002.
29. Ray, A. K. and Bhurchandi, K. M., *Advanced Microprocessors and Peripherals*, Tata McGraw-Hill, New Delhi, 2000.
30. Spiegel, M. R., Schiller, J. and Srinivasan, R. A., *Probability and Statistics*, Tata McGraw-Hill, New Delhi, 2004.

31. Hogg, R. V. and Craig, A. T., *Introduction to Mathematical Statistics*, Pearson Education, Delhi, 2005.
32. Goon, A. M., Gupta, M. K. and Dasgupta, B., *Fundamentals of Statistics, Vol. I*, The World Press Ltd., 1993.
33. Deitel, H. M. and Deitel P. J., *C: How to Program*, Pearson Education, Delhi, 2006.
34. Kamthane, A. N., *Programming with ANSI and Turbo C*, Pearson Education, Delhi, 2002.
35. Gustafson, H. M., Dawson, E. P. and Golic, J. Dj., *Automated Statistical Methods for Measuring the Strength of Block Ciphers*, Statistics and Computing, Vol. 7, 1997, pp. 125-135.
36. Moliner, R. J. D., *On the Statistical Testing of Block Ciphers*, Ph. D. Dissertation, Swiss Federal Institute of Technology, Zurich, 1999.
37. Toz, D., Doğanaksoy, A. and Turan, M. S., *Statistical Analysis of Block Ciphers*, downloaded from <http://www.metu.edu.tr>
38. <http://www.williamstallings.com/Extras/Security-Notes/lectures/blockA.html>
39. <http://www.williamstallings.com/Extras/Security-Notes/lectures/blockB.html>
40. <http://www2.mat.dtu.dk/people/Lars.R.Knudsen/bc.html>

Note: Those, which have not been directly referred to in the text, have been taken as help during designing, programming and implementing the proposed algorithms.

List of Publications by the Author

1. Sinha, S., Mandal, J. K. and Mal, S., *A Microprocessor Based Encoder using Cascaded Orientation of Bits (COB)*, proceedings of the National Seminar RIT-2003, CMRI, Dhanbad, February, 2003.
2. Mal, S., Mandal, J. K., Chatterjee, S. and Sinha, S., *A Microprocessor-Based Encoder Through Transposition of Bits*, proceedings of the International Symposium on "Information Technology: Emerging Trends", IIIT, Allahbad, September 19-21, 2003, pp 17-27.
3. Sinha, S., Mandal, J. K., and Mal, S., *Microprocessor Based Encoder Through Selective Positional Orientation of Bits (SPOB)*, proceedings of the International Conference on "Recent Trends and New Directions of Research in Cybernetics & Systems Theory", IASST, Guwahati, January, 2004.
4. Sinha, S., Mandal, J. K., and Chakraborty, R., *A Microprocessor-based Block Cipher through Overlapped Modulo Arithmetic Technique (OMAT)*, proceedings of the 12th International Conference on Advanced Computing & Communication – ADCOM 2004, Ahmedabad, December 15-18, 2004, pp 276-280.
5. Sinha S., Mandal, J. K. and Chakraborty, R., *A Microprocessor-based Bit-Level Cryptosystem through Arithmetic Manipulation of Blocks (AMB)*, Journal of The Institute of Engineering, Vol. 4, No. 1, TUTA/IOE/PCU, Tribhuvan University, Nepal, December, 2004, pp 1-7.
6. Sinha, S., Mandal, J. K. and Saha, S., *A Microprocessor-based Block Cipher through Bit-pair Operation and Separation (BOS)*, proceedings of the International Conference on Information and Communication Technology – ICT Conference 2006, BICC, Kathmandu, Nepal, March 23-26, 2006.
7. Sinha, S., Mandal, J. K., *A Microprocessor-based Block Cipher through Decimal Equivalent Positional Substitutions (DEPS)*, proceedings of the National Conference on Recent Trends in Intelligent Computing: RTIC-06, Dept. of Computer Science and Engineering, Kalyani Govt. Engineering College, Kalyani, West Bengal, November 17-19, 2006, pp 197-204.

