

# **AN APPROACH TOWARDS DEVELOPMENT OF EFFICIENT ENCRYPTION TECHNIQUES**

11851



**Saurabh Dutta**

A THESIS SUBMITTED  
TO  
THE UNIVERSITY OF NORTH BENGAL  
FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY (SCIENCE)  
2004

Ref.

005.052

H 825 d

**172310**

13 JUN 2005



STOCKTAKING-2011

ANSWER

*To the Soul of my Father,  
Bathed his Last,  
Dreaming of Me to be a Doctorate*

**DEPARTMENT OF COMPUTER SCIENCE & APPLICATION**  
**UNIVERSITY OF NORTH BENGAL**

**Dr. J. K. Mandal**

M. Tech (Comp. Engg.), Ph. D.(Engg.)

**Head**



P.O. North Bengal University  
Dist. Darjeeling, West Bengal  
Pin. 734430  
Phone: (0353) 258 2113(o)  
258 1137(r)  
Fax. : (0353) 258 1546  
Email: slg\_jkmandal@sancharnet.in  
nbucompsc@sancharnet.in

Ref. No.....

Dated: 28.06.2004

## Certificate

*This is to certify that the thesis entitled "An Approach towards Development of Efficient Encryption Techniques" submitted by Sri Saurabh Dutta, who got his name registered on May 03, 2000 for the award of Doctor of Philosophy (Ph. D.) in Science of the University of North Bengal, is absolutely based upon his own work under my supervision and neither the thesis nor any part of it has been submitted for any degree/diploma or any other academic award anywhere before.*

  
\_\_\_\_\_  
**(Dr. JYOTSNA KUMAR MANDAL)**

## **Declaration**

*I hereby declare that neither the thesis nor any part thereof has been submitted for any degree whatsoever.*

Saurabh Dutta  
28.06.04

---

**(SAURABH DUTTA)**

## Acknowledgement

It is the full encouragement and moral support way back in early 1999 from respectable Dr. Jyotsna Kumar Mandal, Reader, Department of Computer Science and Application, The University of North Bengal, due to which I involved myself in this world of research in the field of Data Encryption under his supervision. Today, at the end of this journey, I feel to be proud of being in association of such an inspirational character like Dr. Mandal. His ability to motivate me in successfully searching for new encryption techniques, his cheerful guidance towards analyzing and implementing different proposed techniques, his patient companion in composing this voluminous dissertation will be kept in the cage of my memory for ever.

Whatever mere quality I posses to carry on such a robust work is the result of the careful, thoughtful guidance that I got from my parents throughout my childhood and days of higher education. Their blessing is my biggest asset and my gratefulness towards them is too high to be expressed perfectly. The completion of this work during the lifetime of my father could have been my biggest moral success.

During the last half of my research work, the kind of encouraging support that I got and the kind of invaluable sacrifice that I have seen from my beloved wife, Piali, are beyond imagination. I feel fortunate enough to be accompanied by someone like her.

Technically, I am grateful to respectable Mr. Satadal Mal, Assistant Professor, Jalpaiguri Government Engineering College. His useful tips and co-operation had a great influence in conducting the work successfully.

Mr. Pramod Biswal and Mr. Narendra Nath Patra, both deserve to be thanked for their co-operation.

I am highly grateful to the Principal, Gomendra Multiple College, Nepal, for the financial stability that I was offered by him and the kind of technical infrastructure he provided to me to complete the coding and the composition of this dissertation.

I was impressed in watching the keen interest on every single step of my work in my all other family members and in many of my well wishing relatives, too numerous to be mentioned individually. All these worked like an indirect stimulator to me.

The kind of affection I have received from Mrs. Mandal during the huge period of time when I worked together with Dr. Mandal is an unforgettable experience. She deserves to be thanked a lot.

It also has been an unknowing sacrifice by my 18-month-old sweet kid, Jishu, who missed my accompany a lot during this period.

Saurabh Dutta  
28.06.04

---

(SAURABH DUTTA)

# Contents

## Chapter 1: Introduction

Topic	Page No.
<b>1.1 Introduction</b>	<b>3</b>
<b>1.2 Cryptosystems</b>	<b>4</b>
<b>1.2.1 Definition of Cryptosystem</b>	<b>4</b>
<b>1.2.2 Secret Key Cryptosystem</b>	<b>5</b>
<b>1.2.2.1 Requirements for Secret Key Cryptosystem</b>	<b>6</b>
<b>1.2.2.2 Evaluating a Cryptographic System</b>	<b>8</b>
<b>1.2.2.3 Attacking a Conventional Encryption Scheme</b>	<b>9</b>
<b>1.2.2.4 The Condition for an Encryption Scheme to be Unconditionally Secure</b>	<b>9</b>
<b>1.2.2.5 The Condition for an Encryption Scheme to be Computationally Secure</b>	<b>10</b>
<b>1.2.2.6 Basic Building Blocks in Classical Encryption Techniques</b>	<b>11</b>
<b>1.2.2.6.1 Principles of Substitution Techniques</b>	<b>11</b>
<b>1.2.2.6.1.1 Caesar Cipher</b>	<b>12</b>
<b>1.2.2.6.1.2 Monoalphabetic Substitution Cipher</b>	<b>13</b>
<b>1.2.2.6.1.3 Homophonic Substitution Cipher</b>	<b>13</b>
<b>1.2.2.6.1.4 Playfair Cipher</b>	<b>13</b>
<b>1.2.2.6.1.5 Polyalphabetic Cipher</b>	<b>14</b>
<b>1.2.2.6.1.6 Vigenere Cipher</b>	<b>15</b>
<b>1.2.2.6.1.7 Enhancement from Vigenere Cipher</b>	<b>17</b>
<b>1.2.2.6.1.8 Some Other Substitution Ciphers</b>	<b>18</b>
<b>1.2.2.6.2 Principles of Transposition Techniques</b>	<b>18</b>

<b>Topic</b>	<b>Page No.</b>
<b>1.2.2.6.2.1 The Rail Fence Technique</b>	<b>18</b>
<b>1.2.2.6.2.1.1 A Cascaded Approach</b>	<b>19</b>
<b>1.2.2.6.3 Rotor Machines – Composite Substitution and Transposition Cipher</b>	<b>20</b>
<b>1.2.3 Public Key Cryptosystem</b>	<b>21</b>
<b>1.2.3.1 The Algorithm</b>	<b>22</b>
<b>1.2.3.2 Characteristics of Public Key Cryptography</b>	<b>25</b>
<b>1.3 Evolution in the Field of Cryptography</b>	<b>25</b>
<b>1.4 Some of the Existing Techniques</b>	<b>28</b>
<b>1.4.1 The Data Encryption Standard (DES)</b>	<b>28</b>
<b>1.4.1.1 Basic Principles of DES</b>	<b>28</b>
<b>1.4.1.2 The Basic Algorithm</b>	<b>29</b>
<b>1.4.1.2.1 The Function f</b>	<b>30</b>
<b>1.4.1.3 Applications</b>	<b>30</b>
<b>1.4.1.3.1 Modes of Operations</b>	<b>31</b>
<b>1.4.1.4 The DES Controversy</b>	<b>31</b>
<b>1.4.2 The RSA Technique – The Special Characteristics</b>	<b>31</b>
<b>1.4.2.1 The RSA Algorithm</b>	<b>31</b>
<b>1.4.2.2 The Security of RSA</b>	<b>34</b>
<b>1.5 An Overview of Proposed Techniques</b>	<b>34</b>
<b>1.5.1 An Overview of the RPSP Technique</b>	<b>37</b>
<b>1.5.2 An Overview of the TE Technique</b>	<b>38</b>
<b>1.5.3 An Overview of the RPPO Technique</b>	<b>40</b>
<b>1.5.4 An Overview of the RPMS Technique</b>	<b>41</b>
<b>1.5.5 An Overview of the RSBP Technique</b>	<b>42</b>
<b>1.5.6 An Overview of the RSBM Technique</b>	<b>43</b>
<b>1.5.7 A Proposal on Key Structures for Proposed Techniques</b>	<b>44</b>

<b>Topic</b>	<b>Page No.</b>
<b>1.5.7.1 Proposed Key Structure for Block Ciphers with Direct Block-to-Block Conversion</b>	<b>45</b>
<b>1.5.7.2 Proposed Key Structure for Block Ciphers with Option-based Block-to-Block Conversion</b>	<b>45</b>
<b>1.5.7.3 Proposed Key Structure for Block Ciphers with Non-Contiguous Bit Allocation</b>	<b>46</b>
<b>1.5.7.4 Proposed Key Structure for Block Ciphers with Repeated Block-to-Block Conversion</b>	<b>46</b>
<b>1.5.8 Factors Considered for Evaluating Proposed Techniques</b>	<b>46</b>
<b>1.5.8.1 Frequency Distribution Test</b>	<b>46</b>
<b>1.5.8.2 Chi Square Test</b>	<b>47</b>
<b>1.5.8.3 Analysis of the Key Space</b>	<b>47</b>
<b>1.5.8.4 Computation of Encryption/Decryption Time</b>	<b>48</b>
<b>1.5.8.5 Comparison of Performance with RSA Technique</b>	<b>48</b>
<b>1.6 A Note on Merits of Proposed Techniques</b>	<b>48</b>

## Chapter 2: Encryption Through Recursive Positional Substitution based on Prime-Nonprime (RPSP) of Cluster

<b>Topic</b>	<b>Page No.</b>
<b>2.1 Introduction</b>	<b>53</b>
<b>2.2 The Scheme</b>	<b>54</b>
<b>2.2.1 Example</b>	<b>56</b>
<b>2.2.1.1 Example of Encryption</b>	<b>59</b>
<b>2.2.1.2 Example of Decryption</b>	<b>59</b>
<b>2.3 Implementation</b>	<b>59</b>
<b>2.4 Results</b>	<b>66</b>

Topic	Page No.
<b>2.4.1 Result for Encryption/Decryption Time and Chi Square</b>	
<b>Value</b>	<b>66</b>
<b>2.4.1.1 Result for EXE Files</b>	<b>67</b>
<b>2.4.1.2 Result for COM Files</b>	<b>68</b>
<b>2.4.1.3 Result for DLL Files</b>	<b>69</b>
<b>2.4.1.4 Result for SYS Files</b>	<b>70</b>
<b>2.4.1.5 Result for CPP Files</b>	<b>72</b>
<b>2.4.1.6 Report on Variation of Encryption Time with Varying Block Sizes</b>	<b>73</b>
<b>2.4.2 Results for Frequency Distribution Tests</b>	<b>75</b>
<b>2.4.3 Comparison with RSA Technique</b>	<b>78</b>
<b>2.5 Analysis</b>	<b>80</b>
<b>2.5.1 Proof of Cycle Formation</b>	<b>80</b>
<b>2.5.2 Analysis on Block Size</b>	<b>81</b>
<b>2.5.3 Analysis on Factors Considered for Evaluation Purpose</b>	<b>84</b>
<b>2.6 Conclusion</b>	<b>87</b>

## Chapter 3: Encryption Through Triangular Encryption (TE) Technique

Topic	Page No.
<b>3.1 Introduction</b>	<b>90</b>
<b>3.2 The Scheme</b>	<b>91</b>
<b>3.2.1 Formation of Triangle</b>	<b>91</b>
<b>3.2.2 Options for Forming Target Blocks from Triangle</b>	<b>93</b>
<b>3.2.3 Generating Source Block from a Target Block</b>	<b>94</b>
<b>3.2.3.1 Generating Source Block from Target Block <math>s^{n-1}_0 s^{n-2}_0 s^{n-3}_0 s^{n-4}_0 s^{n-5}_0 \dots s^1_0 s^0_0</math> (With Option Serial No. 010)</b>	<b>94</b>

Topic	Page No.
<b>3.2.3.2 Generating Source Block from Target Block  <math>s_{n-1}^0 s_{n-2}^1 s_{n-3}^2 s_{n-4}^3 s_{n-5}^4 \dots s_1^{n-2} s_0^{n-1}</math> (With Option  Serial No. 011)</b>	<b>95</b>
<b>3.2.4 A Sample Example to Illustrate the Scheme</b>	<b>96</b>
<b>3.3 Implementation</b>	<b>98</b>
<b>3.4 Results</b>	<b>116</b>
<b>    3.4.1 Result for Encryption/Decryption Time and Chi Square  Value</b>	<b>116</b>
<b>        3.4.1.1 Result for EXE Files</b>	<b>117</b>
<b>        3.4.1.2 Result for COM Files</b>	<b>118</b>
<b>        3.4.1.3 Result for DLL Files</b>	<b>120</b>
<b>        3.4.1.4 Result for SYS Files</b>	<b>121</b>
<b>        3.4.1.5 Result for CPP Files</b>	<b>123</b>
<b>    3.4.2 Results for Frequency Distribution Tests</b>	<b>124</b>
<b>    3.4.3 Comparison of TE with RSA Technique</b>	<b>127</b>
<b>3.5 Analysis and Conclusion including Comparison with RPSP</b>	<b>129</b>

## Chapter 4: Encryption Through Recursive Paired Parity Operation (RPPO)

Topic	Page No.
<b>4.1 Introduction</b>	<b>132</b>
<b>4.2 The Scheme</b>	<b>133</b>
<b>    4.2.1 Example</b>	<b>134</b>
<b>4.3 Implementation</b>	<b>136</b>
<b>4.4 Results</b>	<b>143</b>
<b>    4.4.1 Result of Encryption/Decryption Time, Total Number of  Operations, Chi Square Value</b>	<b>143</b>

Topic	Page No.
<b>4.4.1.1 Result for <i>EXE</i> Files</b>	<b>144</b>
<b>4.4.1.2 Result for <i>COM</i> Files</b>	<b>145</b>
<b>4.4.1.3 Result for <i>DLL</i> Files</b>	<b>146</b>
<b>4.4.1.4 Result for <i>SYS</i> Files</b>	<b>148</b>
<b>4.4.1.5 Result for <i>CPP</i> Files</b>	<b>150</b>
<b>4.4.2 Results of Frequency Distribution Tests</b>	<b>151</b>
<b>4.4.3 Comparison with RSA Technique</b>	<b>154</b>
<b>4.5 Analysis and Conclusion including Comparison with RPSP, TE</b>	<b>156</b>
<b>4.5.1 Comparative Analysis with RPSP and TE Techniques</b>	<b>156</b>
<b>4.5.2 Formation of Cycle</b>	<b>156</b>
<b>4.5.3 Proof of the Finiteness in Re-generating Source Block</b>	<b>159</b>
<b>4.5.3.1 Proof for Block Size of 2 Bits</b>	<b>159</b>
<b>4.5.3.2 Proof for Block Size of 3 Bits</b>	<b>159</b>
<b>4.5.3.3 Proof for Block Size of 4 Bits</b>	<b>159</b>
<b>4.5.4 A Conclusive Analysis of Different results Obtained</b>	<b>160</b>

## Chapter 5: Encryption Through Recursive Positional Modulo- 2 Substitution (RPMS) Technique

Topic	Page No.
<b>5.1 Introduction</b>	<b>166</b>
<b>5.2 The Scheme</b>	<b>167</b>
<b>5.2.1 The Encryption</b>	<b>167</b>
<b>5.2.2 The Decryption</b>	<b>171</b>
<b>5.3 Implementation</b>	<b>174</b>
<b>5.3.1 The Process of Encryption</b>	<b>174</b>
<b>5.3.2 The Process of Decryption</b>	<b>177</b>
<b>5.4 Results</b>	<b>178</b>
<b>5.4.1 Computing Encryption/Decryption Time</b>	<b>179</b>

Topic	Page No.
<b>5.4.1.1 Result for <i>EXE</i> Files</b>	<b>179</b>
<b>5.4.1.2 Result for <i>COM</i> Files</b>	<b>180</b>
<b>5.4.1.3 Result for <i>DLL</i> Files</b>	<b>182</b>
<b>5.4.1.4 Result for <i>SYS</i> Files</b>	<b>183</b>
<b>5.4.1.5 Result for <i>CPP</i> Files</b>	<b>185</b>
<b>5.4.1.6 Discussion on Chi Square Tests</b>	<b>186</b>
<b>5.4.2 Result on Frequency Distribution Tests</b>	<b>188</b>
<b>5.4.3 Comparison with RSA Technique</b>	<b>191</b>
<b>5.5 Analysis and Conclusion including Comparison with RPSP, TE, RPPO</b>	<b>193</b>

## Chapter 6: Encryption Through Recursive Substitution of Bits Through Prime-Nonprime (RSBP) Detection of Sub-stream

Topic	Page No.
<b>6.1 Introduction</b>	<b>197</b>
<b>6.2 The Scheme</b>	<b>198</b>
<b>6.2.1 The Encryption Technique</b>	<b>198</b>
<b>6.2.2 The Decryption Technique</b>	<b>202</b>
<b>6.3 Implementation</b>	<b>206</b>
<b>6.3.1 Implementation of Encryption Technique of RSBP</b>	<b>207</b>
<b>6.3.2 Implementation of Decryption Technique of RSBP</b>	<b>210</b>
<b>6.4 Results</b>	<b>217</b>
<b>6.4.1 Result of Encryption/Decryption Time, Chi Square value and File Size Alteration</b>	<b>218</b>
<b>6.4.1.1 Result for <i>EXE</i> Files</b>	<b>218</b>
<b>6.4.1.2 Result for <i>COM</i> Files</b>	<b>219</b>
<b>6.4.1.3 Result for <i>DLL</i> Files</b>	<b>221</b>

Topic	Page No.
<b>6.4.1.4 Result for <i>SYS</i> Files</b>	<b>223</b>
<b>6.4.1.5 Result for <i>CPP</i> Files</b>	<b>225</b>
<b>6.4.2 Result for Frequency Distribution Tests</b>	<b>226</b>
<b>6.4.3 Comparison with RSA Technique</b>	<b>229</b>
<b>6.5 Analysis and Conclusion including Comparison with RPSP, TE, RPPO, RPMS</b>	<b>231</b>

## **Chapter 7: Encryption Through Recursive Substitution of Bits Through Modulo-2 (RSBM) Detection of Sub- stream**

Topic	Page No.
<b>7.1 Introduction</b>	<b>235</b>
<b>7.2 The Scheme</b>	<b>236</b>
<b>7.2.1 The Encryption Technique</b>	<b>236</b>
<b>7.2.2 The Decryption Technique</b>	<b>240</b>
<b>7.3 Implementation</b>	<b>244</b>
<b>7.4 Results</b>	<b>252</b>
<b>7.4.1 Result of Encryption/Decryption Time and Chi Square Value</b>	<b>253</b>
<b>7.4.1.1 Result for <i>EXE</i> Files</b>	<b>253</b>
<b>7.4.1.2 Result for <i>COM</i> Files</b>	<b>254</b>
<b>7.4.1.3 Result for <i>DLL</i> Files</b>	<b>256</b>
<b>7.4.1.4 Result for <i>SYS</i> Files</b>	<b>257</b>
<b>7.4.1.5 Result for <i>CPP</i> Files</b>	<b>258</b>
<b>7.4.2 Result of Frequency Distribution Tests</b>	<b>259</b>
<b>7.4.3 Comparison with RSA Technique</b>	<b>262</b>
<b>7.5 Analysis and Conclusion including Comparison with RPSP, TE, RPPO, RPMS, RSBP</b>	<b>264</b>

## **Chapter 8: Formation of Secret Key**

<b>Topic</b>	<b>Page No.</b>
<b>8.1 Introduction</b>	<b>268</b>
<b>8.2 Proposed Key Structures</b>	<b>268</b>
<b>8.2.1 Proposed Key Structure for RPSP and RPPO Techniques</b>	<b>269</b>
<b>8.2.2 Proposed Key Structure for TE Technique</b>	<b>270</b>
<b>8.2.3 Proposed Key Structure for RSBP Technique</b>	<b>272</b>
<b>8.2.4 Proposed Key Structure for RPMS and RSBM Techniques</b>	<b>274</b>
<b>8.3 Conclusion</b>	<b>276</b>

## **Chapter 9: Encryption Through Cascaded Implementation of Proposed Techniques**

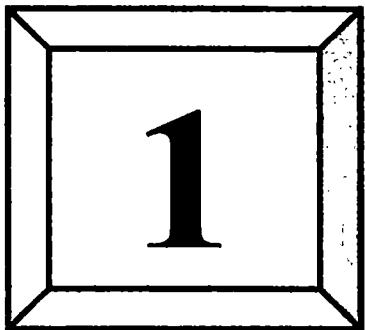
<b>Topic</b>	<b>Page No.</b>
<b>9.1 Introduction</b>	<b>280</b>
<b>9.1.1 Basic Principle of Cascading</b>	<b>280</b>
<b>9.1.2 Strength of the Cascaded Approach</b>	<b>281</b>
<b>9.2 Implementation</b>	<b>282</b>
<b>9.2.1 Encrypting Source File using Proposed Techniques in Cascaded Manner</b>	<b>282</b>
<b>9.2.2 Decrypting Encrypted File, Encrypted in Section 9.2.1</b>	<b>284</b>
<b>9.2.3 Analysis of Implementation</b>	<b>286</b>
<b>9.3 Results</b>	<b>288</b>
<b>9.4 Analysis</b>	<b>293</b>
<b>9.5 Proposal of An Integrated Encryption System</b>	<b>293</b>
<b>9.5.1 Principles of the Proposed Integrated System</b>	<b>294</b>
<b>9.5.2 Schematic Characteristics of the Proposed Integrated System</b>	<b>294</b>

<b>Topic</b>	<b>Page No.</b>
<b>9.5.3 Operational Characteristics of the Proposed Integrated System</b>	<b>295</b>
<b>9.5.4 Structure of the Secret Key for the Integrated System</b>	<b>296</b>
<b>9.5.4.1 Criteria for an Efficient Key Generation</b>	<b>296</b>
<b>9.5.4.2 Formation of Different Segments in the 252-bit Key</b>	<b>296</b>
<b>9.5.4.3 The 252-bit Secret Key</b>	<b>299</b>
<b>9.6 Conclusion</b>	<b>300</b>

## Chapter 10: A Conclusive Discussion

<b>Topic</b>	<b>Page No.</b>
<b>10.1 Introduction</b>	<b>303</b>
<b>10.2 A Comparison among Different Implementations</b>	<b>303</b>
<b>10.3 Conclusion on Different Model Implementations</b>	<b>310</b>

<b>Appendix</b>	<b>Topic</b>	<b>Page No.</b>
<b>A</b>	<b>References</b>	<b>i</b>
<b>B</b>	<b>List of Publications</b>	<b>vii</b>
<b>C</b>	<b>Listing of Source Codes</b>	<b>x</b>
<b>D</b>	<b>Bibliography</b>	<b>lxxiv</b>



## **Introduction**

<b><u>Contents</u></b>	<b><u>Page</u></b>
<b>1.1 Introduction</b>	<b>3</b>
<b>1.2 Cryptosystems</b>	<b>4</b>
<b>1.3 Evolution in the Field of Cryptography</b>	<b>25</b>
<b>1.4 Some of the Existing Techniques</b>	<b>28</b>
<b>1.5 An Overview of Proposed Techniques</b>	<b>34</b>
<b>1.6 A Note on Merits of Proposed Techniques</b>	<b>48</b>

## 1.1 Introduction

The requirements of information security within an organization have undergone a major change in last two decades. Before the widespread use of data processing equipment, the security of information used to be provided primarily by physical and administrative means. With the introduction of computers, the need of automated tools for protecting files and other information stored in the computer became evident. This is especially the case for a shared system, such as a time-sharing system, and the need is even more acute for systems that can be accessed over a public telephone network, data network, or the Internet [1, 2].

With the remarkable advancement of technology and availability of facilities, the interest of people to connect computers to form networks is increasing rapidly. In this digital system, a huge amount of data flow exists. Besides this, different distributed processing involves a large amount of complex digital data transfer [4].

In general, there exist following types of problems associated with such data transmission [5].

- A huge amount of data is to be handled.
- Much of the data is very sensitive to errors.
- The security of data transmitted from source to destination over communication links via different nodes is the most important matter to be worried.

Message can be intercepted by someone during the process of transmission that may cause problem. Hence data security and communication privacy have become a fundamental requirement for such systems [1, 2, 5].

Encoding a message prior to its transmission is the process of **Data Encryption**. The corresponding **Data Decryption** technique is used to decode the encrypted message. All the research activities of this researcher for the last few years were based on the field of cryptography, involving the planning, developing, designing and analyzing of some bit-level encryption/decryption techniques. Representation of this entire activity is the basic objective of this dissertation [1, 2, 45].

Section 1.2 of this chapter discusses cryptosystem with its different aspects. The historic background of the field of cryptography is represented in section 1.3. A brief

description of some of the current popular cryptographic techniques is represented in section 1.4. Section 1.5 represents an overview of all the proposed techniques created during this entire research activity. Section 1.6 points out merits of these proposed techniques followed by the concluding remark.

## 1.2 Cryptosystems

The fundamental objective of cryptography is to enable the transmission of message from one source point to the corresponding destination point over a transmission media in such a manner that the message during its transmission cannot be intercepted by someone [1, 11, 18].

An original message is termed as the **plaintext**. The coded message that is to be transmitted is referred to as the **ciphertext**. The process of converting from the plaintext to the ciphertext is known as **enciphering** or **encryption**. Restoring the plaintext from the ciphertext is termed as **deciphering** or **decryption**. The specialized area of study involving many schemes used for enciphering is known as **cryptography**. Such a scheme is known as **cryptographic system** or **cryptosystem** or **cipher**. Techniques used for deciphering a message without any knowledge of enciphering details constitute the area of **cryptanalysis**. The areas of cryptography and cryptanalysis together are called **cryptology** [2, 19, 20].

The concept of a cryptosystem can be formally defined by the mathematical notation presented in section 1.2.1.

### 1.2.1 Definition of Cryptosystem

A cryptosystem is a 5-tuple  $(\Pi, X, K, E, \Delta)$ , where the following conditions are satisfied [2, 4, 52]:

1.  $\Pi$  is a finite set of possible plaintexts.
2.  $X$  is a finite set of possible ciphertexts.
3.  $K$ , the keyspace, is a finite set of possible keys.
4. For each  $K \in K$ , there exists an encryption rule  $e_K \in E$  and a corresponding decryption rule  $d_K \in \Delta$ . Each  $e_K: \Pi \rightarrow X$  and  $d_K: X$

→  $\Pi$  are functions such that  $d_K(e_K(x)) = x$  for every plaintext  $x \in \Pi$ .

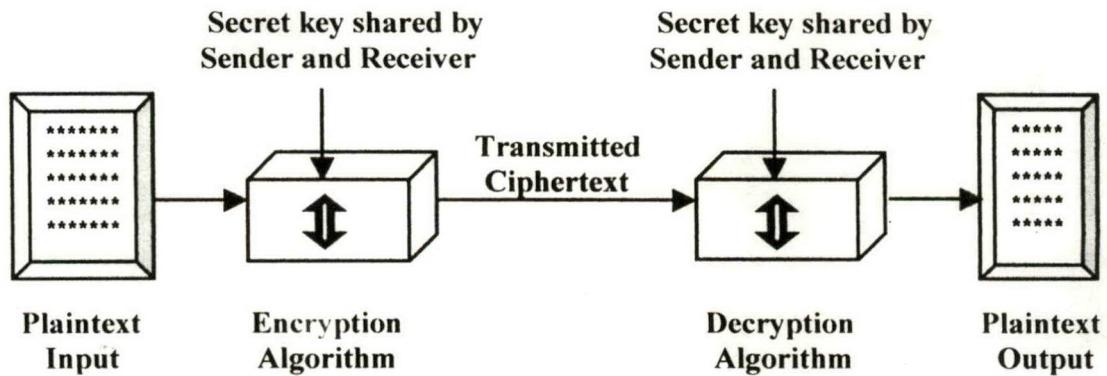
The main property is property 4. It says that if a plaintext  $x$  is encrypted using  $e_K$ , and the resulting ciphertext is subsequently decrypted using  $d_K$ , then the original plaintext  $x$  results.

Now, basically there are the two types of cryptosystems:

1. **Secret Key Cryptosystem**
2. **Public Key Cryptosystem**

Section 1.2.2 discusses the secret key cryptosystem with its different aspects and the different aspects of the public key cryptosystem are described in section 1.2.3.

### 1.2.2 Secret Key Cryptosystem



**Figure 1.2.2.1**  
**Simplified Model of Secret Key Cryptosystem**

The secret key cryptosystem has the following five ingredients [1, 2, 20]:

- **Plaintext:** This is the original intelligible message or data that is fed into the encryption algorithm as input.
- **Encryption algorithm:** The encryption algorithm performs various substitutions and transformations on the plaintext.
- **Secret key:** The secret is also input to the encryption algorithm. The key is a value independent to the plaintext. The algorithm will produce a different output depending on the specific key being

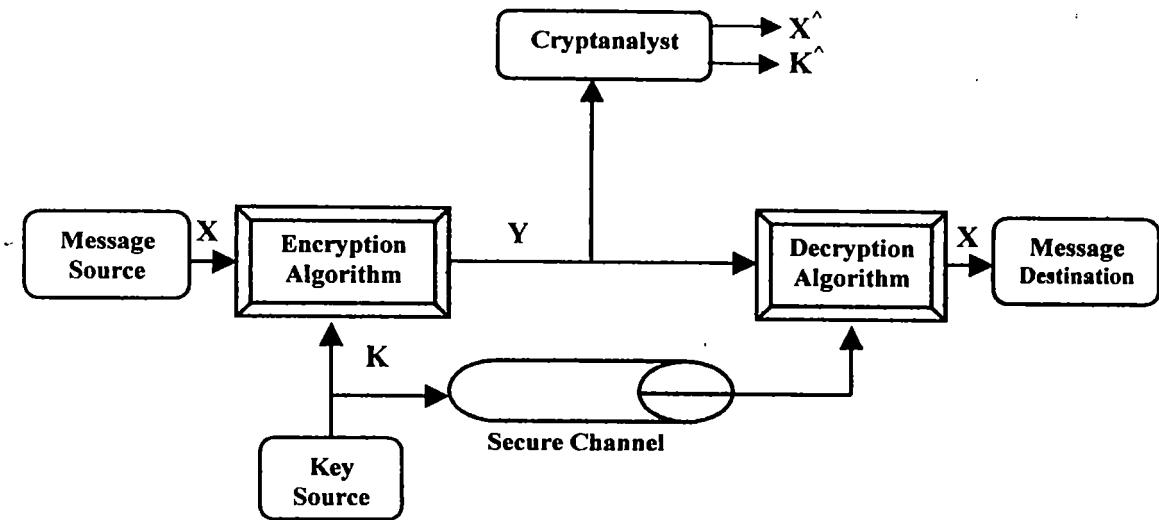
- used at the time. The exact substitution and transformation performed by the algorithm depend on the key.
- **Ciphertext:** This is the scrambled message produced as output. It depends on the plaintext and the secret key. For a given message, two different keys will produce two different ciphertexts. The ciphertext is an apparently random stream of data and it is unintelligible.
  - **Decryption algorithm:** This is essentially the encryption algorithm run in reverse. It takes the ciphertext and the secret key and produces the original plaintext.

#### 1.2.2.1 Requirements for Secret Key Cryptosystem

There are two requirements for the secured use of secret key cryptosystem [1]:

1. A strong encryption algorithm is needed. At a minimum, the algorithm should be such that an opponent who knows the algorithm and has access to one or more ciphertexts will be unable to decipher the ciphertext or figure out the key.
2. The sender and the receiver must have obtained copies of the secret key in a secure fashion and must keep the key secure. If someone can discover the key and knows the algorithm, all communication using this key will be readable.

In case of the secret key cryptosystem, it is assumed that it is impractical to decrypt a message on the basis of the ciphertext and the knowledge of the encryption/decryption algorithm. Therefore, generally, it is not needed to keep the algorithm secret. We need to keep only the key secret.



**Figure 1.2.2.1.1  
Model of Secret Key Cryptosystem**

Figure 1.2.2.1.1 shows the typical model of the secret key cryptosystem. A source produces a message in plaintext,  $X = [X_1, X_2, \dots, X_M]$ . The  $M$  elements of  $X$  are letters in some finite alphabet. It can be a stream of bits also. For the purpose of encryption, a key of the form  $K = [K_1, K_2, \dots, K_j]$  is generated. If the key is generated at the source point, then it must also be provided to the destination point by means of some secure channel. Alternatively, a third party can generate the key and securely deliver it to both source and destination [1, 2].

With the message  $X$  and the encryption key  $K$  as input, the encryption algorithm forms the ciphertext  $Y = [Y_1, Y_2, Y_3, \dots, Y_N]$ . We can write this as

$$Y = E_K(X)$$

This notation indicates that  $Y$  is produced by using the encryption algorithm  $E$  as a function of the plaintext  $X$ , with the specific function determined by the value of the key  $K$ .

The intended receiver, in possession of the key, is able to invert the transformation:

$$X = D_K(Y)$$

An opponent, observing  $Y$  but not having access to  $K$  or  $X$ , may attempt to recover  $X$  or  $K$  or the both  $X$  and  $K$ . It is assumed that the opponent knows the

encryption algorithm (E) and the decryption algorithm (D). If the opponent is interested in only this particular message, then the focus of the effort is to recover X by generating a plaintext estimate  $X^{\wedge}$  [1, 2, 4].

### 1.2.2.2 Evaluating a Cryptographic System

A cryptographic system is generally characterized by the following three independent dimensions [1]:

1. **The type of operations used for transforming plaintext into ciphertext:** All encryption algorithms are based on two general principles. One is **substitution**, in which each element in the plaintext (bit, letter, group of bits, or group of letters) is mapped into another element; and another is **transposition**, in which elements in the plaintext are rearranged. The fundamental requirement is that no information be lost, which means that all operations are to be reversible. Most of the cryptographic systems involve multiple stages of substitutions and transpositions.
2. **The number of keys used:** If both the sender and the receiver use the same key, the system is referred to as the **symmetric encryption**, or the **secret key encryption**, or the **conventional encryption** or the **classical encryption**. If the sender and the receiver use different keys, the system is referred to as the **asymmetric encryption**, or the **two-key encryption**, or the **public key encryption**.
3. **The way in which the plaintext is processed:** The **block cipher** processes the input one block of elements at a time, producing an output block for each input block. A **stream cipher** processes the input elements continuously, producing output one element at a time.

#### **1.2.2.3 Attacking a Conventional Encryption Scheme**

There are two general approaches to attack a conventional encryption scheme [1].

1. **Cryptanalysis:** Cryptanalytic attacks rely on the nature of the algorithm plus some knowledge of the general characteristics of the plaintext or even some sample plaintext-ciphertext pairs. This type of attack exploits the characteristics of the algorithm to attempt to deduce a specific plaintext or to deduce the key being used. If the attack succeeds in deducing the key, the effect is catastrophic: All future and past messages encrypted with that key are compromised [9, 10, 23].
2. **Brute-force Attack:** The attacker tries every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained.

For cryptanalytic attacks, the most difficult problem arises when all that is available is the ciphertext only. In some cases, not even the encryption algorithm is known. But, in general, we can assume that the opponent does know the encryption algorithm.

One possible attack under these circumstances is the brute-force approach of trying all possible keys. If the key space is very large, this becomes impractical. Thus the opponent must rely on an analysis of the ciphertext itself, generally applying various statistical tests to it. To use this approach, the opponent must have some general idea of the type of plaintext that is concealed [1, 9, 21, 22].

#### **1.2.2.4 The Condition for an Encryption Scheme to be Unconditionally Secure**

An encryption scheme is **unconditionally secure** if the ciphertext generated by the scheme does not contain enough information to determine uniquely the corresponding plaintext, no matter how much ciphertext is available. This means, no matter how much time an opponent has, it is impossible to the opponent to decrypt the ciphertext, simply because of the reason that there is no required information [1].

Except the encryption scheme, known as **one-time pad**, there is no encryption that is unconditionally secure.

In the research field of cryptography, there has been a continuous trend of developing encryption algorithms that are **computationally secure**.

#### 1.2.2.5 The Condition for an Encryption Scheme to be Computationally Secure

An encryption algorithm is said to be **computationally secure** if the following two criteria are met [1, 24, 39]:

- The cost of breaking the cipher exceeds the value of the encrypted information.
- The time required to break the cipher exceeds the useful lifetime of the information.

**Table 1.2.2.5.1**  
**Average Time Required for Exhaustive Key Search**

Key Size (Bits)	Number of Alternative Keys	Time Required at 1 Encryption / $\mu s$	Time Required at $10^6$ Encryptions / $\mu s$
56	$2^{56} = 7.2 \times 10^{16}$	$2^{55} \mu s = 1142 \text{ years}$	$10.01 \text{ hours}$
128	$2^{128} = 3.4 \times 10^{38}$	$2^{127} \mu s = 5.4 \times 10^{24} \text{ years}$	$5.4 \times 10^{18} \text{ years}$
168	$2^{168} = 3.7 \times 10^{50}$	$2^{167} \mu s = 5.9 \times 10^{36} \text{ years}$	$5.9 \times 10^{30} \text{ years}$
26 characters (Permutation)	$26! = 4 \times 10^{26}$	$2 \times 10^{26} \mu s = 6.4 \times 10^{12} \text{ years}$	$6.4 \times 10^6 \text{ years}$

If we consider the time required to use a brute-force approach, which simply involves trying every possible key until an intelligible translation of the ciphertext into plaintext is obtained. On the average, half of all possible keys must be tried to achieve success.

Table 1.2.2.5.1 shows how much time is involved for various key spaces used in some secret key encryption schemes, which will be discussed in brief in section 1.4. Four sample results have been shown in the table [1].

- The 56-bit key size is used with the DES (Data Encryption Standard) algorithm [13, 14, 15].
- The 128-bit key size is used with the AES (Advanced Encryption Standard) algorithm [1].

- The 168-bit key size is used with triple DES [13, 14, 15].

One result is also shown in table 1.2.2.5.1 for the substitution code that uses a 26-character key.

Now, it is assumed that 1  $\mu$ s is required for a single decryption, which is a reasonable order of magnitude for today's machines. With the use of massively parallel organization of microprocessors, it may be possible to achieve processing rates many orders of magnitude greater. The final column of table 1.2.2.5.1 considers the results for a system that can process 1 million keys per microsecond ( $10^6$  keys /  $\mu$ s). It is observed from the table that at this performance level, DES can no longer be considered computationally secure.

### **1.2.2.6 Basic Building Blocks in Classical Encryption Techniques**

In this section, a sampling has been considered what might be called classical encryption techniques. A study and presentation of these techniques enables this researcher to illustrate the basic approaches used in all the proposed techniques [1, 2, 24].

The two basic building blocks of all encryption techniques are:

- 1. Substitution Techniques**
- 2. Transposition Techniques**

Section 1.2.2.6.1 discusses on principles of substitution techniques with examples and section 1.2.2.6.2 discusses on principles of transposition techniques, also with examples. An application by combining substitution and transposition techniques has been pointed out in section 1.2.2.6.3.

#### **1.2.2.6.1 Principles of Substitution Techniques**

If the plaintext is viewed as a sequence of English alphabets, the substitution technique is one in which the letters of a plaintext are replaced by other letters [2, 4].

In this section, some classical substitution techniques have been discussed from different perspectives clearly indicating the evolution in minimizing the chance of breaking ciphers. These include the following techniques:

- Caesar Cipher
- Monoalphabetic Substitution Cipher

- Homophonic Substitution Cipher
- Playfair Cipher
- Polyalphabetic Cipher (Vigenere Cipher)
- One-time Pad

#### 1.2.2.6.1.1 Caesar Cipher

The earliest and the simplest use of the substitution cipher was **Caesar Cipher**, in which each letter of the alphabet in the plaintext is replaced in the ciphertext with the letter standing three places further down the alphabet. The alphabet is wrapped around, so that the letter following Z is A [1]. We can define the transformation by listing all possibilities, as follows:

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

The algorithm for Caesar Cipher can be expressed as follows:

$$C = E(p) = (p+3) \bmod (26)$$

Here p represents the numerical equivalent of a plaintext letter (0 is the numerical equivalent of a, 1 is the numerical equivalent of b, and so on) and C represents the numerical equivalent of a ciphertext letter.

Making the shift of any amount, we get the **General Caesar Algorithm** simply as follows:

$$C = E(p) = (p+k) \bmod (26)$$

Here k can be anything in the range of 1 to 25.

For this general Caesar algorithm, the decryption algorithm is simply as:

$$p = D(C) = (C-k) \bmod (26)$$

If it is known that a given ciphertext is a Caesar cipher, then a brute-force cryptanalysis is easily performed because there are only 25 keys to try.

#### **1.2.2.6.1.2 Monoalphabetic Substitution Cipher**

From Caesar Cipher, a dramatic increase in the key space can be achieved by allowing an arbitrary substitution. This approach results in having as many as  $26! = 4 \times 10^{26}$  keys, as shown in table 1.2.2.5.1. Such an approach is referred to as a **monoalphabetic substitution cipher** [1, 2].

But monoalphabetic ciphers are also easy to break, because they reflect the frequency data of the original alphabet. An enhancement is done using the **Homophonic Substitution Cipher** [4].

#### **1.2.2.6.1.3 Homophonic Substitution Cipher**

In homophonic substitution cipher, for a single letter multiple substitutes are provided. For example, the letter “e” may be assigned a number of different cipher symbols, such as 16, 74, 35, and 21, with each homophone used in rotation. If the number of symbols assigned to each letter is proportional to the relative frequency of that letter, then single-letter frequency information is completely obliterated [4].

However, even with homophones, each element of plaintext affects only one element of ciphertext, making cryptanalysis relatively straightforward.

#### **1.2.2.6.1.4 Playfair Cipher**

The Playfair algorithm is based on the use of a  $5 \times 5$  matrix of letters constructed using a keyword [1]. Table 1.2.2.6.1.4.1 gives an example.

**Table 1.2.1.6.1.4.1  
An Example of Playfair Cipher**

M	O	N	A	R
C	H	Y	B	D
E	F	G	I/J	K
L	P	Q	S	T
U	V	W	X	Z

In this example, the keyword is MONARCHY.

The rules for filling the matrix are as follows:

- The letters of the keyword (minus duplicates) are to be placed from left to right and from top to bottom of the matrix.
- The remainder part of the matrix is to be filled in with the remaining letters in alphabetical order. Any one of the letters I and J is to be used, not the both.

The plaintext is to be encrypted two letters at a time, according to the following set of rules:

- Repeating plaintext letters that would fall in the same pair are separated with a filler letter, such as x, so that “balloon” would be treated as “ba lx lo on”.
- Plaintext letters that fall in the same row of the matrix are each replaced by the letter to the right, with the first element of the row circularly following the last. For example, “ar” is encrypted as “RM”.
- Plaintext letters that fall in the same column are each replaced by the letter beneath, with the top element of the row circularly following the last. For example, “mu” is encrypted as “CM”.
- Otherwise, each plaintext letter is replaced by the letter that lies in its own row and the column occupied by the other plaintext letter in the pair. For example, “hs” becomes “BP”, “ea” becomes “IM” or “JM”.

The Playfair cipher is a great advance over simple monoalphabetic ciphers.

#### **1.2.2.6.1.5 Polyalphabetic Cipher**

The basic principle of the polyalphabetic cipher is to use different monoalphabetic substitutions as one proceeds through the plaintext message. All the polyalphabetic substitution cipher techniques have the following two things in common [1]:

- A set of related monoalphabetic substitution rules is used.
- A key determines which particular rule is chosen for a given transformation.

The best known and one of the simplest of such algorithms is referred to as **Vigenere Cipher**.

#### 1.2.2.6.1.6    **Vigenere Cipher**

To understand the concept of this Vigenere cipher, table 1.2.2.6.1.6.1 is constructed that is known as Vigenere table [1, 2, 4].

Using this table, to encrypt a message, a key is needed, which is a repeating keyword. For example, if the keyword is “deceptive”, the message “we are discovered save yourself” is encrypted as is shown in figure 1.2.2.6.1.6.1.

**Table 1.2.2.6.1.6.1**  
**Vigenere Table**

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	
a	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
b	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
c	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
d	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
e	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
f	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
g	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
h	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	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
j	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
k	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
l	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
m	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
n	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
o	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
p	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
q	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
r	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
s	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
t	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
u	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
v	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
w	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
x	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
y	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
z	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

Key																										
Plaintext																										
Ciphertext																										
z	i	c	v	t	w	q	n	g	r	z	g	v	t	w	a	v	z	h	c	q	y	g	l	m	g	j

**Figure 1.2.2.6.1.6.1**  
**Example of Vigenere Cipher**

The process of encryption is very simple: Given a key letter  $x$  and a plaintext letter  $y$ , the ciphertext letter is at the intersection of the row labeled  $x$  and the column labeled  $y$ ; in this case the ciphertext letter is V.

For the purpose of encryption, a key is needed that is long as the message. Usually the key is a repeating keyword, as is shown in figure 1.2.1.1.1.

Decryption also is usually simple. The key letter identifies the row. The position of the ciphertext 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 ciphertext letters for each plaintext letter, one for each unique letter of the keyword. Thus the letter frequency information is obscured.

However, even this scheme is vulnerable to cryptanalysis. Since the key and the plaintext share the same frequency distribution of letters, a statistical technique can be applied.

#### 1.2.2.6.1.7 Enhancement from Vigenere Cipher

The scheme, As per the pattern, the same as Vigenere cipher, known as the one-time pad, which uses a random key, is unbreakable. It produces random output that bears no statistical relationship to the plaintext. The ciphertext contains no information whatsoever about the plaintext, so that there is simply no way to break the code [1, 2, 4].

The only problem associated with the one-time pad technique is the length and the randomness of the key used. 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.

#### **1.2.2.6.1.8 Some Other Substitution Ciphers**

Besides all that have been discussed, there are few more classical substitution ciphers including [2]:

- **Hill Cipher**
- **Affine Cipher**

#### **1.2.2.6.2 Principles of Transposition Techniques**

The basic philosophy of the transposition cipher is to keep the plaintext characters unchanged, but to alter their positions by rearranging them.

In this section, one classical transposition cipher systems has been discussed briefly to illustrate the concept.

##### **1.2.2.6.2.1 The Rail Fence Technique**

In this approach, the plaintext is written down as a sequence of diagonals and then read off as a sequence of rows [1].

For example, we consider the following plaintext:

###### **The rail fence technique**

If it is encrypted with a rail fence of depth 2, we write the following:

t	e	a	l	e	c	t	c	n	q	e
h	r	i	f	n	e	e	h	i	u	

The resulting encrypted message is:

**TEALECTCNQEHRIFNEEHIU**

The rail fence technique is trivial to cryptanalyze. A more complex scheme is to write the message in a rectangle, row by row, and read the message off, column by column, but permute the order of the columns. The order of the column then becomes the key of the algorithm.

Following this approach, the following plaintext is considered:

**attack postponed until two am**

The matrix is constructed as follows, with specifying the key:

<b>Key:</b>	4	2	3	5	1
<b>Plaintext:</b>	a	t	t	a	c
	k	p	o	s	t
	p	o	n	e	d
	u	n	t	i	l
	t	w	o	a	m

**Ciphertext:** **CTDLMTPONWTONTTOAKPUTASEIA**

A pure transposition cipher is easily recognized because it has the same letter frequencies as the original plaintext.

#### **1.2.2.6.2.1.1 A Cascaded Approach**

By performing more than one stage of transposition, this type of transposition technique can be made significantly more secured [1, 36].

If the same technique is applied on the ciphertext of the previous stage, we get the following structure:

<b>Key:</b>	4	2	3	5	1
<b>Plaintext:</b>	c	t	d	l	m
	t	p	o	n	w
	t	o	n	t	o
	a	k	p	u	t
	a	s	e	i	a

**Ciphertext:** **MWOTATPOKSDONPECTTAALNTUI**

With having 25 letters in the source message, the original sequence of letters is:

01	02	03	04	05	06	07	08	09	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24
25											

After the first transposition, the following sequence is obtained:

05	10	15	20	25	02	07	12	17	22	03	08
13	18	23	01	06	11	16	21	04	09	14	19
24											

It has a somewhat regular structure. But after the second transposition, the sequence obtained is as follows:

25	22	23	21	24	10	07	08	06	09	15	12
13	11	14	05	02	03	01	04	20	17	18	16
19											

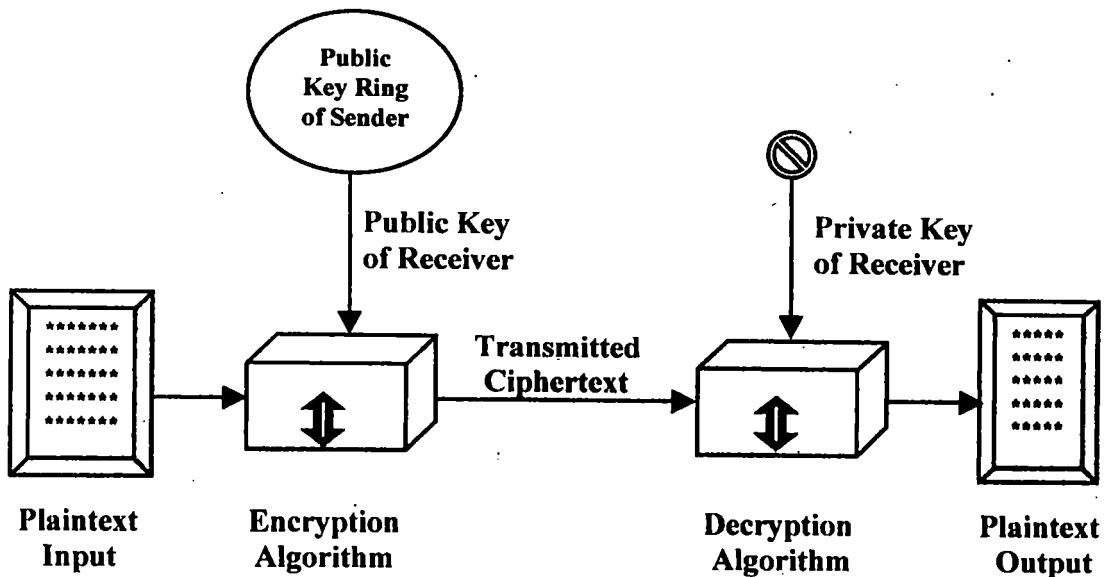
This is a much less structured permutation and is much more difficult to cryptanalyze.

#### 1.2.2.6.3 Rotor Machines – Composite Substitution and Transposition Cipher

Before the introduction of Data Encryption Standard (DES), the most important application of the principle of multiple stages of encryption was a class of systems known as the rotor machines [1].

A rotor machine has a keyboard and series of rotors and implements a version of the Vigenere cipher. Each rotor is an arbitrary permutation of the alphabet, has 26 positions and performs a simple substitution. 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. Because the rotors all move at different rates, the period for n-rotor machine is  $26^n$ . It gives a frustrating picture to the cryptanalysts.

### 1.2.3 Public Key Cryptosystem



**Figure 1.2.3.1**  
**Typical Model of Public Key Cryptosystem**

A public key encryption scheme has the following six ingredients [1, 2, 7, 8]:

- **Plaintext:** This is the readable message or data that is fed into the algorithm as input.
- **Encryption algorithm:** The encryption algorithm performs various transformations on the plaintext.
- **Public and private key:** This is a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the encryption algorithm depend on the public or private key that is provided as input.
- **Ciphertext:** This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different messages will produce two different ciphertexts.
- **Decryption algorithm:** This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

### **1.2.3.1 The Algorithm**

The following essential steps are to be followed in this regard [1, 2]:

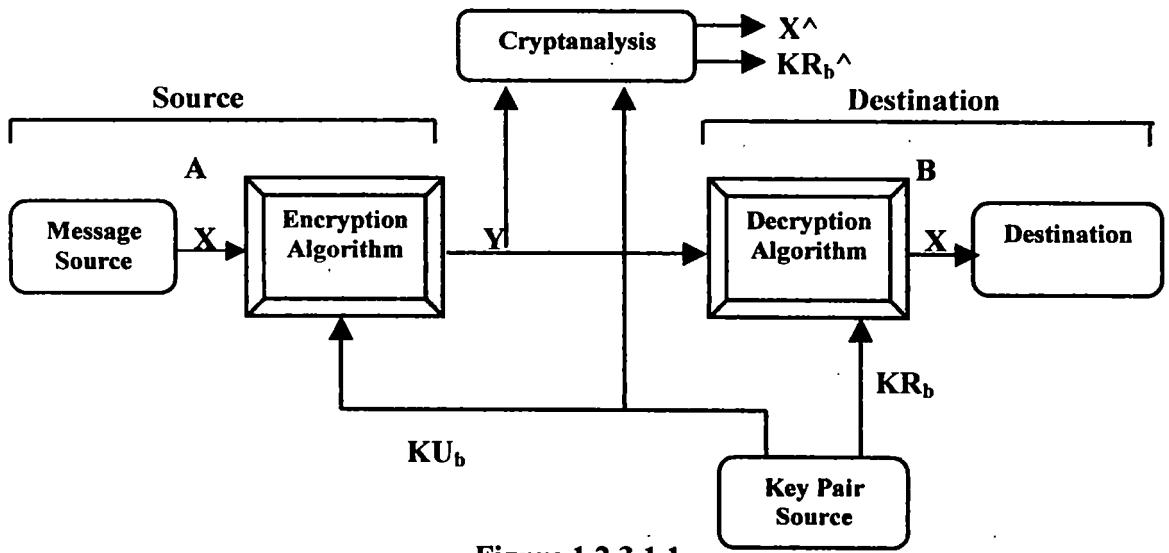
1. Each user generates a pair of keys to be used for the encryption and the decryption of messages.
2. Each user places one of the two keys in a public register or any other accessible file. This is the public key. The companion key is kept private. As given in figure 1.2.2.1, each user maintains a collection of public keys from others.
3. If the sender wants to send a confidential message to the receiver, he encrypts the message using the public key.
4. When the receiver receives the message, he decrypts it using his private key. No other recipient can decrypt the message because only the receiver known his private key.

With this approach, all participants have access to public keys, and private keys are generated locally by each participant and therefore need never be distributed. As long as a system, controls its private key, its incoming communication is secured. At any time, a system can exchange its private key and publish the companion public key to replace its old public key.

Table 1.2.3.1.1 summarizes some of the important aspects of symmetric and public key encryption [1].

**Table 1.2.3.1.1**  
**Conventional and Public Key Encryption**

<b>Conventional Encryption</b>	<b>Public Key Encryption</b>
<b><u>For Functional Purpose</u></b>	<b><u>For Functional Purpose</u></b>
<p>1. The same algorithm with the same key is used for encryption and decryption.</p> <p>2. The sender and the receiver must share the algorithm and the key.</p>	<p>1. One algorithm is used for encryption and decryption with a pair of keys, one for encryption and one for decryption.</p> <p>2. The sender and the receiver must each have one of the matched pair of keys (not the same one)</p>
<b><u>For Security Purpose</u></b>	<b><u>For Security Purpose</u></b>
<p>1. The key must be kept secret.</p> <p>2. It must be impossible or at least impractical to decipher a message if no other information is available.</p> <p>3. Knowledge of the algorithm plus samples of ciphertext must be insufficient to determine the key.</p>	<p>1. One of the two keys must be kept secret.</p> <p>2. It must be impossible or at least impractical to decipher a message if no other information is available.</p> <p>3. Knowledge of the algorithm plus one of the keys plus samples of ciphertext must be insufficient to determine the other key.</p>



**Figure 1.2.3.1.1  
Public Key Cryptosystem**

A closer look at the essential elements of a public key encryption scheme is shown in figure 1.2.3.1.1. There is a source A that produces a message in plaintext, say,  $X=[X_1, X_2, \dots, X_M]$ . The  $m$  elements of  $X$  are letters in some finite alphabet. The message is intended for destination B. B generates a related pair of keys: a public key,  $KU_b$ , and a private key,  $KR_b$ .  $KR_b$  is known only to B, whereas  $KU_b$  is publicly available and therefore accessible by A [2, 4, 24].

With the message  $X$  and the encryption key  $KU_b$  as input, A forms the ciphertext  $Y=[Y_1, Y_2, \dots, Y_N]$ . The intended receiver, in possession of the matching private key, is able to invert the transformation.

An opponent, observing  $Y$  and having access to  $KU_b$ , but not having access to  $KR_b$  or  $X$ , must attempt to recover  $X$  and/or  $KR_b$ . It is assumed that the opponent does have the knowledge of the encryption (E) and the decryption (D) algorithms. If the opponent is interested only in this particular message, then the focus of effort is to recover  $X$ , by generating a plaintext estimate  $X^{\wedge}$ .

Often, however, the opponent is interested in being able to read future messages as well, in which case an attempt is made to recover  $KR_b$  by generating an estimate  $KR_b^{\wedge}$ .

### **1.2.3.2 Characteristics of Public Key Cryptography**

Following are the requirements to be satisfied for the public key cryptography [1, 7, 8]:

- It should be computationally easy for a party B to generate a pair (public key  $KU_b$  and private key  $KR_b$ ).
- It should be computationally easy for a sender A, knowing the public key and the message to be encrypted, M, to generate the corresponding ciphertext C.
- It should be computationally easy for the receiver B to decrypt the resulting ciphertext using the private key to recover the original message.
- It should be computationally infeasible for an opponent, knowing the public key,  $KU_b$ , to determine the private key,  $KR_b$ .
- It should be computationally infeasible for an opponent, knowing the public key,  $KU_b$ , and a ciphertext C, to recover the original message, M.

One more point can be added in this regard, which, although useful, is not necessary for all public key applications:

- The encryption and the decryption functions can be applied in either order.

## **1.3 Evolution in the Field of Cryptography**

The field of cryptography has a curious history. Until the First World War, important developments in this field appeared in a more or less timely fashion and the field moved forward in much the same way like other specialized disciplines [1].

In 1920, one of the most influential cryptanalytic papers of twentieth century, William F. Friedman's monograph "**The index of Coincidence and its applications in cryptography**", appeared as a first research report of the private Riverbank Laboratories [2].

In the same year, Edward H. Hebern of Oakland, California, filed the first patent for a rotor machine, the device destined to be a mainstay of military cryptography for nearly 50 years [2, 4].

After the First World War, things began to change. U.S. army and Navy organizations, working entirely in this field, began to make fundamental advances in cryptography [1, 24].

During the thirties and forties, a few basic papers did appear in the open literature and several treatises on the subject were published, but the later were farther and farther behind the state of art [1].

By the end of the war, the transition was complete. With one notable exception, the public literature had died. The exception was Claude Shanon's paper, "**The Communication Theory of Secrecy Systems**", which appeared in the Bell System Technical Journal in 1949 [1, 2].

During the period of 1949 to 1967, the cryptographic literature was fruitless. During this period, a different sort of contribution appeared, which was David Kahn's history, "**The Codebreakers**". It did not contain any new technical ideas, but it did contain a remarkably complete history of what had gone before, including mention of something that the Govt. of USA still considers secret. The significance of the Codebreakers lay not just its remarkable scope, but also in the fact that it enjoyed good sales and made tens of thousands of people aware of cryptography, who had never given the matter a moment's thought [1, 2, 4, 5].

At about the same time, Horst Fiestel, who had earlier worked on identification of friend or foe devices for the Air Force, took his life long passion for cryptography to the IBM Watson Laboratory in Yorktown Heights, New York. There, he began development of what was to become the US Data Encryption Standard. By early 1970s, several technical reports on this subject by Fiestel and his colleagues had been made public by IBM [1, 5].

This was the situation when W. Diffie entered the field in late 1972 [1].

In 1975, Hellman and W. Diffie proposed public Cryptography. One of the indirect aspects of their contribution was to introduce a problem that does not even

appear easy to solve. This enthused a number of people in Cryptography, the number of meetings held, and number of books and papers published [1, 2].

W. Diffie told the following statements to the audience in his acceptance speech for the Donald E. Fink award – given for the best expository paper, “**Privacy and Authentication: An Introduction to Cryptography**”, to appear in an IEEE Journal – which he received jointly with Hellman in 1980 [1]:

*“I had an experience that I suspected was rare even among the prominent scholars who populate the IEEE award Ceremony: I had written the paper I had wanted to study, but could not find, when I first became seriously interested in Cryptography. Had I been able to go to the Stanford Bookstore and picked up a modern cryptography text, I would probably have learned about the fields years earlier. But only things available in the fall of 1972 were a few classic papers and some obscure technical reports.”*

The contemporary researcher has no such problem. The problem now is choosing where to start among the thousands of papers and dozens of books. It has been necessary to spend long hours hunting out and then studying the research literature before being able to design the sort of cryptographic utilities glibly described in popular articles [1].

This gap has been filled by Bruce Schneier’s “**Applied Cryptography**”. Bruce Schneier has given a panoramic view of the fruits of 20 years (1976 – 1996) of public research. He has included an account of the world in which Cryptography is developed and applied, and discusses entities ranging from the International Association for Cryptologic Research (IACR) to the National Security Agency (NSA) [2].

In early 80s, NSA made several attempts to control Cryptography and issued a letter to the IEEE mentioning that the publication of Cryptographic material is a violation of the International Traffic Arms Regulations (ITAR). This viewpoint turned out not even to be supported by the regulation of them – which contained an explicit exemption for published material [5].

## **1.4 Some of the Existing Techniques**

While discussing cryptosystem and its different aspects in section 1.2, some classical encryption techniques have already been discussed. These include the following techniques [1]:

- Caesar Cipher
- Monoalphabetic Substitution Cipher
- Homophonic Substitution Cipher
- Playfair Cipher
- Polyalphabetic Cipher (Vigenere Cipher)
- One-time Pad
- Rail-Fence Technique
- Rotor Machines

In this section, without going to the detailed discussion, the special characteristics of **The Data Encryption Standard (DES)**, the most widely used cryptosystem in the world, and **The RSA Technique**, the most successful public key cryptosystem, have been pointed out in section 1.4.1 and section 1.4.2 respectively.

### **1.4.1 The Data Encryption Standard (DES)**

The DES was adopted in 1977 by the then National Bureau of Standards, now the National Institute of Standards and Technology (NIST), as Federal Information Processing Standard 46 (FIPS PUB 46) [[16, 17].

#### **1.4.1.1 Basic Principles of DES**

1. It is a symmetric block cipher.
2. Data are encrypted in 64-bit blocks using a 56-bit key.
3. The algorithm transforms 64-bit input in a series of steps into a 64-bit output. The same steps with the same key are used to reverse the encryption [25].

### 1.4.1.2 The Basic Algorithm

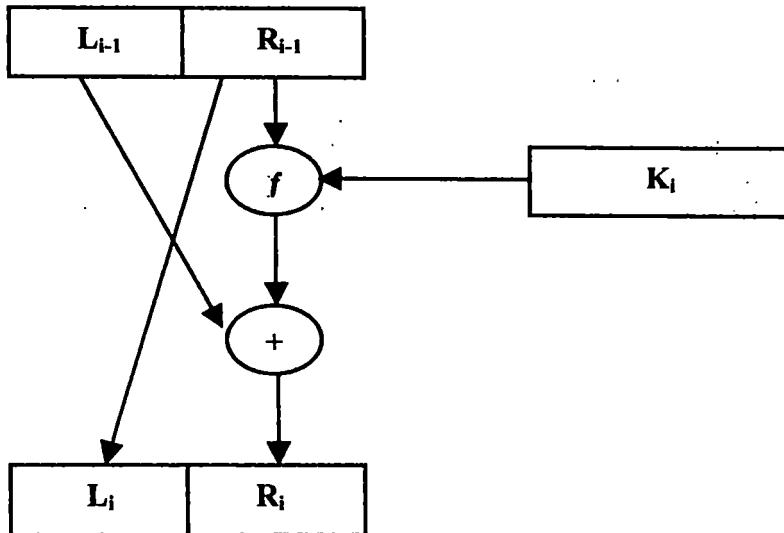
The algorithm proceeds in the following three stages [13, 27, 28]:

1. Given a plaintext  $x$ , a bitstring  $x_0$  is constructed by permuting the bits of  $x$  according to a fixed initial permutation IP. It can be represented as  $x_0 = IP(x) = L_0R_0$ , where  $L_0$  comprises the first 32 bits of  $x_0$  and  $R_0$  comprises the last 32 bits.
2. 16 iterations of a certain function are then computed. For  $i \leq 16$ ,  $L_iR_i$  is computed as follows:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

Here  $\oplus$  denotes the XOR operation of two bitstrings,  $f$  is a function, and  $K_1, K_2, \dots, K_{16}$  are each bitstring of length 48 computed as a function of the key  $K$ . In fact, each  $K_i$  is a permuted selection of bits from  $K$ .  $K_1, K_2, \dots, K_{16}$  comprises the key schedule. One round of encryption is depicted in figure 1.4.1.2.1.



**Figure 1.4.1.2.1**  
**One Round of DES Encryption**

3. Apply the inverse permutation  $IP^{-1}$  to the bitstring  $R_{16}L_{16}$ , obtaining the ciphertext  $y$ . That is,  $y = IP^{-1}(R_{16}L_{16})$ .

#### 1.4.1.2.1 The Function $f$

The function  $f$  takes as input a first argument  $A$ , which is a bitstring of length 32, a second argument  $J$  that is a bitstring of length 48, and produces as output a bitstring of length 32. The following steps are executed [33, 35]:

1. The first argument  $A$  is “expanded” to a bitstring of length 48 according to a fixed expansion function  $E$ .  $E(A)$  consists of 32 bits from  $A$ , permuted in a certain way, with 16 of the bits appearing twice.
2.  $E(A) \oplus J$  is computed and the result is to be written as the concatenation of eight 6-bit strings  $B = B_1B_2B_3B_4B_5B_6B_7B_8$ .
3. The next step uses eight S-boxes  $S_1, S_2, \dots, S_8$ . Each  $S_i$  is a fixed  $4 \times 16$  array whose entries come from the integers 0 – 15.

Given a bitstring of length six, say,  $B_j = b_1b_2b_3b_4b_5b_6$ ;  $S_j(B_j)$  is to be computed as follows:

The two bits  $b_1b_6$  determine the binary representation of a row  $r$  of  $S_j$  ( $0 \leq r \leq 3$ ), and the four bits  $b_2b_3b_4b_5$  determine the binary representation of a column  $c$  of  $S_j$  ( $0 \leq c \leq 15$ ). Then  $S_j(B_j)$  is defined to be the entry  $S_j(r, c)$ , written in binary as a bitstring of length four.

Hence each  $S_j$  can be thought of as a function that accepts as input a bitstring of length two and one of length four, and produces as output a bitstring of length four.

In this fashion,  $C_j$  can be calculated as  $C_j = S_j(B_j)$ ,  $1 \leq j \leq 8$ .

4. The bitstring  $C = C_1C_2C_3C_4C_5C_6C_7C_8$  of 32 is permuted according to a fixed permutation  $P$ . The resulting bitstring  $P(C)$  is defined to  $f(A, J)$ .

#### 1.4.1.3 Applications

It can be implemented very easily, either in hardware or in software. One important application of DES is in banking transaction, using standards developed by the American Bankers Association. DES is used to encrypt personal identification numbers

(PINs) and account transactions carried out by automated teller machines (ATMs). DES is also used by the Clearing House Interbank Payments System (CHIPS) to authenticate transactions involving over  $\$1.5 \times 10^{12}$  per week [1, 16].

#### 1.4.1.3.1 Modes of Operations

There are the following four modes of operations that have been developed for DES [1, 17]:

1. **Electronic Codebook Mode (ECB)**
2. **Cipher Feedback Mode (CFB)**
3. **Cipher Block Chaining Mode (CBC)**
4. **Output Feedback Mode (OFB)**

A detailed discussion on the scheme for each of these modes is excluded from this section.

#### 1.4.1.4 The DES Controversy

When DES was proposed as a standard, there was a considerable criticism. One objection to DES concerned the S-boxes. All computations in DES, with the exception of the S-boxes, are linear. The S-boxes, being the non-linear component of the cryptosystem, are vital to its security. However, the design criteria of the S-boxes were not completely known until 1976, when the National Security Agency (NSA) asserted a set of properties as the design criteria of the S-boxes [13].

The most pertinent criticism of DES is that the size of the keyspace,  $2^{56}$ , is too small to be really secure [12].

### 1.4.2 The RSA Technique – The Special Characteristics

The first realization of a public key system came in 1977 by Rivest, Shamir, and Adleman, who invented the well-known RSA Cryptosystem [2, 8].

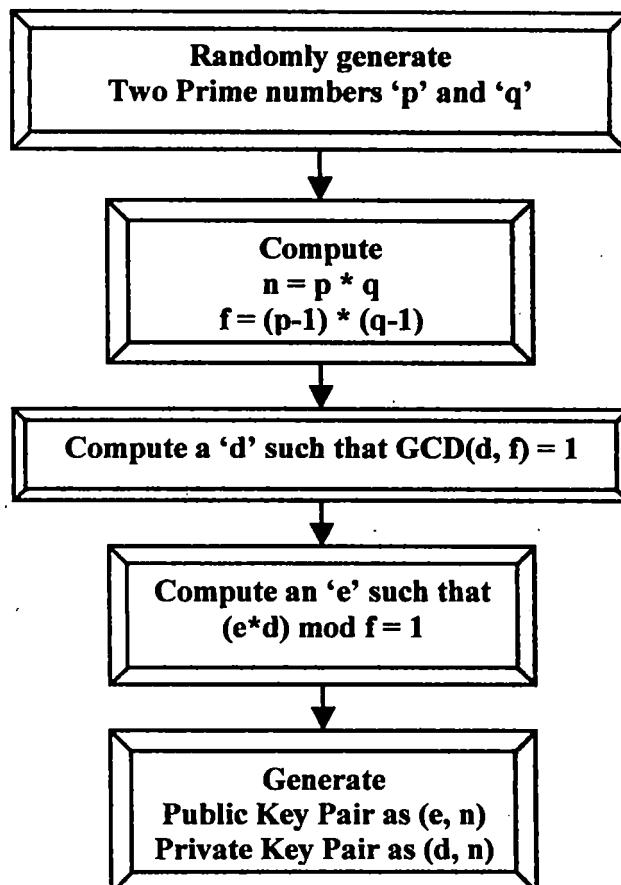
#### 1.4.2.1 The RSA Algorithm

To create an RSA public/private key pair, following are the basic steps [7]:

1. Choose two prime numbers, p and q. Calculate  $n = pq$ .

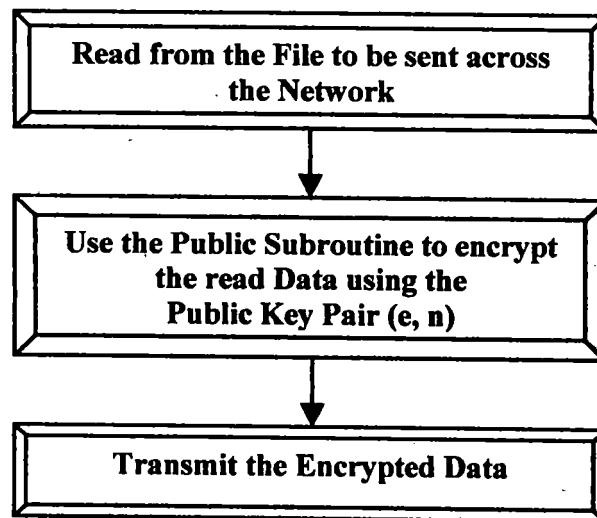
2. Select a third number,  $e$ , that is relatively prime to the product  $(p-1)(q-1)$ . The number  $e$  is the public exponent.
3. Calculate an integer  $d$  from the quotient  $(ed-1)/[(p-1)(q-1)]$ . The number  $d$  is the private exponent.
4. The public key is the number pair  $(n, e)$ . Although these values are publicly known, it is computationally infeasible to determine  $d$  from  $n$  and  $e$ , if  $p$  and  $q$  are large enough.
5. To encrypt a message,  $M$ , with the public key, create the ciphertext,  $C$ , using the equation:  $C = M^e \text{ mod } n$ .
6. The receiver then decrypts the ciphertext with the private key using the equation:  $M = C^d \text{ mod } n$ .

Figure 1.4.2.1.1 depicts the key generation by a pictorial representation.

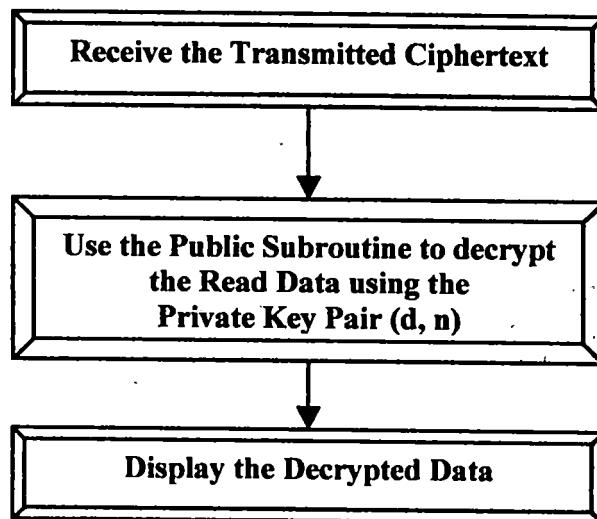


**Figure 1.4.2.1.1**  
**Key Generation in RSA**

Figure 1.4.2.1.2 and figure 1.4.2.1.3 respectively show the pictorial representations of the encryption and the decryption techniques in the RSA technique.



**Figure 1.4.2.1.2**  
**Encryption in RSA Technique**



**Figure 1.4.2.1.3**  
**Decryption in RSA Technique**

#### **1.4.2.2 The Security of RSA**

Possible approaches to defend against at least two possible attacks on the RSA algorithm are as follows [40, 41]:

1. **Brute Force:** The defense against the brute force approach is the same for RSA as for other cryptosystems, namely, use a large key space. Therefore it is better to take as larger as possible number of bits in e and d. However, because the calculations involved, both in key generation and in encryption/decryption, are complex, the larger the size of the key, the slower the system will run [1, 23].
2. **Mathematical Attacks:** To avoid values of n that may be factored more easily, the inventors of the algorithm suggest the following constraints on p and q [7]:

- Lengths of p and q should differ by only a few digits. For example, for a 1024-bit key (309 decimal digits), both p and q should be on the order of magnitude of  $10^{75}$  to  $10^{100}$ .
- Both (p-1) and (q-1) should contain a large prime factor.
- The greatest common divisor (gcd) of (p-1) and (q-1) should be small.

### **1.5 An Overview of Proposed Techniques**

This section provides an overall idea on all the techniques proposed in the thesis. Following are the proposed names of the set of independent techniques developed during the research work:

1. **Recursive Positional Substitution Based on Prime-Nonprime of Cluster (RPSP)**
2. **Triangular Encryption Technique (TE)**
3. **Recursive Paired Parity Operation (RPPO)**
4. **Recursive Positional Modulo-2 Substitution Technique (RPMS)**
5. **Recursive Substitutions of Bits Through Prime-Nonprime Detection of Sub-stream (RSBP)**

## **6. Recursive Substitutions of Bits Through Modulo-2 Detection of Sub-stream (RSBM)**

Before pointing out the overview of each technique separately, a combined overview of all the techniques has been summarized in table 1.5.1. The table points out schematic characteristics of the techniques in terms of the following attributes:

- Whether bit-level implementation or not
- Whether public key system or private key system
- Whether techniques of encryption and decryption exactly the same (symmetric) or not (asymmetric) or even there exists a combination of the both
- Whether block cipher or stream cipher
- Whether a substitution technique or a transposition technique
- Whether the basic operation is Boolean or Non-Boolean
- Whether there is data compression/expansion or no alteration in size
- Whether there is any formation of cycle or not

**Table 1.5.1**  
**Schematic Characteristics of Proposed Techniques**

	RPSP	TE	RPPO	RPMs	RSBP	RSBM
<b>Implementation in Bit-Level</b>	√	√	√	√	√	√
<b>Implementation not in Bit-Level</b>						
<b>Public Key System</b>						
<b>Private Key System</b>	√	√	√	√	√	√
<b>Symmetric</b>	√		√			
<b>Asymmetric</b>				√	√	√
<b>Both Symmetric and Asymmetric</b>		√				
<b>Block Cipher</b>	√	√	√	√	√	√
<b>Stream Cipher</b>						
<b>Substitution Technique</b>		√	√	√	√	√
<b>Transposition Technique</b>	√					
<b>Boolean as Basic Operation</b>		√	√			
<b>Non-Boolean as Basic Operation</b>	√			√	√	√
<b>Chance of Alteration in Size</b>						√
<b>No Alteration in Size</b>	√	√	√	√		√
<b>Formation of Cycle</b>	√		√			
<b>No Formation of Cycle</b>		√		√	√	√

From table 1.5.1, we get the following attributes that are common to all the proposed techniques:

- **Implementation in Bit-Level**
- **Private Key System**
- **Block Cipher**

Also, except the RPSP technique, remaining all the techniques follow the principle of substitution, and except the RSBP technique, for remaining all techniques there is no chance of any alteration in size while encrypting the source file.

Since all the techniques are block ciphers and have to be implemented in bit-level, for all the techniques, the source file to be encrypted is to be converted into a stream of

bits and the whole stream is to be decomposed into a finite number of blocks before the actual scheme is to be implemented for each block.

Section 1.5.1 to section 1.5.6 give an overview for all the proposed techniques. Section 1.5.7 presents a brief proposal on the key structures of different proposed techniques. Section 1.5.8 points out the factors considered in the thesis for each of the proposed techniques for the purpose of evaluation.

### 1.5.1 An Overview of the RPSP Technique

The technique of RPSP involves in generating a cycle. A generating function,  $g(s, t)$ , used to generate a target block ( $t$ ) from a source block ( $s$ ), is applied for a block in the first iteration. The rules to be followed while applying the generating function  $g(s, t)$  are as follows [52, 55]:

1. A bit in the position  $i$  ( $1 \leq i \leq n-2$ ) in the block  $s$  becomes the bit in the position  $(n-i)$  in the block  $t$ , if  $(n-i)$  is a non-prime integer.
2. A bit in the position  $i$  ( $1 \leq i \leq (n-2)$ ) in the block  $s$  becomes the bit in the position  $j$  ( $1 \leq j \leq (n-i-1)$ ) in the block  $t$ , where  $j$  is the precedent prime integer (if any) of  $(n-i)$ , if  $(n-i)$  is a prime integer.
3. A bit in the position  $n$  in the block  $s$  remains in the same position in the block  $t$ .
4. A bit in the position  $(n-2)$  in the block  $s$  is transferred in the block  $t$  to the position unoccupied by any bit after rules 1, 2 and 3 are applied.

The same generating function is applied to all the subsequent blocks. After a finite number of such iterations, the source block is regenerated.

Preferably all blocks are not to be of the same size, so that the number of iterations required to regenerate the source block itself for one block may not be the same for an other block. These varying numbers of iterations are synchronized by evaluating the least common multiple (LCM) of all these numbers. If for all the blocks, iterations are applied for number of times exactly equal to the value of the LCM, it can be ensured that for each of the blocks its respective source block will be generated.

The value of the LCM is the total number of iterations required to complete the tasks of encryption as well as decryption. Therefore any intermediate value between 1 and the value of the LCM may be considered as the number of iterations performed during the process of encryption. For the purpose of decryption the same process is to be applied for the remaining number of times.

Thus, as shown in table 1.5.1, the technique consists of the following schematic characteristics:

- **Implementation in Bit-Level**
- **Symmetric**
- **Block Cipher**
- **Transposition Technique**
- **Non-Boolean as Basic Operation**
- **No Alteration in Size**
- **Formation of Cycle**

Also, according to table 1.5.1, it is a **private key system**, regarding which a discussion is presented in section 1.5.7.

Chapter 2 of the thesis discusses the RPSP technique in detail from different perspectives.

### **1.5.2 An Overview of the TE Technique**

The Triangular Encryption (TE) technique involves in generating a triangle-like shape for each of the blocks generated out of the source stream of bits during the process of encryption [48, 57].

From a source block of size  $N$  bits, a block of size  $(N-1)$  bits is generated using a Boolean algebra-based operation. In this operation, every two consecutive bits in the source block are exclusively ORed.

Recursively the same process is applied for all the subsequent blocks, so that finally a 1-bit block is generated in this approach. The source block and all the subsequent blocks, if are placed carefully in line-by-line manner, an equilateral triangle-like shape is formed. A total  $(N-1)$  number of such iterations are required to complete the triangle-like format.

Now, out of this constructed equilateral triangle, four options exist to be chosen as the encrypted block. These options are:

1. The block of N bits constructed from all the most significant bits (MSBs) starting from the source block to the finally generated 1-bit block
2. The block of N bits constructed from all the MSBs starting from the finally generated 1-bit block to the source block
3. The block of N bits constructed from all the least significant bits (LSBs) starting from the source block to the finally generated 1-bit block
4. The block of N bits constructed from all the LSBs starting from the finally generated 1-bit block to the source block

Now, out of these four options to be chosen as the encrypted block, blocks corresponding to option 1 and option 4 require exactly the same process to decrypt them, and for the same purpose different processes are to be adopted for blocks corresponding to option 2 and option 3.

Now, different blocks may not necessarily be of the same size and it is the key that will decide which option to be chosen for a certain block, and more about this is discussed in section 1.5.7, from which it can be clear why the system is a **private key system**.

As given in table 1.5.1, the TE technique consists of the following schematic characteristics, apart from the fact that it is a private key system.

- **Implementation in Bit-Level**
- **Both Symmetric and Asymmetric**
- **Block Cipher**
- **Substitution Technique**
- **Boolean as Basic Operation**
- **No Alteration in Size**
- **No Formation of Cycle**

Chapter 3 of the thesis discusses the TE technique in detail from different perspectives.

### 1.5.3 An Overview of the RPPO Technique

The RPPO technique also, like the RPSP technique, involves in forming a cycle. For a certain block of size  $N$  bits, a Boolean algebra-based operation is performed, so that a block of the same size is generated. The operation is as follows [45, 46, 49, 57]:

Let  $P = s_0^0 s_1^0 s_2^0 s_3^0 s_4^0 \dots s_{N-1}^0$  is a block of size  $N$  in the plaintext. Then the first intermediate block  $I_1 = s_0^1 s_1^1 s_2^1 s_3^1 s_4^1 \dots s_{N-1}^1$  can be generated from  $P$  in the following ways:

$$s_0^1 = s_0^0$$

$$s_i^1 = s_{i-1}^1 \oplus s_i^0, 1 * i * (N-1); \oplus \text{ stands for the exclusive OR operation.}$$

After applying this operation recursively for a fixed number of times, the original block is regenerated. This total number of iterations follows a mathematical policy.

For attaining a better security, either different blocks may be considered of different sizes, or even if blocks sizes are taken the same, intermediate blocks after different number of iterations are considered as the encrypted blocks, or both can be followed.

Section 1.5.7 discusses the possible approach in designing its key and hence shows why it is a **private key system**.

As given in table 1.5.1, different schematic characteristics of this technique are as follows:

- **Implementation in Bit-Level**
- **Symmetric**
- **Block Cipher**
- **Substitution Technique**
- **Boolean as Basic Operation**
- **No Alteration in Size**
- **Formation of Cycle**

Chapter 4 of the thesis discusses the RPPO technique in detail from different perspectives.

#### **1.5.4 An Overview of the RPMS Technique**

The technique of RPMS is asymmetric although the process of decryption can be easily be derived from that of encryption [50, 54].

To encrypt a certain block in the source stream the following policy is adopted:

The decimal equivalent of the block of bits under consideration is one integral value from which the recursive modulo-2 operation starts. The modulo-2 operation is performed to check if the integral value is even or odd. Then the position of that integral value in the series of natural even or odd numbers is evaluated. The same process is started again with this positional value. Recursively this process is carried out to a finite number of times, which is exactly the length of the source block. After each modulo-2 operation, 0 or 1 is pushed to the output stream in MSB-to-LSB direction; depending on the fact whether the integral value is even or odd respectively.

Corresponding to this encryption policy, the task of decryption is to be carried out in the following manner:

Bits in the encrypted block are to be considered along LSB-to-MSB direction, where obviously 0 stands for even and 1 stands for odd. Following the same logic in reverse manner we are to reach to the MSB, after which we get an integral value, the binary equivalent of which is the source block.

Obviously it is suggested to consider blocks to be of different sizes and the value of the key guides in this regard to encrypt the source stream of bits and decrypt the encrypted stream of bits. More about this and the reason behind the fact that it is a **private key system** is discussed in section 1.5.7.

As given in table 1.5.1, following are the schematic characteristics of this technique:

- **Implementation in Bit-Level**
- **Asymmetric**
- **Block Cipher**
- **Substitution Technique**
- **Non-Boolean as Basic Operation**
- **No Alteration in Size**
- **No Formation of Cycle**

Chapter 5 of the thesis discusses the RPMS technique in detail from different perspectives.

### **1.5.5 An Overview of the RSBP Technique**

Out of all the proposed techniques, the RSBP is the only technique in which there is a possibility of having alteration in the size of the encrypted file from the source one [50, 54].

In another respect, this block cipher is different from the others. Here for a certain block, bits generated during the process of encryption are not distributed contiguously. Instead, for all the blocks, code values and rank values are obtained, and to form the encrypted block, all code values sequentially are placed together, followed by all rank values preferably in exact reverse direction, and in between all code values and all rank values, a set of extra bits may have to be inserted to make the size of the encrypted stream a perfect multiple of eight for the convenience of converting into stream of characters.

For a certain block of size N, its 1-bit code value depends on whether its corresponding decimal value is prime or nonprime, and accordingly either 1 or 0 is set as its code value. On the other hand, the rank value represents the position of that prime/nonprime number in the sequence of prime/nonprime numbers. The size of the rank value depends on the minimum number of bits required to represent the position of the maximum prime/nonprime number possible to be generated out of the block of size N.

The uncertainty in the size of the encrypted stream corresponding to a source stream of a fixed size in turn ensures a better security.

The system is a **private key system**, more about which will be discussed in section 1.5.7.

As given in table 1.5.1, the system consists of the following set of schematic characteristics:

- **Implementation in Bit-Level**
- **Symmetric**
- **Block Cipher**

- **Substitution Technique**
- **Non-Boolean as Basic Operation**
- **Chance of Alteration in Size**
- **No Formation of Cycle**

Chapter 6 of the thesis discusses the RSBP technique in detail from different perspectives.

### **1.5.6 An Overview of the RSBM Technique**

Like the RSBP technique, here also for each of the blocks its code value and rank value are to be computed. Code values for all the blocks are to be placed together sequentially, followed by all rank values preferably in exact reverse direction, and in between the code value section and the rank value section, As per requirement to be able to convert into a stream of characters, a few extra bits, the number of which should not exceed seven, may have to be inserted [51].

Unlike the RSBP technique, here there is no possibility of any alteration of size through the process of encryption. If performing the modulo-2 operation on the decimal equivalent of the block, it is seen that the value is even, then 0 is set as its code value, and otherwise it is set to 1. Another modulo-2 operation is to be performed to find the position of that even/odd number in the sequence of natural even/odd numbers. For a block of size N bits, exactly  $(N-1)$  bits are required to represent this position in binary form. This  $(N-1)$ -bit block constitutes the rank value for the N-bit block. Due to the fact that given a block of a fixed size, its rank value is exactly 1 bit lesser in length, and the code value is represented by 1 bit, here there is no possibility of any alteration in size.

Blocks may be constructed to be of varying sizes and regarding this the entire thing will have to be controlled by the key. More about this and the reason behind the fact that it is a **private key system**, are discussed in section 1.5.7.

As given in table 1.5.1, following are the schematic characteristics of this technique:

- **Implementation in Bit-Level**
- **Asymmetric**
- **Block Cipher**

- **Substitution Technique**
- **Non-Boolean as Basic Operation**
- **No Alteration in Size**
- **No Formation of Cycle**

Chapter 7 of the thesis discusses the RSBM technique in detail from different perspectives.

### **1.5.7 A Proposal on Key Structures for Proposed Techniques**

Before proposing key structures of different proposed techniques, we categorize all the proposed techniques into following four types [6, 23, 24, 26]:

- **Block Cipher with Direct Block-to-Block Conversion:**  
Technique(s) in which the task of encryption is done block wise and for a block of length N, all the N bits of the corresponding encrypted block are placed contiguously, so that the one-to-one relationship between the source block and the encrypted block can be established.  
Example: The RPMS technique
- **Block Cipher with Option-based Block-to-Block Conversion:**  
Technique(s) in which the task of encryption is done block wise and for a block of length N, all the N bits of the corresponding encrypted block are placed contiguously (so far like category 1), but there exist more than one option to choose one block as the encrypted block, so that the one-to-many relationship between the source block and the encrypted block can be established.  
Example: The TE technique
- **Block Cipher with Non-Contiguous Bit-Allocation:**  
Technique(s) in which the task of encryption is done block wise but for a block of length N, different resultant bits are not placed contiguously, so that no one-to-one or one-to-many relationship between the source block and the corresponding encrypted block can be established.

Example: The RSBP and the RSBM techniques

- **Block Cipher with Repeated Block-to-Block Conversion:**

Technique(s) in which the task of encryption is done block wise and for a block of length N, after a finite number of specific iterations, the source block is regenerated, so that a cycle is formed. The one-to-many relationship between the source block and the encrypted block can be established as any of the intermediate blocks can be considered as the corresponding encrypted block.

Example: The RPSP and the RPPO techniques

#### **1.5.7.1 Proposed Key Structure for Block Cipher with Direct Block-to-Block Conversion**

To ensure a better security, varying lengths can be chosen for different blocks, and accordingly, the key is to be structured, so that from the key, one get the information regarding the lengths of different blocks. Once this information becomes available, the task of decryption can be performed easily to generate the exact source stream of bits. Naturally, this key is to be kept absolutely secret and hence the technique in this category is a **private key system**. Having different block lengths causes the key space to be reasonably large [12, 23, 43].

#### **1.5.7.2 Proposed Key Structures for Block Cipher with Option-based Block-to-Block Conversion**

In this case also it is proposed to choose varying block lengths and corresponding to each block, its length and the code of the option chosen also are to be included in the key. Obviously, the key is to be kept fully secret, so that it is a **private key system**. The existence of the one-to-many relationship between a source block and its corresponding encrypted block helps in enlarging the key space [12, 23, 43].

### **1.5.7.3 Proposed Key Structure for Block Cipher with Non-Contiguous Bit-Allocation**

Here the scenario is the same as category 1. Although in the technique of RSBP, choosing varying block lengths will cause the process of encryption as well as decryption a bit more complicated, but, if can be handled properly, this can produce the security of the highest level. Obviously, the key will consist of the information on the different blocks and, hence, it should be kept secret. As a result, these are **private key systems** [12, 23, 43].

### **1.5.7.4 Proposed Key Structure for Block Cipher with Repeated Block-to-Block Conversion**

In this case, for each of the blocks its length as well as the identification value of the intermediate block considered as the encrypted block are to be placed in the key. The key is to be made secret, so that the techniques falling under this category are **private key systems** [12, 23, 43].

More about the key structures of these different proposed techniques are discussed and analyzed in the respective chapters and also in chapter 9, which is especially based on the key distribution.

## **1.5.8 Factors considered for Evaluating Proposed Techniques**

Several factors have been considered to evaluate the proposed techniques. These include the following:

- **Frequency Distribution Test**
- **Chi Square Test**
- **Analysis of the Key Space**
- **Computation of the Encryption/Decryption Time**
- **Comparison of Performance with the RSA Technique**

### **1.5.8.1 Frequency Distribution Test**

Frequency distribution test is considered to asses the degree of security of the proposed techniques against the cryptanalytic attack. Through this test, performed

simultaneously on the original as well as the encrypted files, the frequencies of all 256 characters in two files are compared graphically. Vertical lines of the color blue stand for frequencies of different source characters, and those of the color red stand for frequencies of different encrypted characters. It is the objective of the frequency distribution test to show that there exists no fixed relationship between the frequency of a character in the source file and that of the same character in the corresponding encrypted file, and that the characters are well distributed in the encrypted file.

### 1.5.8.2 Chi Square Test

Through the chi square test performed between the original and the encrypted files, the non-homogeneity of the two files is tested.

The “Pearsonian Chi-square test” or the “Goodness-of-fit Chi-square test” has been performed here to decide whether the observations onto encrypted files are in good agreement with a hypothetical distribution, which means whether the sample of encrypted files may be supposed to have arisen from a specified population. In this case, the chi-square distribution is being performed with  $(256-1)=255$  degrees of freedom, 256 being the total number of classes of possible characters in the source as well as in the encrypted files. If the observed value of the statistic exceeds the tabulated value at a given level, the null hypothesis is rejected.

The “Pearsonian Chi-square” or the “Goodness-of-fit Chi-square” is defined as follows [44]:

$$\chi^2 = \sum \{(f_o - f_e)^2 / f_e\}$$

Here  $f_e$  and  $f_o$  respectively stand for the frequency of a character in the source file and that of the same character in the corresponding encrypted file.

On the basis of this formula, the Chi-square values have been calculated for sample pairs of source and encrypted files.

### 1.5.8.3 Analysis of the Key Space

The key space plays an important role in attempting to tackle the Brute-force attack successfully. The key space of each technique has been attempted to enlarge reasonably to make the techniques computationally secure [12, 23, 43].

#### **1.5.8.4 Computation of Encryption/Decryption Time**

The result of the encryption/decryption time plays an important role in assessing the efficiencies of the algorithms from the execution point of view. Here it has been attempted to establish a relationship between the size of the file being encrypted and the encryption/decryption time.

Now, the time consumed in encrypting and decrypting files is related to the code written for that purpose and the architecture of the machine where the code is being executed. All the results in this regard shown in different chapters are taken after compiling and executing C codes in a machine of the following configuration:

<b>Mother Board</b>	<b>: 810T</b>
<b>CPU</b>	<b>: 500 MHz Celeron</b>
<b>RAM</b>	<b>: 64 MB</b>
<b>HDD</b>	<b>: 20GB</b>

Now, the same code, if is executed on a machine of a different configuration, will require different amounts of time to encrypt and decrypt. Also, if the code is written in a different approach, it may offer different execution time even if it is run on the same machine. But such changes do not produce any change in the relationship between execution time and the source file size. Still there will exist the “almost linear nature” of the relationship.

#### **1.5.8.5 Comparison of Performance with the RSA Technique**

By comparing the performance of the proposed techniques with the RSA technique, it is attempted to evaluate the proposed techniques with respect to the existing field of cryptography. The most popular public key system, the RSA technique, has been considered here as the model and the comparative analysis is done on the basis of frequency distribution and chi square distribution for the purpose of evaluation.

### **1.6 A Note on Merits of Proposed Techniques**

All the six independent techniques included in this thesis offer security of highly satisfactory level. The statistical results have been shown in the respective chapters using

the frequency distribution test and the chi square test. For each technique, a reasonably large key space has been proposed, so that by the brute-force attack the chance of breaking the cipher by key estimation becomes computationally infeasible. Moreover, implementation of each of the algorithms is well tested with satisfactory performance. The execution time varies with the size of the file being encrypted. Only one out of the six proposed techniques causes alteration in size of file after being encrypted. But, in turn, this alteration in size helps in ensuring a better security and space efficiency also.

Following is the list of all the chapters in which all the proposed independent techniques have been presented:

- Chapter 2 entitled, ***Encryption Through Recursive Positional Substitution Based on Prime-Nonprime (RPSP) of Cluster***, based on the RPSP technique
- Chapter 3 entitled, ***Encryption Through Triangular Encryption (TE) Technique***, based on the TE technique
- Chapter 4 entitled, ***Encryption Through Recursive Paired Parity Operation (RPPO)***, based on the RPPO technique
- Chapter 5 entitled, ***Encryption Through Recursive Positional Modulo-2 Substitution (RPMS) Technique***, based on the RPMS technique
- Chapter 6 entitled, ***Encryption Through Recursive Substitutions of Bits Through Prime-Nonprime (RSBP) Detection of Sub-stream***, based on the RSBP technique
- Chapter 7 entitled, ***Encryption Through Recursive Substitutions of Bits Through Modulo-2 (RSBM) Detection of Sub-stream***, based on the RSBP technique
- The structure of the key plays the most important role, as all the proposed techniques are private key systems. The following chapter discusses this aspect:
- Chapter 8 entitled, ***Formation of Secret Key***, based on suitable key structure for the proposed techniques

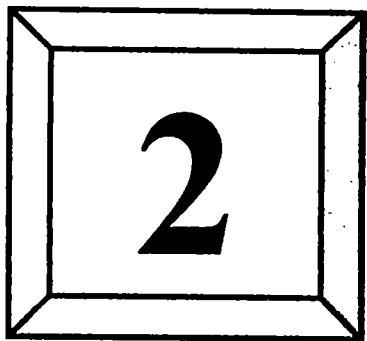
Apart from introducing six independent cryptographic techniques, there is also one proposal to implement these techniques in cascaded manner. The following chapter shows how this cascading can be performed tactfully:

- Chapter 9 entitled, *Encryption Through Cascaded Implementation of the Proposed Techniques*, based on the cascaded approach

Finally, a conclusive discussion on the entire development work, including the final assessment of all the proposed cryptographic systems, is done in the following chapter:

- Chapter 10 entitled, *A Conclusive Discussion*, based on the conclusion on the entire work

The references are included in Appendix A. List of publications of the candidate is included in Appendix B. Source codes of proposed implemented techniques are included in Appendix C. Bibliography is included in Appendix D.



## **Encryption Through Recursive Positional Substitution Based on Prime-Nonprime (RPSP) of Cluster**

<u>Contents</u>	<u>Pages</u>
<b>2.1      Introduction</b>	<b>53</b>
<b>2.2      The Scheme</b>	<b>54</b>
<b>2.3      Implementation</b>	<b>59</b>
<b>2.4      Results</b>	<b>66</b>
<b>2.5      Analysis</b>	<b>80</b>
<b>2.6      Conclusion</b>	<b>87</b>

## 2.1 Introduction

The proposed technique in this chapter named, Encryption through Recursive Positional Substitution based on Prime-nonprime of cluster or RPSP, is a secret-key cryptosystem.

In this technique, after decomposing the source stream of bits into a finite number of blocks of finite length, the positions of the bits of each of the blocks is re-oriented using a generating function. For a particular length of block, the block itself is regenerated after a finite number of such iterations. Any of the intermediate blocks during this cycle is considered to be the encrypted block. To decrypt the encrypted block from the ciphertext, the same process is to be followed but the generating function may have to be applied different number of times.

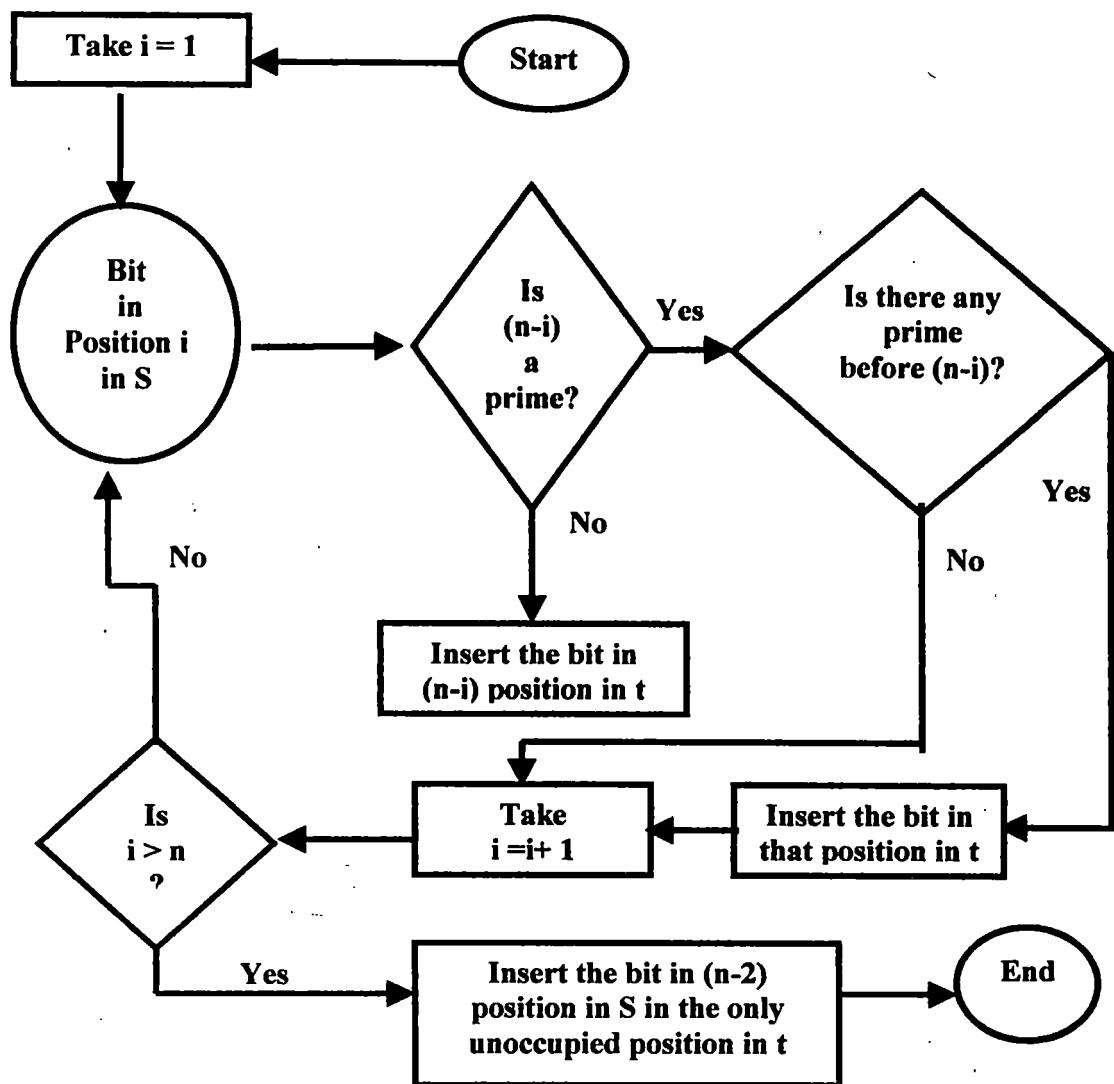
To achieve the security of a satisfactory level, it is proposed that different blocks or blocks should be of different sizes. Accordingly, for different blocks, number of iterations during the encryption and the number of iterations during the decryption also not necessarily should be fixed. This information in a proposed fixed format, described later in this chapter, constitutes the secret key for the system, which is to be transmitted by the sender to the receiver, either with the message or in an isolated manner.

The technique does not cause any storage overhead. It provides a large key space, so that the chance of breaking the ciphertext is almost nullified by any technique of cryptanalysis. The implementation on practical scenario is well proven with positive outcome [3, 52, 55].

Section 2.2 of this chapter describes the scheme of this technique with simple examples. Since the entire scheme is the combination of the encryption and the decryption processes, this section also includes how one part of the scheme can be used for the encryption and how the remaining part can be used for the decryption. Section 2.3 shows a simple implementation of the technique, where a sample text message has been considered for the purpose of transmission using the encryption. Section 2.4 gives the results obtained after implementing the RPSP technique on a number of real-time files of different categories like .exe, .com, .dll, etc. Section 2.5 is an analytical presentation of the technique, where the RPSP technique has been analyzed from different perspectives. Section 2.6 draws a conclusion on the technique.

## 2.2 The Scheme

The technique considers the plaintext as a stream of finite number of bits  $N$ , and is divided into a finite number of blocks, each also containing a finite number of bits  $n$ , where  $1 \leq n \leq N$ . Here orientation of the position of bits within a block is performed through a generating function. If we repeat the operation using the same generating function, the original block is regenerated after a finite number of iterations forming a cycle [38, 42, 46, 52].



**Figure 2.2.1**  
Overview of Entire Technique

The generating function  $g(s, t)$ , aiming to orient the positions of different bits, is applied on a block  $s$  of size  $n$  to generate an intermediate block  $t$  of the same size. The intermediate block  $t$  is generated by the following rules:

1. A bit in the position  $i$  ( $1 \leq i \leq n-2$ ) in the block  $s$  becomes the bit in the position  $(n-i)$  in the block  $t$ ; if  $(n-i)$  is a non-prime integer.
2. A bit in the position  $i$  ( $1 \leq i \leq (n-2)$ ) in the block  $s$  becomes the bit in the position  $j$  ( $1 \leq j \leq (n-i-1)$ ) in the block  $t$ , where  $j$  is the precedent prime integer (if any) of  $(n-i)$ , if  $(n-i)$  is a prime integer.
3. A bit in the position  $n$  in the block  $s$  remains in the same position in the block  $t$ .
4. A bit in the position  $(n-2)$  in the block  $s$  is transferred in the block  $t$  to the position unoccupied by any bit after rules 1, 2 and 3 are applied.

The technique is illustrated using a flow diagram given in figure 2.2.1.

Now, for a block of finite size ( $n$ ), a finite number of iterations ( $I$ ) are required to regenerate the source block, where for an iteration, the source is the target block of the previous block and for the 1<sup>st</sup> iteration, the source is the source block of the entire technique. It can be shown that if for the  $p^{\text{th}}$  bit from MSB ( $1 \leq p \leq n$ ) in the source block,  $i_p$  is the number of iterations required to be re-oriented to its source position, the total number of iterations ( $I$ ) required to regenerate the source block is LCM of  $i_1, i_2, i_3, \dots, i_p$ .

The process of encryption is the sub set of the entire set of work to form the cycle for a given block. Any intermediate block during the process of forming the cycle may be considered as the encrypted block. Hence if the number of iterations required to form the cycle is  $I$  for a block of size  $n$ , the number of intermediate blocks generated in the cycle is  $(I-1)$ , because each iteration generates one block and the final iteration generates the target block, which, in turn, is the source block. Therefore if the  $p^{\text{th}}$  block formed in the cycle is considered to be the encrypted block,  $p$  may vary from 1 to  $(I-1)$ .

Like in encryption, the process of decryption is also a sub set of the entire set of work required to form the cycle for the block. The only difference is that one intermediate block of the cycle (the encrypted block) is considered to be the source block

in the process of decryption. If the block generated after the  $p^{\text{th}}$  iteration in the cycle ( $1 \leq p \leq (I-1)$ ,  $I$  being the total number of iterations required to form the cycle) is considered to be the encrypted block, number of iterations required to decrypt the encrypted block is  $(I-p)$ .

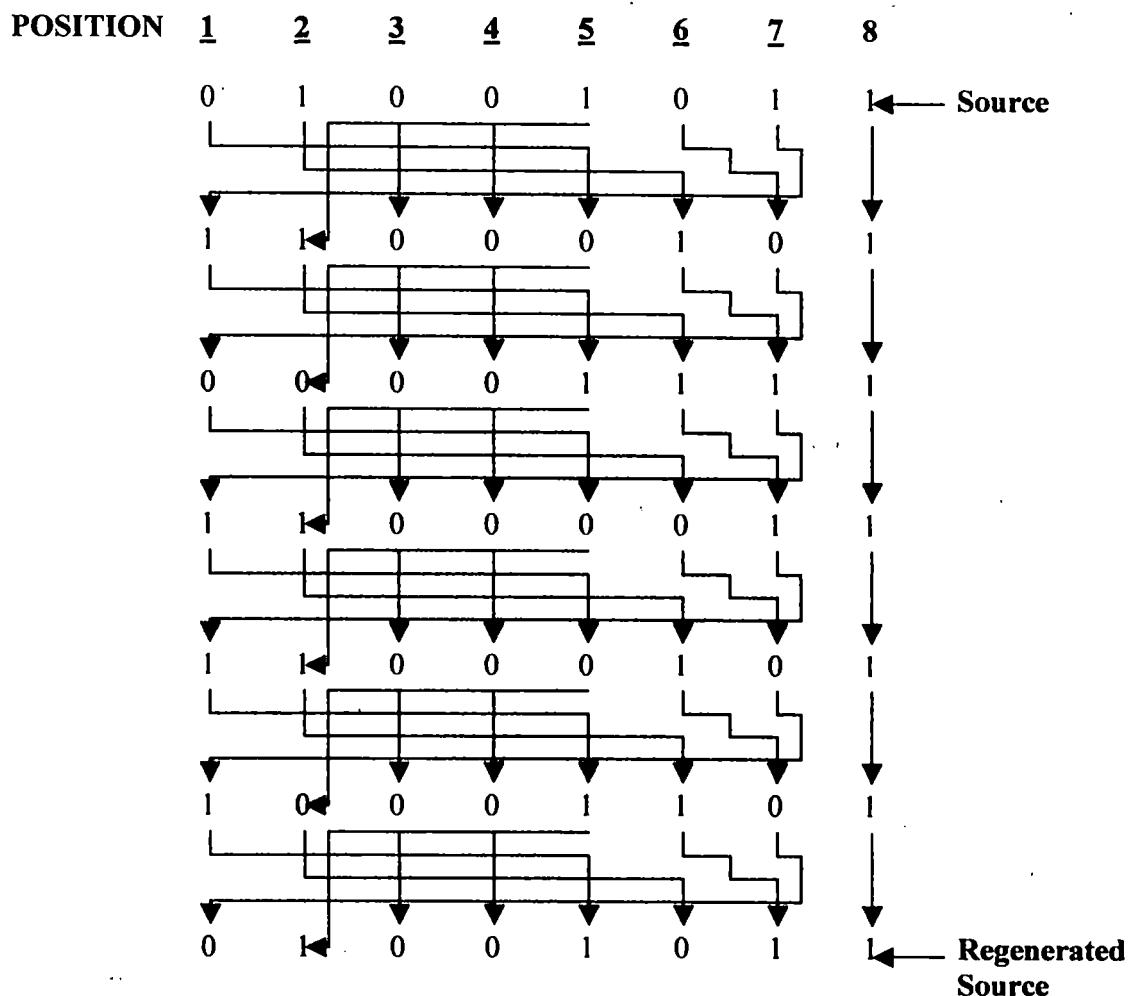
### 2.2.1 Example

Reorientation of bits in a finite length of block can be explained better with an example. Let us consider the block  $s = 01001011$  of size 8 bits. Figure 2.2.1.1 depicts the different intermediate blocks and the target block generated during the encoding process. The Flow diagram to show how positions of the bits of  $s$  and the different intermediate blocks can be reoriented to complete the cycle is shown in figure 2.2.1.2. In this diagram, each arrow indicates positional orientation of a bit during iteration.

Source Block								
0	1	0	0	1	0	1	1	
1	2	3	4	5	6	7	8	
Block After 1 <sup>st</sup> Iteration								
1	1	0	0	0	1	0	1	
1	2	3	4	5	6	7	8	
Block After 2 <sup>nd</sup> Iteration								
0	0	0	0	1	1	1	1	
1	2	3	4	5	6	7	8	
Block After 3 <sup>rd</sup> Iteration								
1	1	0	0	0	0	1	1	
1	2	3	4	5	6	7	8	
Block After 4 <sup>th</sup> Iteration								
1	0	0	0	1	1	0	1	
1	2	3	4	5	6	7	8	
Block After 5 <sup>th</sup> Iteration (Source Block)								
0	1	0	0	1	0	1	1	
1	2	3	4	5	6	7	8	

**Figure 2.2.1.1**  
**Different Intermediate and Target Blocks Generated in**  
**Forming the Cycle for Block 01001011**

Here since number of iterations ( $I$ ) required to form the cycle is 5, the  $p^{\text{th}}$  block may be considered as the encrypted block, where  $p$  ranges from 1 to 4.



**Figure 2.2.1.2**  
**Formation of a Cycle for the Block 01001011**

For this given example, the mapping ( $X \rightarrow Y$ ) between the position (X) of a bit in source block s and that (Y) of the same bit in target block t is shown in table 2.2.1.1 along with the logic followed. Here the block size  $n = 8$ . Table 2.2.1.2 shows the flow of bits As per the proposed algorithm and number of iterations needed to regenerate the 8-bit source block considering "01001001" as the input stream.

**Table 2.2.1.1**  
**Illustration of Mapping ( $X \rightarrow Y$ ) for Block of Size 8**

X	Y	Logic Followed
1	5	Here $8 - 1 = 7$ , a prime; the precedent prime of 7 is 5
2	6	Here $8 - 2 = 6$ , a non-prime
3	3	Here $8 - 3 = 5$ , a prime, the precedent prime of 5 is 3
4	4	Here $8 - 4 = 4$ , a non-prime
5	2	Here $8 - 5 = 3$ , a prime, the precedent prime of 3 is 2
6	?	Here $8 - 6 = 2$ , a prime, there is no precedent prime, allocation suspended
7	1	Here $8 - 7 = 1$ , a non-prime
8	8	Here 8 being the position of the LSB, no change in position
6	7	One allocation was suspended earlier; since the position 7 is only the unoccupied position so far, that allocation is made there

**Table 2.2.1.2**  
**Illustration of Number of Iterations Required to Regenerate the Block of 8 Bits**

Position in the Source Block $p$ ( $0 \leq p \leq 7$ )	Reorientation of Position through Different Steps					Total No. of Iterations Required to Retain the Original Position $i_p$ ( $0 \leq p \leq 7$ )
	Step 1	Step 2	Step 3	Step 4	Step 5	
1	$1 \rightarrow 5$	$5 \rightarrow 2$	$2 \rightarrow 6$	$6 \rightarrow 7$	$7 \rightarrow 1$	5
2	$2 \rightarrow 6$	$6 \rightarrow 7$	$7 \rightarrow 1$	$1 \rightarrow 5$	$5 \rightarrow 2$	5
3	$3 \rightarrow 3$	--	--	--	--	1
4	$4 \rightarrow 4$	--	--	--	--	1
5	$5 \rightarrow 2$	$2 \rightarrow 6$	$6 \rightarrow 7$	$7 \rightarrow 1$	$1 \rightarrow 5$	5
6	$6 \rightarrow 7$	$7 \rightarrow 1$	$1 \rightarrow 5$	$5 \rightarrow 2$	$2 \rightarrow 6$	5
7	$7 \rightarrow 1$	$1 \rightarrow 5$	$5 \rightarrow 2$	$2 \rightarrow 6$	$6 \rightarrow 7$	5
8	$8 \rightarrow 8$	--	--	--	--	1
<b>Total No. of Iterations Required to Regenerate The Block</b>	<b>LCM of 5, 5, 1, 1, 5, 5, 5, 1</b>					<b>5</b>

### **2.2.1.1 Example of Encryption**

Let us take the block  $s = 01001011$ . considered in section 2.2.1. As shown in table 2.2.1.2, the number of iterations required to form the cycle is 5. Figure 2.2.1.1 shows different intermediate and target block generated through different iterations in the cycle.

### **2.2.1.2 Example of Decryption**

Following the example given in section 2.2.1.2, if the block  $00001111$ , generated after iteration 2, is considered to be the encrypted block, number of iterations required to decrypt the encrypted block is  $5-2=3$ .

## **2.3 Implementation**

In this section, we will consider a simple text message that is to be transmitted by encrypting using RPSP technique and after the transmission is over the encrypted message is to be decrypted using the same technique.

Consider the plaintext (P) as: **Data Encryption**. This stream is taken as the source stream.

Table 2.3.1 shows how each character in P can be converted into a byte.

**Table 2.3.1**  
**Character-to-Byte Conversion for the String “Data Encryption”**

Character	Byte
D	01000100
a	01100001
t	01110100
a	01100001
<blank>	11111111
E	01000101
n	01101110
c	01100011
r	01110010
y	01111001
p	01110000
t	01110100
i	01101001
o	01101111
n	01101110

Combining all these bytes we obtain the following source stream of bits (S) consisting of 120 bits, where “/” is used as the separator between two consecutive bytes:

01000100/01100001/01110100/01100001/11111111/01000101/01101110/01100011/  
 01110010/01111001/01110000/01110100/01101001/01101111/01101110.

For the simplicity, let us take block size as 16, so that the number of source blocks is 7 and the final 8 bits are being kept as it is.

Now, using the analogy of table 2.2.1.1, another table may be formed to illustrate the mapping ( $X \rightarrow Y$ ) for blocks of size 16. Table 2.3.2 illustrates the same.

**Table 2.3.2**  
**Illustration of Mapping ( $X \rightarrow Y$ ) for Source Block of Size 16**

X	Y	Logic Followed
1	15	Here $16 - 1 = 15$ , a non-prime
2	14	Here $16 - 2 = 14$ , a non-prime
3	11	Here $16 - 3 = 13$ , a prime, the precedent prime of 13 is 11
4	12	Here $16 - 4 = 12$ , a non-prime
5	7	Here $16 - 5 = 11$ , a prime, the precedent prime of 11 is 7
6	10	Here $16 - 6 = 10$ , a non-prime
7	9	Here $16 - 7 = 9$ , a non-prime
8	8	Here $16 - 8 = 8$ , a non-prime
9	5	Here $16 - 9 = 7$ , a prime, the precedent prime of 7 is 5
10	6	Here $16 - 10 = 6$ , a non-prime
11	3	Here $16 - 11 = 5$ , a prime, the precedent prime of 5 is 3
12	4	Here $16 - 12 = 4$ , a non-prime
13	2	Here $16 - 13 = 3$ , a prime, the precedent prime of 3 is 2
14	?	Here $16 - 14 = 2$ , a prime, but since it has no precedent prime, allocation is temporarily suspended
15	1	Here $16 - 15 = 1$ , a non-prime
16	16	Being the position of the LSB, it is kept as it is
14	13	One allocation was suspended earlier; since the position 13 is only the unoccupied position so far, that allocation is made there

Now, the source blocks generated from the source stream are:

$$S_1 = 0100010001100001 \quad S_2 = 0111010001100001$$

$$S_3 = 1111111101000101 \quad S_4 = 0110111001100011$$

$$S_5 = 0111001001111001 \quad S_6 = 0111000001110100$$

$$S_7 = 0110100101101111$$

It is seen that for the block of size 16, the number of iterations required to regenerate the source itself is 6. Table 2.3.3 to Table 2.3.9 show the formation of cycles for blocks  $S_1$ ,  $S_2$ ,  $S_3$ ,  $S_4$ ,  $S_5$ ,  $S_6$  and  $S_7$  respectively. Now, for each of the blocks, an arbitrary intermediate block, as indicated in each table, is considered as the encrypted stream.

**Table 2.3.3**  
**Formation of Cycle for Block S<sub>1</sub>**

<b>Source Block</b>	0100010001100001
<b>Block (I<sub>11</sub>) after iteration 1</b>	0010010001000101
<b>Block (I<sub>12</sub>) after iteration 2</b>	0000010001101001
<b>Block (I<sub>13</sub>) after iteration 3</b>	0110010001000001
<b>Block (I<sub>14</sub>) after iteration 4</b>	0000010001100101
<b>Block (I<sub>15</sub>) after iteration 5</b>	0010010001001001
<b>Block after iteration 6 (Source Block)</b>	0100010001100001

**Encrypted Block**

**Table 2.3.4**  
**Formation of Cycle for Block S<sub>2</sub>**

<b>Source Block</b>	0111010001100001
<b>Block (I<sub>21</sub>) after iteration 1</b>	0010010001110101
<b>Block (I<sub>22</sub>) after iteration 2</b>	0011010001101001
<b>Block (I<sub>23</sub>) after iteration 3</b>	0110010001110001
<b>Block (I<sub>24</sub>) after iteration 4</b>	0011010001100101
<b>Block (I<sub>25</sub>) after iteration 5</b>	0010010001111001
<b>Block after iteration 6 (Source Block)</b>	0111010001100001

**Encrypted Block**

**Table 2.3.5**  
**Formation of Cycle for Block S<sub>3</sub>**

<b>Source Block</b>	1111111101000101
<b>Block (I<sub>31</sub>) after iteration 1</b>	0000011111111111
<b>Block (I<sub>32</sub>) after iteration 2</b>	1111110111001001
<b>Block (I<sub>33</sub>) after iteration 3</b>	0100111011101111
<b>Block (I<sub>34</sub>) after iteration 4</b>	101101111001101
<b>Block (I<sub>35</sub>) after iteration 5</b>	010011011111011
<b>Block after iteration 6 (Source Block)</b>	1111111101000101

**Encrypted Block**

**Table 2.3.6**  
**Formation of Cycle for Block S<sub>4</sub>**

<b>Source Block</b>	0110111001100011
<b>Block (I<sub>41</sub>) after iteration 1</b>	1010011011100101
<b>Block (I<sub>42</sub>) after iteration 2</b>	0010110011101011
<b>Block (I<sub>43</sub>) after iteration 3</b>	1110111001100001
<b>Block (I<sub>44</sub>) after iteration 4</b>	0010011011100111
<b>Block (I<sub>45</sub>) after iteration 5</b>	1010110011101001
<b>Block after iteration 6 (Source Block)</b>	0110111001100011

**Encrypted Block**

**Table 2.3.7**  
**Formation of Cycle for Block S<sub>5</sub>**

<b>Source Block</b>	0111001001111001
<b>Block (I<sub>51</sub>) after iteration 1</b>	0111010010110101
<b>Block (I<sub>52</sub>) after iteration 2</b>	0011100001111101
<b>Block (I<sub>53</sub>) after iteration 3</b>	0111011000111001
<b>Block (I<sub>54</sub>) after iteration 4</b>	0111000011110101
<b>Block (I<sub>55</sub>) after iteration 5</b>	0011110000111101
<b>Block after iteration 6 (Source Block)</b>	0111001001111001

**Encrypted Block**

**Table 2.3.8**  
**Formation of Cycle for Block S<sub>6</sub>**

<b>Source Block</b>	0111000001110100
<b>Block (I<sub>61</sub>) after iteration 1</b>	0011010000111100
<b>Block (I<sub>62</sub>) after iteration 2</b>	0111000001111000
<b>Block (I<sub>63</sub>) after iteration 3</b>	0111010000110100
<b>Block (I<sub>64</sub>) after iteration 4</b>	0011000001111100
<b>Block (I<sub>65</sub>) after iteration 5</b>	0111010000111000
<b>Block after iteration 6 (Source Block)</b>	0111000001110100

**Encrypted Block**

**Table 2.3.9**  
**Formation of Cycle for Block S<sub>7</sub>**

<b>Source Block</b>	0110100101101111
<b>Block (I<sub>71</sub>) after iteration 1</b>	1110011100101101
<b>Block (I<sub>72</sub>) after iteration 2</b>	011000011101111
<b>Block (I<sub>73</sub>) after iteration 3</b>	1110110100101101
<b>Block (I<sub>74</sub>) after iteration 4</b>	0110001101101111
<b>Block (I<sub>75</sub>) after iteration 5</b>	1110010110101101
<b>Block after iteration 6 (Source Block)</b>	0110100101101111

**Encrypted Block**

As indicated in tables 2.3.3 to 2.3.9, intermediate blocks I<sub>14</sub> (0000010001100101), I<sub>21</sub> (0010010001110101), I<sub>33</sub> (010011101110111), I<sub>42</sub> (0010110011101011), I<sub>55</sub> (0011110000111101), I<sub>62</sub> (0111000001111000) and I<sub>72</sub> (011000011101111) are considered as the encrypted blocks, so that these, together with the last 8-bit block (01101110) that was kept as it is, form the encrypted stream as follows:

0000010001100101/0010010001110101/010011101110111/0010110011101011/00111  
 10000111101/0111000001111000/0110000111101111/01101110, “/” being used as only the separator.

The encrypted stream can be rewritten as the series of bytes as follows:

00000100/01100101/00100100/01110101/01001111/01110111/00101100/11101011/  
 00111100/00111101/01110000/01111000/01100001/11101111/01101110.

Converting the bytes into the corresponding characters, the following text is obtained as the encrypted text, which is to be transmitted:

$$C = \text{Xe\$uOw,8<=pxa\square n}$$

Now, the process for the decryption is the same as the encryption. After converting the ciphertext C into a stream of bits, it is to be decomposed into the 16-bit blocks (C<sub>1</sub>, C<sub>2</sub>, ..., C<sub>7</sub>). For each block, the same process as the encryption process is to

be followed but for varying number of iterations. For example, while decrypting the encrypted block  $C_1$ , the number of iterations should be  $6 - 4 = 2$ , as the encrypted block was obtained after 4 iterations and forming a cycle requires a total of 6 iterations. The same logic is to be applied for the remaining blocks. After obtaining the source blocks in this way, they are combined together along with the last 8-bit block, which was kept as it is, in the same sequence and thus the source stream of bit is obtained, from which the source text is regenerated.

## 2.4 Results

For the purpose of the practical implementation, like all the other techniques, RPSP technique has also been implemented on total 50 files. Out of these, there are *EXE*, *COM*, *DLL*, *SYS* and *CPP* files, each category being 10 in number.

Section 2.4.1 presents report of the encryption/decryption times and the Chi Square values. Section 2.4.2 presents result of frequency distribution tests. A comparative result with the RSA system is given in section 2.4.3.

### 2.4.1 Result for Encryption/Decryption Time and Chi Square Value

Section 2.4.1.1 shows the result on *EXE* files, section 2.4.1.2 shows the result on *COM* files, section 2.4.1.3 shows the result on *DLL* files, section 2.4.1.4 shows the result on *SYS* files and section 2.4.1.5 shows the result on *CPP* files. In all the cases, the sample blocks are taken of the same 64-bit size. In this situation, it is seen that the total number of iterations required to regenerate the source block is 84. During encrypting the source files, the number of iterations is the integral part of the half of the total number of iterations required to regenerate the blocks, i.e.,  $(\text{int}) (84/2) = 42$ , in this case. The remaining iterations are performed during the decryption [44, 51, 55, 56].

Section 2.4.1.6 provides a report on the results for different block sizes to show how the encryption time changes accordingly for the same sample file.

#### **2.4.1.1 Result for *EXE* Files**

Table 2.4.1.1.1 gives the result of implementing the RPSP technique on different executable files. The result includes the source size, the time for encryption and the time for decryption. From the table, it is clear that the encryption and the decryption time increase with the increment in the size of the source file.

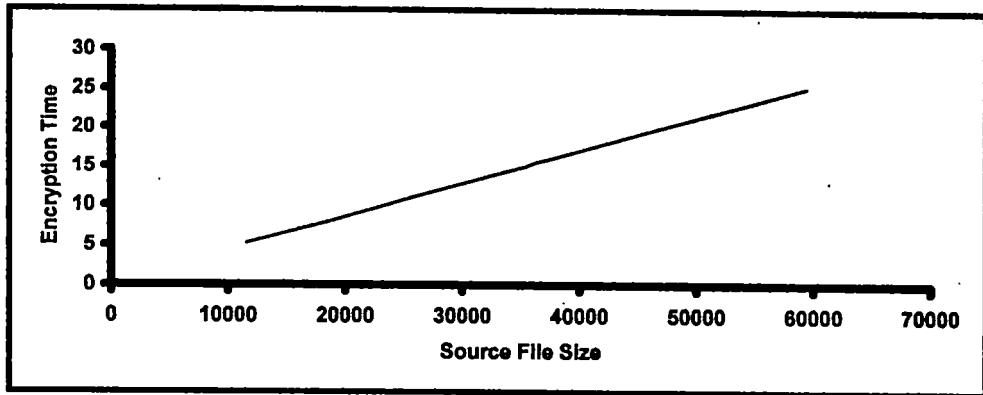
Ten executable files are taken. The size of the files varies from 11611 bytes to 59398 bytes. The encryption time varies from 5.2747 seconds to 25.2198 seconds, whereas the decryption time also varies from 5.2747 seconds to 25.2198 seconds. The values of the Chi Square test results vary from 2239 to 19973 with the degree of freedom varies from 248 to 255. From these Chi Square values a high degree of non-homogeneity of each encrypted file is seen in comparison to the corresponding source file.

**Table 2.4.1.1.1  
Result for *EXE* Files**

Source File	Encrypted File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	Chi Square Value	Degree of Freedom
<i>TLIB.EXE</i>	<i>A1.EXE</i>	37220	15.8791	15.8242	14479	255
<i>MAKER.EXE</i>	<i>A2.EXE</i>	59398	25.2198	25.2198	19973	255
<i>UNZIP.EXE</i>	<i>A3.EXE</i>	23044	9.8352	9.7802	2239	255
<i>RPPO.EXE</i>	<i>A4.EXE</i>	35425	15.0000	15.0000	5277	255
<i>PRIME.EXE</i>	<i>A5.EXE</i>	37152	15.8242	15.7692	5485	255
<i>TCDEF.EXE</i>	<i>A6.EXE</i>	11611	5.2747	5.2747	3791	254
<i>TRIANGLE.EXE</i>	<i>A7.EXE</i>	36242	15.4945	15.4945	5690	255
<i>PING.EXE</i>	<i>A8.EXE</i>	24576	10.4945	10.4396	4335	248
<i>NETSTAT.EXE</i>	<i>A9.EXE</i>	32768	13.9011	13.9011	8725	255
<i>CLIPBRD.EXE</i>	<i>A10.EXE</i>	18432	7.8571	7.8571	4964	255

Here since the operations while encrypting and decrypting are almost the same, the times needed for the encryption and the decryption are of little or no difference.

Figure 2.4.1.1.1 depicts the relationship between the source file size and the encryption time for *EXE* files. As is shown in the graph in figure 2.4.1.1.1, the encryption time increases almost linearly with the source file size.



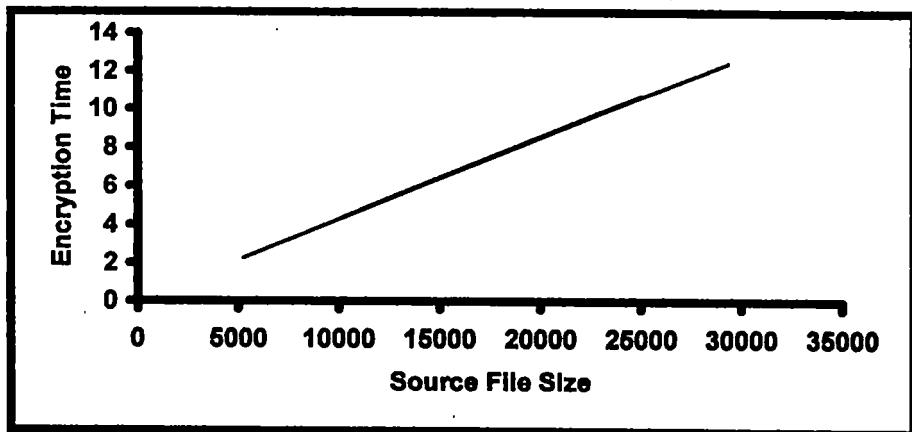
**Figure 2.4.1.1.1**  
**Graph to establish Relationship between Source Size and Encryption Time**  
**(For EXE Files)**

#### 2.4.1.2 Result for COM Files

Table 2.4.1.2.1 shows the result of implementing the technique on 10 sample COM files. Ten command files are taken. The size of the files varies from 5239 bytes to 29271 bytes. The encryption time varies from 2.2527 seconds to 12.4176 seconds, whereas the decryption time also varies from 2.2527 seconds to 12.4725 seconds. The values of the Chi Square test results vary from 731 to 5019 with the degree of freedom vary from 230 to 255. From these Chi Square values a high degree of non-homogeneity of each encrypted file is seen in comparison to the corresponding source file.

**Table 2.4.1.2.1**  
**Result for .COM Files**

Source File	Encrypted File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	Chi Square Value	Degree of Freedom
EMSTEST.COM	A1.COM	19664	8.4066	8.4066	3890	255
THELP.COM	A2.COM	11072	4.7802	4.7253	2563	250
WIN.COM	A3.COM	24791	10.6044	10.5494	3887	252
KEYB.COM	A4.COM	19927	8.5165	8.4615	4023	255
CHOICE.COM	A5.COM	5239	2.2527	2.2527	1225	232
DISKCOPY.COM	A6.COM	21975	9.3407	9.3407	4221	254
DOSKEY.COM	A7.COM	15495	6.5934	6.5934	4105	253
MODE.COM	A8.COM	29271	12.4176	12.4725	5019	255
MORE.COM	A9.COM	10471	4.5055	4.4505	731	230
SYS.COM	A10.COM	18967	8.0769	8.0769	3131	254



**Figure 2.4.1.2.1**  
**Graph to establish Relationship between Source Size and Encryption Time  
(For COM Files)**

Figure 2.4.1.2.1 shows the graphical relationship between the source size and the encryption time for *COM* files. As shown in the figure, the relationship is linear. This means that the encryption time increases linearly with the size of the source file considered for encryption.

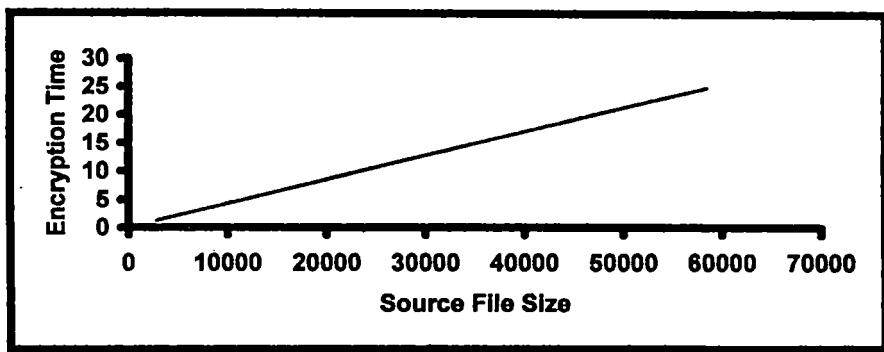
#### 2.4.1.3 Result for DLL Files

Table 2.4.1.3.1 gives the result of implementing the technique on 10 sample *DLL* files. Ten *dll* files are taken. The size of the files varies from 3216 bytes to 58368 bytes. The encryption time varies from 1.4286 seconds to 24.8352 seconds, whereas the decryption time also varies from 1.3736 seconds to 24.8352 seconds. The values of the Chi Square test results vary from 872 to 23953 with the degree of freedom vary from 217 to 255. From these Chi Square values a high degree of non-homogeneity of each encrypted file is seen in comparison to the corresponding source file.

**Table 2.4.1.3.1**  
**Result for .DLL Files**

Source File	Encrypted File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	Chi Square Value	Degree of Freedom
<i>SNMPAPI.DLL</i>	<i>A1.DLL</i>	32768	13.9560	13.9011	5987	253
<i>KPSHARP.DLL</i>	<i>A2.DLL</i>	31744	13.5165	13.5165	23318	254
<i>WINSOCK.DLL</i>	<i>A3.DLL</i>	21504	9.1758	9.1758	9648	252
<i>SPWHPT.DLL</i>	<i>A4.DLL</i>	32792	13.9560	13.9560	7781	255
<i>HIDCI.DLL</i>	<i>A5.DLL</i>	3216	1.4286	1.3736	872	217
<i>PFPICK.DLL</i>	<i>A6.DLL</i>	58368	24.8352	24.8352	12324	255
<i>NDDEAPI.DLL</i>	<i>A7.DLL</i>	14032	5.9890	5.9890	4693	249
<i>NDDENB.DLL</i>	<i>A8.DLL</i>	10976	4.7253	4.7253	8269	251
<i>ICCCODES.DLL</i>	<i>A9.DLL</i>	20992	8.9011	8.9560	9693	252
<i>KPSCALE.DLL</i>	<i>A10.DLL</i>	31232	13.2967	13.2967	23953	255

Figure 2.4.1.3.1 shows the relationship between the source file size and the encryption time for these *DLL* files. There exists a linear relationship between these two parameters. This proves that for *DLL* files also the encryption time increase linearly with the source file size.



**Figure 2.4.1.3.1**  
**Graph to establish Relationship between Source Size and Encryption Time  
(For *DLL* Files)**

#### 2.4.1.4 Result for *SYS* Files

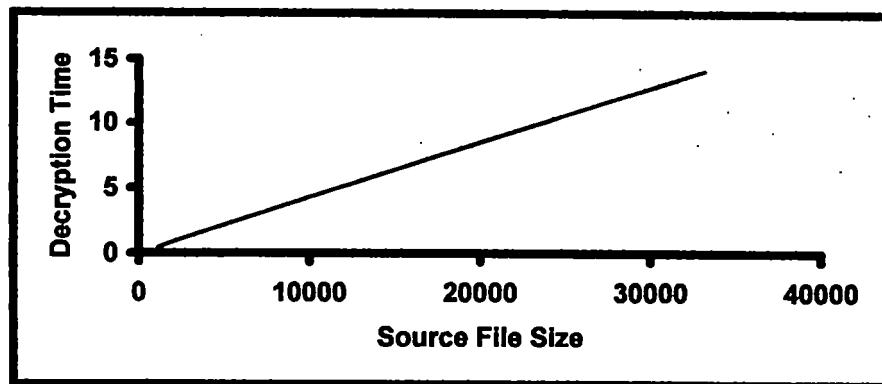
Table 2.4.1.4.1 shows the result after implementing the technique on 10 sample *SYS* files. Ten system files are taken. The size of the files varies from 1105 bytes to 33191 bytes. The encryption time varies from 1.1538 seconds to 14.1758 seconds,

whereas the decryption time also varies from 1.1538 seconds to 14.1209 seconds. The values of the Chi Square test results vary from 169 to 20163 with the degree of freedom vary from 165 to 255. From these Chi Square values a high degree of non-homogeneity of each encrypted file is seen in comparison to the corresponding source file.

**Table 2.4.1.4.1  
Result for SYS Files**

Source File	Encrypted File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	Chi Square Value	Degree of Freedom
HIMEM.SYS	A1.SYS	33191	14.1758	14.1209	9189	255
RAMDRIVE.SYS	A2.SYS	12663	5.3846	5.3846	1425	241
USBD.SYS	A3.SYS	18912	8.0769	8.0220	9225	255
CMD640X.SYS	A4.SYS	24626	10.4945	10.4396	6248	255
CMD640X2.SYS	A5.SYS	20901	8.8462	8.9011	5759	255
REDBOOK.SYS	A6.SYS	5664	2.4725	2.4176	1946	230
IFSHLP.SYS	A7.SYS	3708	1.5934	1.5934	1040	237
ASPI2HLP.SYS	A8.SYS	1105	0.4945	0.4396	169	165
DBLBUFF.SYS	A9.SYS	2614	1.1538	1.1538	519	215
CCPORT.SYS	A10.SYS	31680	13.4615	13.4615	20163	255

Figure 2.4.1.4.1 establishes the graphical relationship between the source size and the decryption time for *SYS* files. It indicates that here also there exists a linear relationship between these two parameters. Therefore the decryption time increase linearly with the source file size.



**Figure 2.4.1.4.1  
Graph to establish Relationship between Source Size and Decryption Time  
(For *SYS* Files)**

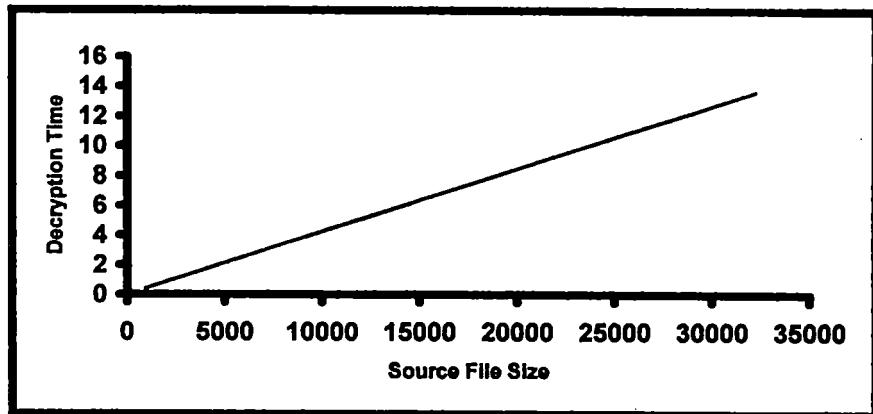
#### 2.4.1.5 Result for *CPP* Files

Table 2.4.1.5.1 summarizes the report of running the technique on 10 sample *CPP* files. Ten cpp files are taken. The size of the files varies from 4071 bytes to 14557 bytes. The encryption time varies from 0.5495 seconds to 13.6813 seconds, whereas the decryption time also varies from 0.5495 seconds to 13.6264 seconds. The values of the Chi Square test results vary from 415 to 101472 with the degree of freedom vary from 248 to 255. From these Chi Square values a high degree of non-homogeneity of each encrypted file is seen in comparison to the corresponding source file.

**Table 2.4.1.5.1  
Result for *CPP* Files**

Source File	Encrypted File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	Chi Square Value	Degree of Freedom
<i>BRICKS.CPP</i>	<i>A1.CPP</i>	16723	7.1429	7.0879	26406	88
<i>PROJECT.CPP</i>	<i>A2.CPP</i>	32150	13.6813	13.6264	101472	90
<i>ARITH.CPP</i>	<i>A3.CPP</i>	9558	4.1209	4.0659	14157	77
<i>START.CPP</i>	<i>A4.CPP</i>	14557	6.2088	6.1538	36849	88
<i>CHARTCOM.CPP</i>	<i>A5.CPP</i>	14080	5.9890	5.9890	35754	84
<i>BITIO.CPP</i>	<i>A6.CPP</i>	4071	1.7582	1.7582	2628	70
<i>MAINC.CPP</i>	<i>A7.CPP</i>	4663	1.9780	1.9780	2380	83
<i>TTEST.CPP</i>	<i>A8.CPP</i>	1257	0.5495	0.5495	415	69
<i>DO.CPP</i>	<i>A9.CPP</i>	14481	6.1538	6.1538	31551	88
<i>CAL.CPP</i>	<i>A10.CPP</i>	9540	4.0659	4.0659	13499	77

Figure 2.4.1.5.1 establishes a graphical relationship between the source file size and the decryption time for these *CPP* files to draw the conclusion that here also there exists an almost linear relationship between these two values. Therefore for *CPP* files also the decryption time increases linearly with the source file size.



**Figure 2.4.1.5.1**

**Graph to establish Relationship between Source Size and Decryption Time  
(For CPP Files)**

#### 2.4.1.6 Report on Variation of Encryption Time with Varying Block Sizes

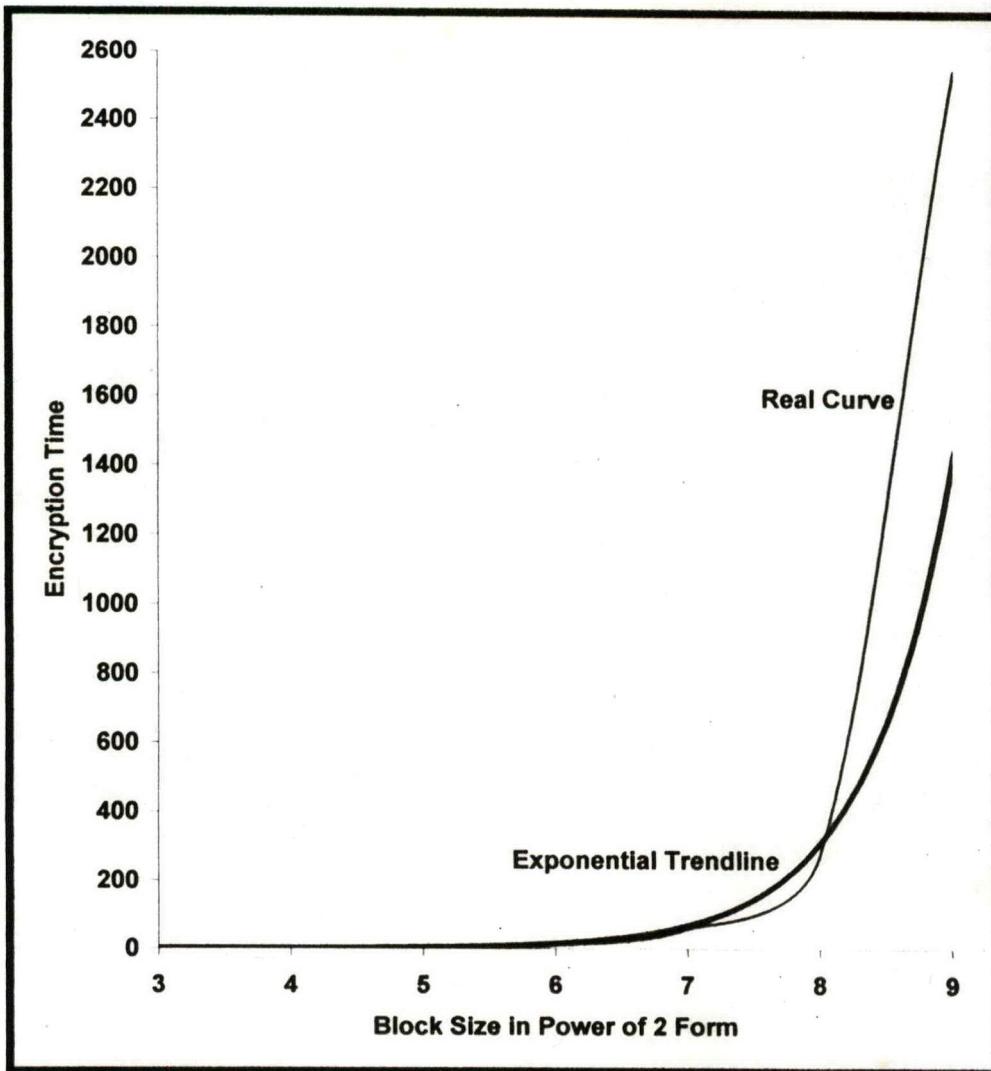
For the purpose of observing the change in the encryption time with the change in the block size, one sample file CMD640X2.SYS has been considered. Table 2.4.1.6.1 shows the result.

If the unique block size of 8 bits is considered, the encryption time is observed as 0.3297 seconds. For the unique block size of 16 bits, the observed encryption time is 0.5495 seconds. The encryption time is observed as 2.7473 seconds for the unique block size of 32 bits. If the unique block length is considered as 64 bits, the encryption time is observed as 8.9011 seconds. For the block size of 128 bits, the encryption time is observed as 57.5824 seconds. If the block size is taken as 256 bits, the encryption time is seen as 276.8681 seconds. Finally, for the unique block size of 512 bits, the encryption time for encrypting the file CMD640X2.SYS is observed as 2545.9338 seconds.

**Table 2.4.1.6.1**  
**Encryption Time for Different Block Sizes for CMD640X2.SYS**

Block Size	Encryption Time
8	0.3297
16	0.5495
32	2.7473
64	8.9011
128	57.5824
256	276.8681
512	2545.9338

Figure 2.4.1.6.1 illustrates graphically how the change in the encryption time is related to the block size. In this figure, the term “Real Curve” indicates the curve denoting the relationship between the source size in power of 2 form and the encryption time for the sample file considered, and the term “Exponential Trendline” indicates the typical exponential curve. The analogy between these two curves establishes the fact that there exists a tendency of the exponential change in the encryption time with  $N$ , where  $2^N$  is the unique block size considered.



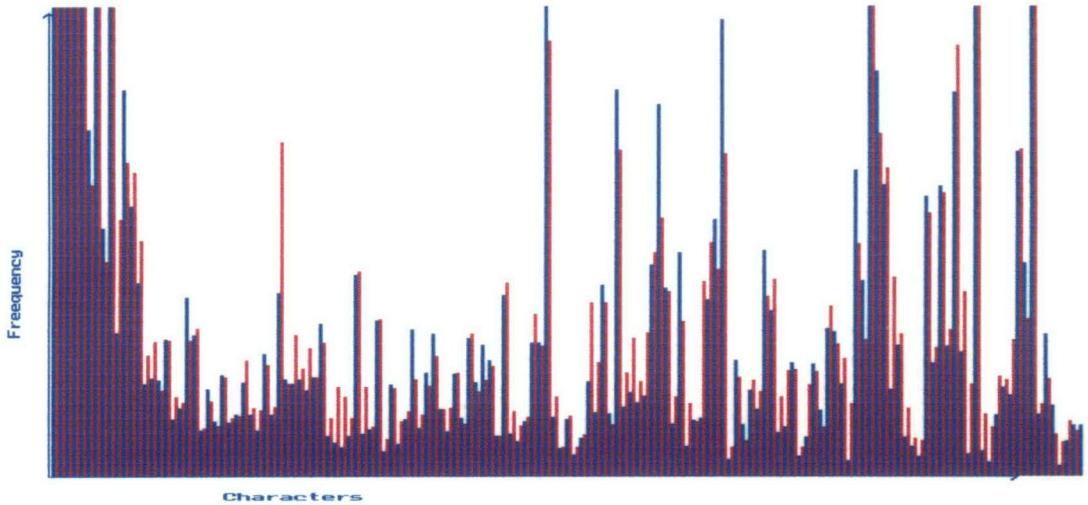
**Figure 2.4.1.6.1**  
**Tendency of Encryption Time to Change Exponentially with N, for Block Size of  $2^N$**   
**(Result Observed for CMD640X2.SYS)**

#### 2.4.2 Result for Frequency Distribution Tests

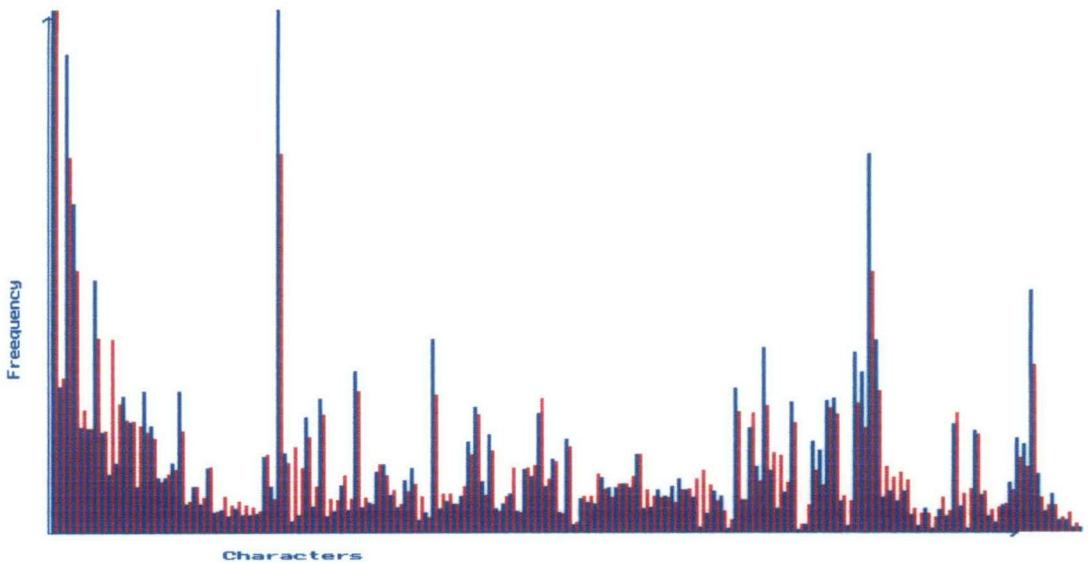
The frequency of each of the 255 characters in the source file and the same in the encrypted file were calculated and compared to assess the efficiency of the proposed technique. All 50 sample files are considered for this purpose and the unique block size of 64 bits is considered while encrypting the files. Figure 2.4.2.1 to figure 2.4.2.5 show segments of graphical outcome only for arbitrarily chosen 5 files of different categories.

In each figure, red lines show the frequencies represented by the characters of the encrypted file whereas blue lines represent the frequencies of the characters of the source

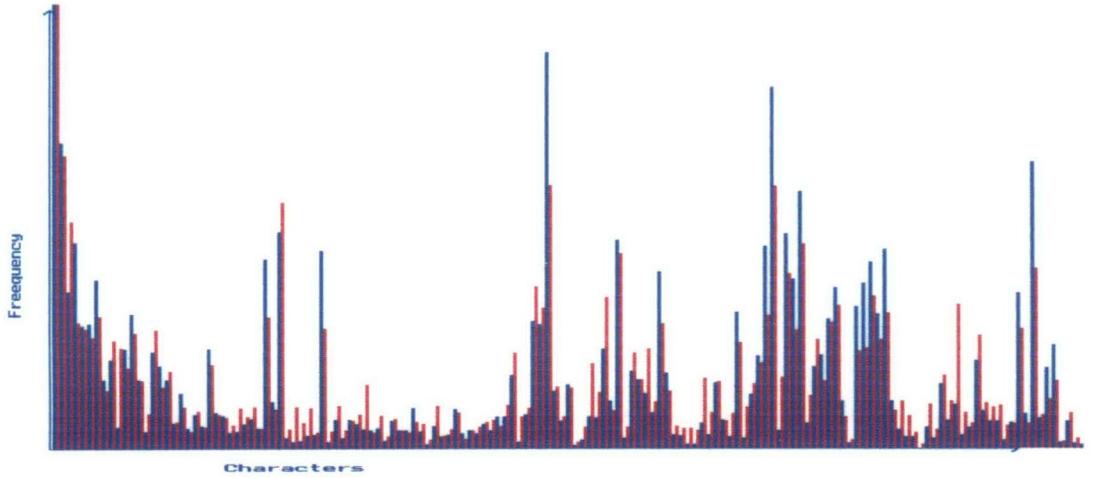
file. From these figures it is very much clear that the source and the corresponding encrypted files are heterogeneous in nature. This can be interpreted that the proposed technique obtains a good quality of encryption.



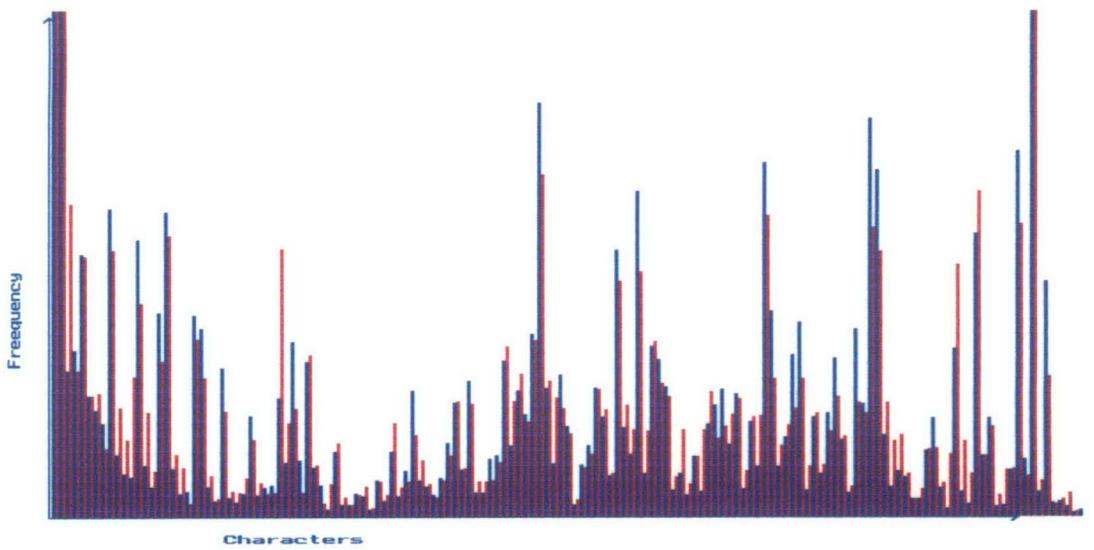
**Figure 2.4.2.1**  
**Frequency Distribution Chart for**  
**RPPO.EXE and Encrypted FOX1.EXE**



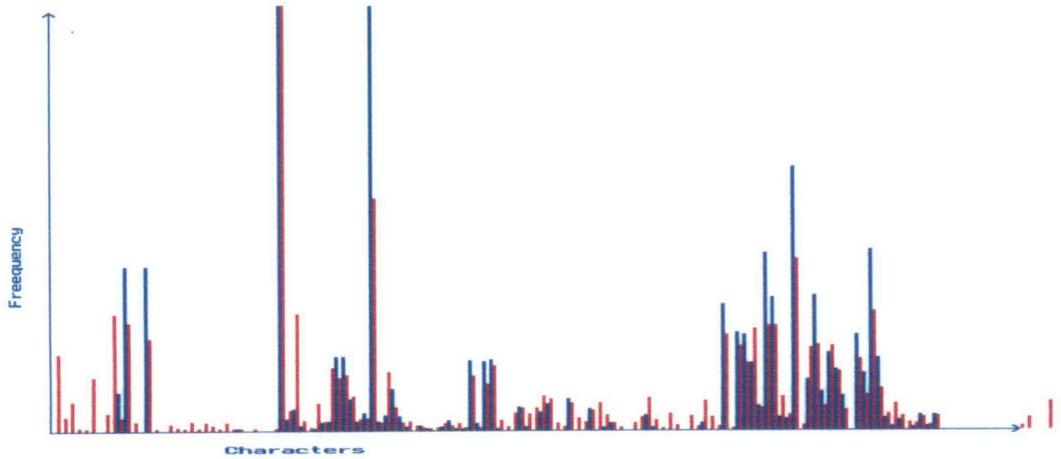
**Figure 2.4.2.2**  
**Frequency Distribution Chart for**  
**DOSKEY.COM and Encrypted FOX1.COM**



**Figure 2.4.2.3**  
**Frequency Distribution Chart for**  
**NDDEAPI.DLL and Encrypted FOX1.DLL**



**Figure 2.4.2.4**  
**Frequency Distribution Chart for**  
**USBD.SYS and Encrypted FOX1.SYS**



**Figure 2.4.2.4**  
**Frequency Distribution Chart for**  
**BITIO.CPP and Encrypted FOX1.CPP**

### 2.4.3 Comparison with RSA Technique

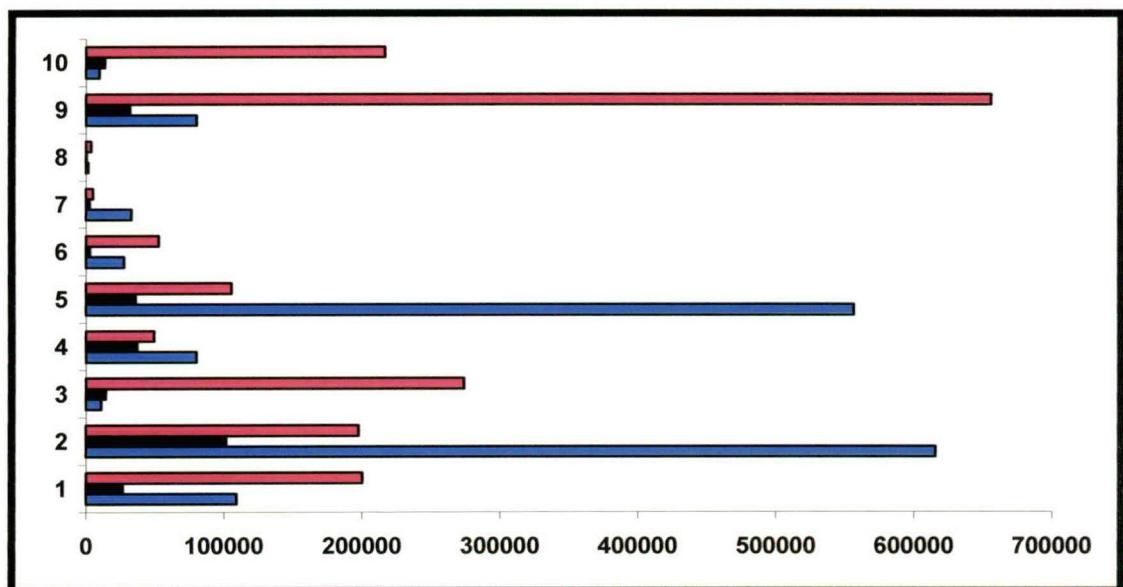
Chi square values between the sample .CPP files and corresponding three encrypted files, first using the RPSP technique with the unique block length of only 8 bits, second using the same with the unique block length of 64 bits, and the third using the existing RSA technique, are compared and the results are given in table 2.4.3.1 [7, 8].

When the proposed RPSP technique is implemented with 8-bit blocks, Chi Square values range from 1471 to 615796, and the same when is implemented for 64-bit blocks, Chi Square values range from 415 to 101472. On the other hand, if the same files are encrypted using the RSA technique, Chi Square values range from 3652 to 655734. In each case, the degree of freedom ranges from 69 to 90. From the table, it is also observed that the proposed technique produces the higher Chi Square values than the RSA technique for PROJECT.CPP (using 8-bit block), START.CPP (using 8-bit block), CHARTCOM.CPP (using 8-bit block), and MAINC.CPP (for 8-bit block).

**Table 2.4.3.1**  
**Comparison between Proposed RPSP and Existing RSA**

Source File	Chi Square Values			Degree of Freedom
	For RPSP (8-bit Block)	For RPSP (64-bit Block)	For RSA	
<i>BRICKS.CPP</i>	109186	26406	200221	88
<i>PROJECT.CPP</i>	615796	101472	197728	90
<i>ARITH.CPP</i>	10842	14157	273982	77
<i>START.CPP</i>	80174	36849	49242	88
<i>CHARTCOM.CPP</i>	556552	35754	105384	84
<i>BITIO.CPP</i>	27462	2628	52529	70
<i>MAINC.CPP</i>	32724	2380	4964	83
<i>TTEST.CPP</i>	1471	415	3652	69
<i>DO.CPP</i>	80098	31551	655734	88
<i>CAL.CPP</i>	9540	13499	216498	77

Figure 2.4.3.1 is the diagrammatic representation of the comparison shown in table 2.4.3.1, where “red lines” stand for the RSA, “black lines” stand for the RPSP with 64-bit blocks, and “blue lines” stands for the RPSP with 8-bit blocks.



**Figure 2.4.3.1**  
**Graphical Comparison of Results of Chi Square Tests among Proposed RPSP (8-bit Block, 64-bit Block) and Existing RSA**

## 2.5 Analysis

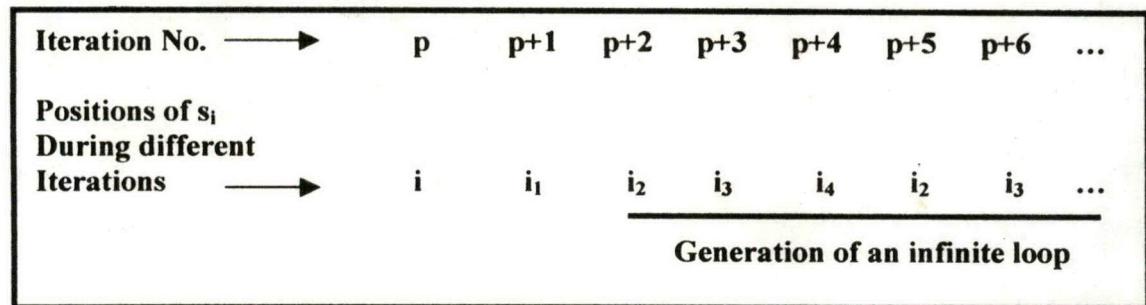
This section consists of analyses on three issues. One is the issue related to block size, discussed in section 2.5.2. Another issue is related to different factors that were considered to evaluate the technique, discussed in section 2.5.3. The last issue is the key of the technique, which is analyzed in chapter 8, a special chapter on the key generation. Prior to all these issues, the proof on the finiteness of the series of iterations has also been done in section 2.5.1.

### 2.5.1 Proof of Cycle Formation

The RPSP technique will not work if a source block is never regenerated. Now, the regeneration of a source block  $S = s_0 s_1 s_2 s_3 s_4 \dots s_{n-1}$  means re-occupancy of the original positions by all the  $n$  bits of  $S$ . This re-occupancy is to be made after some positional reorientation of all the  $n$  bits in  $S$ . Now, if one or more bits never reoccupy their original positions, the source block will not be regenerated and hence the cycle will not be completed [3, 29, 30].

Now, it is to be proved that any bit  $s_i$  ( $0 \leq i \leq n-1$ ) comes back into its original position ( $i$ ) after a finite number of iterations.

Let, if possible, a bit  $s_i$  cannot reach its original position because it is stuck inside a loop shown in figure 2.5.1.1.



**Figure 2.5.1.1**  
**Generation of Infinite Loop (if possible) during Formation of Cycle**

So, effectively what is happening here is two different bits, including  $s_i$ , are attempting to occupy the position  $i_2$  during iteration no. ( $p+2$ ); one from the position  $i_1$ , which is the bit  $s_i$ , and the other from the position  $i_4$ . Effectively the situation is like figure 2.5.1.2.

Iteration No.	p	p+1	p+2	p+3	p+4	p+5	p+6	...
Positions of $s_i$ During different Iterations	i	$i_1$	$i_2$	$i_3$	$i_4$	$i_2$	$i_3$	...
The bit of Position $i_4$ After Iteration (p+1)		$i_1 \rightarrow$	$i_2$	$i_3$	$i_4$	$i_2$	$i_3$	...

**Generation of an infinite loop**

**Figure 2.5.1.2**  
**Two bits attempting to occupy the same position (if possible)**

But as per the scheme concerned, with respect to blocks of a fixed size, for each and every position in the intermediate and the final blocks there exists a unique position in the previous block, so that no two bits of two positions can be transferred to a single position. Therefore our assumption is contradicting the basic philosophy of the scheme. Hence our assumption is wrong.

So, it can be concluded that the process of regenerating the source block is absolutely a finite process.

## 2.5.2 Analysis on Block Size

To enhance the security, one criterion should be to choose the block size such a manner that it requires a huge number of iterations to complete the cycle. Now, generally the number of such iterations increases as the block size increases, but there exist many exceptions also in this regard. Table 2.5.2.1 shows how the number of such iterations changes with changes in block size [31, 32, 34].

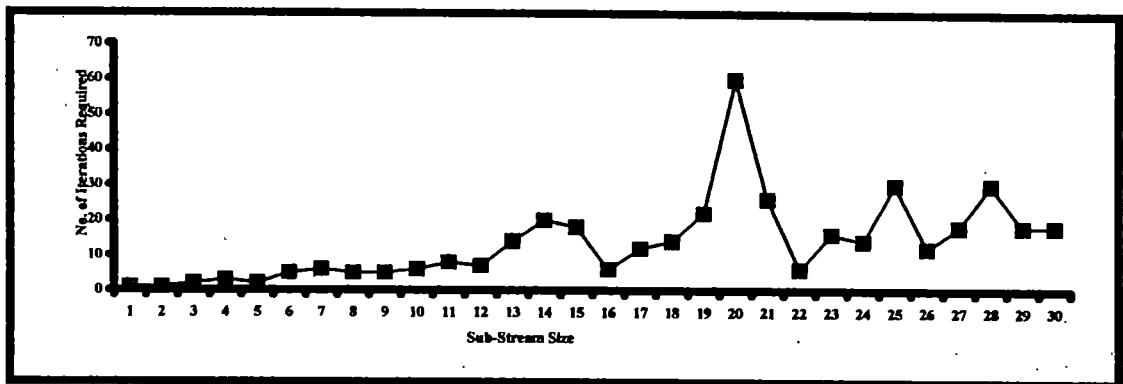
**Table 2.5.2.1**  
**No. of Iterations (I) Required to form Cycles for Blocks of different Sizes (n)**

n	I	n	I	n	I	n	I	n	I
1	1	7	6	13	14	19	22	25	30
2	1	8	5	14	20	20	60	26	12
3	2	9	5	15	18	21	26	27	18
4	3	10	6	16	6	22	6	28	30
5	5	11	8	17	12	23	16	29	18
6	5	12	7	18	14	24	14	30	18

The graph shown in figure 2.5.2.1 to represent table 2.5.2.1 depicts the clear picture in this regard. In this graph, block sizes only in the range of 1 to 30 have been considered. The most satisfactory result appears when the size of a block is taken as 20 because it requires a total of 64 iterations to complete the cycle.

But in practical case it is suggested to take the block size as of  $2^n$  ( $n = 1, 2, 3, \dots$ ) form, preferably of at least 64-bit size. In that case we can observe a steady increase in the number of iterations to complete the cycle, as the value of  $n$  increases. Table 2.5.2.2 gives this information in a tabular presentation and figure 2.5.2.2 shows this relationship graphically.

Figure 2.5.2.1 shows that there exists a non-linear relationship between the number of iterations required to complete the cycle and the unique block size if the block size ranges from 1 to 30.



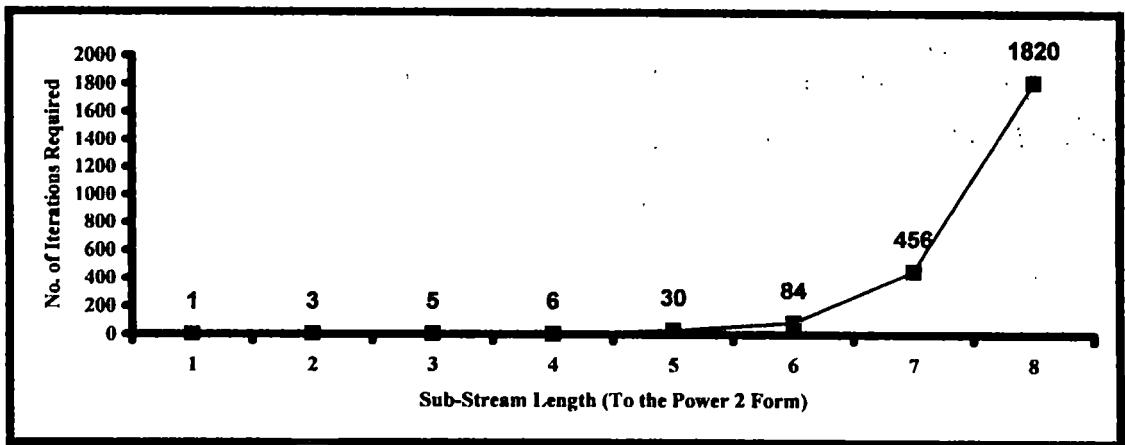
**Figure 2.5.2.1**  
**Relationship between Block Size and Number of Cycles to Form a Cycle**

Table 2.5.2.2 shows a steady increment in the number of iterations to complete the cycle if the block size is considered in the form of  $2^n$ . It is seen that for the values of  $n$  ranging from 0 to 8, numbers of iterations required are respectively 1, 1, 3, 5, 6, 30, 84, 456, and 1820.

**Table 2.5.2.2  
No. of Iterations (I) Required to form Cycles for Blocks of different Sizes ( $2^n$ )**

<b>n</b>	<b>I</b>	<b>n</b>	<b>I</b>	<b>n</b>	<b>I</b>
0	1	3	5	6	84
1	1	4	6	7	456
2	3	5	30	8	1820

The steady increment in the number of iterations required completing the cycle is shown pictorially in figure 2.5.2.2.



**Figure 2.5.2.2  
Graphical Relationship between Number of Iterations (I) Required to form Cycles for Blocks of different Sizes ( $2^n$ )**

Now, the RPSP technique may be implemented in an intelligent way by making the blocks to be of different sizes. In that case, different blocks will require different number of iterations to complete the cycle. Naturally the total number of iterations to regenerate the entire stream of bits also will be different.

In this regard, let us consider a simple example. Say, there is a stream of bits of size 128 bits. Using this intelligent approach, the stream is being decomposed into blocks  $B_1$  (64 bits),  $B_2$  (20 bits),  $B_3$  (19 bits),  $B_4$  (13 bits) and  $B_5$  (12 bits). Now, following table 2.5.2.1 and table 2.5.2.2, we get the information that  $B_1$  requires 84 iterations,  $B_2$  requires

60 iterations,  $B_3$  requires 22 iterations,  $B_4$  requires 14 iterations and  $B_5$  requires 7 iterations. Therefore to regenerate the entire stream of bits, the minimum number of iterations required is 4620, which is obtained by taking the LCM of 84, 60, 22, 14 and 7; whereas a fixed block size of 64 bits would have required only 84 iterations to regenerate the entire stream. Thus by allowing blocks to be of varying sizes; a far better security can be achieved.

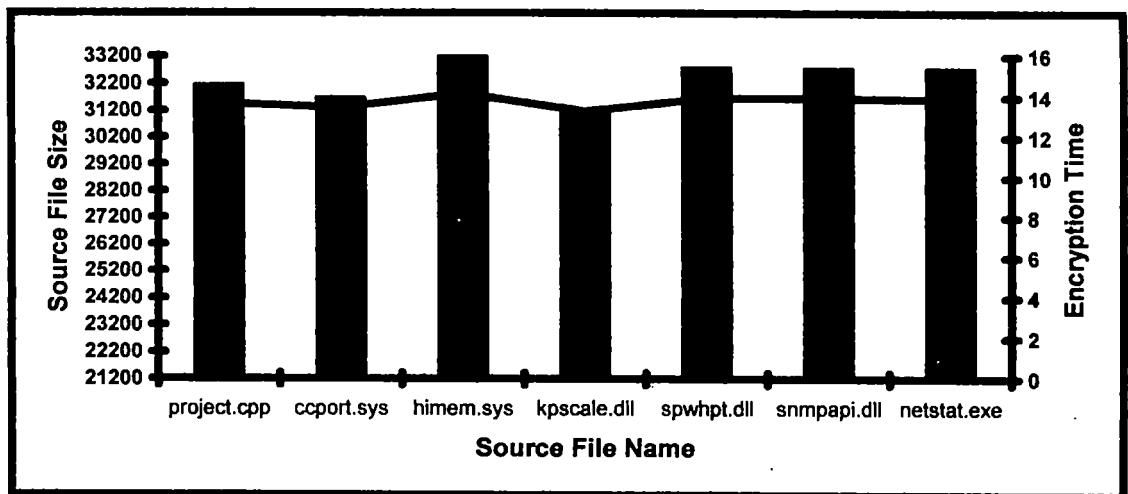
### **2.5.3 Analysis on Factors Considered for Evaluation Purpose**

From the results shown in section 2.4, it is clear that for a set of fixed blocks, the execution time (the encryption as well as the decryption) varies almost linearly with the source file size. Now, since any type of source file is being considered simply as a stream of bits, the encryption (or the decryption) time does not depend on the type of the source file. Hence if the technique is applied for different types of files, all being of around the same size, the encryption (or the decryption) time will be more or less the same for a particular block size or for a particular set of different blocks. Table 2.5.3.1 establishes this fact and it is graphically shown in figure 2.5.3.1.

In table 2.5.3.1, seven files of almost the same size ranging from 31232 bytes to 33191 bytes have been considered. It is observed that using the proposed RPSP technique if these files are encrypted, the encryption time ranges from 13.2967 seconds to 14.1758 seconds, and the decryption time ranges from 13.2967 seconds to 14.1209 seconds. This means that for these files of almost similar sizes, the encryption/decryption times are also almost similar. Graphically this fact is established in figure 2.5.3.1

**Table 2.5.3.1  
Result of Encryption/Decryption Time for  
Different Types of Files of Almost Same Sizes**

<b>File Name</b>	<b>File Size (In Bytes)</b>	<b>Encryption Time (In Seconds)</b>	<b>Decryption Time (In seconds)</b>
<i>PROJECT.CPP</i>	32150	13.6813	13.6264
<i>CCPORT.SYS</i>	31680	13.4615	13.4615
<i>HIMEM.SYS</i>	33191	14.1758	14.1209
<i>KPSCALE.DLL</i>	31232	13.2967	13.2967
<i>SPWHPT.DLL</i>	32792	13.9560	13.9560
<i>SNMPAPI.DLL</i>	32768	13.9560	13.9011
<i>NETSTAT.EXE</i>	32768	13.9011	13.9011



**Figure 2.5.3.1**  
**Graphical Relationship among Encryption Times of**  
**Files of Different Categories but almost of Same Sizes**

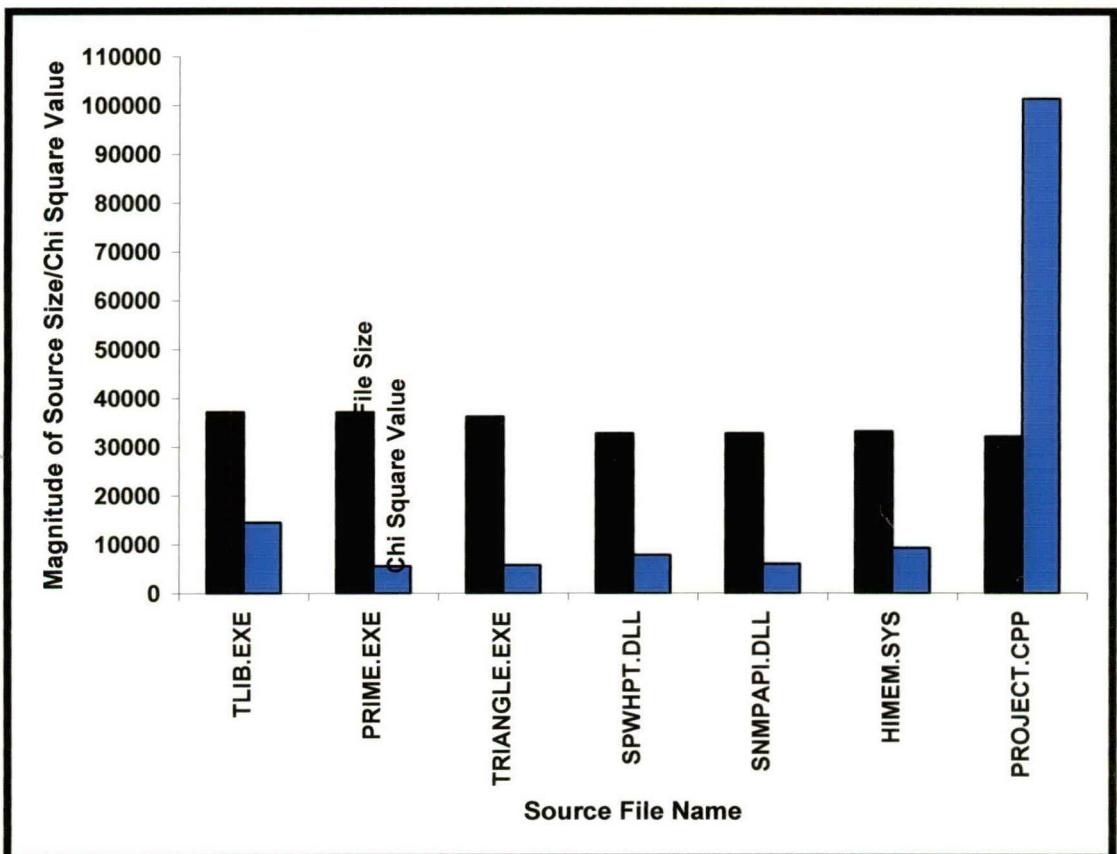
Results obtained in section 2.4 on chi square values suggest the fact it is exactly file-dependent since there exists no fixed relationship between the file size and the chi square value. Out of all the results obtained, only those for almost the same file sizes have been considered to analyze their chi square values. Table 2.5.3.2 enlists these results and the graphical relationship is shown in figure 2.5.3.2.

In table 2.5.3.2, seven files have been considered, the sizes ranging from 32150 bytes to 37220 bytes. Five out of these files are with the degree of freedom as 255, one with 253, and the remaining with 90. For these sample files of almost similar sizes, Chi Square values range from 5485 to 14479, which means that in spite of the fact the files considered are of almost similar sizes, there Chi Square values vary to a large extend. It indicates that the Chi Square value is dependent to the content of the file, not to the size of the file.

In figure 2.5.3.2, the black pillars, which are almost of the similar heights, stand for the sizes of sample files, and the corresponding adjacent blue pillars, the heights of which vary to a large extend, stand for the corresponding Chi Square values.

**Table 2.5.3.2**  
**Result of Chi Square Values for**  
**Different Types of Files of Almost Same Sizes**

File Name	File Size	Chi Square Value	Degree of Freedom
<i>TLIB.EXE</i>	37220	14479	255
<i>PRIME.EXE</i>	37152	5485	255
<i>TRIANGLE.EXE</i>	36242	5690	255
<i>SPWHPT.DLL</i>	32792	7781	255
<i>SNMPAPI.DLL</i>	32768	5987	253
<i>HIMEM.SYS</i>	33191	9189	255
<i>PROJECT.CPP</i>	32150	101472	90



**Figure 2.5.3.2**  
**Graphical Relationship between Chi Square Values of**  
**Files of Different Categories but almost of Same Sizes**

Results of the frequency distribution tests presented in section 2.4.2 suggest the existence of a wide range of distribution of characters in all encrypted files, which is so effective in making it difficult to break the ciphertext using cryptanalysis.

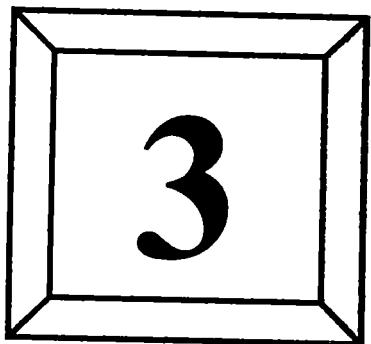
One point to be noted in section 2.4.3 is that the better result is available in terms of the chi square values for lower block size. But only judging from this angle it cannot be concluded that it is a preferable option to choose smaller blocks, since having a unique size of blocks of only 8 bits makes the implementation of the encryption process so easy.

## 2.6 Conclusion

Analyzing the proposed RPSP technique from different perspectives, it can be pointed out that it is the structure of the scheme that helps in forming a large key space, which in turn helps ensuring the security of a very satisfactory level. If blocks of varying sizes are constructed from a source stream of bits, and for each block arbitrary number of iterations are performed during the process of encryption, and if all these information are accommodated in the secret key, then for ensuring the correct decryption, a reasonably long key space is required. One proposal for such a key is presented in figure 8.2.1.1 in chapter 8.

Also, section 2.5.1 shows the finiteness in regenerating the source block, which ensures that whatever is the size of a block, after a finite number of iterations it is regenerated, and since this finite number of iterations does not follow any mathematical policy it ensures a better security.

Therefore it can be concluded that the RPSP encryption policy itself can produce a satisfactory degree of information security, and, as it is discussed in chapter 9, when it participates in the cascaded approach of encryption, it further enhances the performance a lot.



# **Encryption Through Triangular Encryption (TE) Technique**

<b><u>Contents</u></b>	<b><u>Pages</u></b>
<b>3.1 Introduction</b>	<b>90</b>
<b>3.2 The Scheme</b>	<b>91</b>
<b>3.3 Implementation</b>	<b>98</b>
<b>3.4 Results</b>	<b>116</b>
<b>3.5 Analysis and Conclusion including Comparison with RPSP</b>	<b>129</b>

### **3.1 Introduction**

In chapter 2, the RPSP technique has been discussed in detail. In this chapter, the Triangular Encryption (TE) Technique is proposed.

The nomenclature followed for this proposed technique, Triangular Encryption (TE) Technique, is on the basis of the structure that is supposed to be formed out of the source as well as different intermediate and final blocks of bits during the process of encryption. In fact, an equilateral triangular shape is formed if the source block of bits and different intermediate blocks along with the final 1-bit block are shown in line-by-line manner.

Now, the basic characteristic of this TE technique, in which it is entirely different from the RPSP technique, discussed in chapter 2, is the use of the Boolean operation. Also, unlike the RPSP technique, here no attempt is made for the formation of a cycle. Another important aspect of this TE technique is that here there is no question of the positional reorientation of bits; rather all the bits in a block are directly participating in a Boolean operation.

Since, like all the other proposed techniques, this TE technique is also a bit-level technique, the stream of bits corresponding to the source file to be encrypted is to be decomposed into a finite number of blocks that are advised to be of varying lengths. For each of the blocks, the technique of TE is to be applied to generate the corresponding target block.

Now, in the Triangular Encryption (TE) technique, from a source block of size, say,  $n$ , an intermediate block of size  $(n-1)$  is generated by applying the exclusive OR (XOR) operation between each two consecutive bits. The same process is again applied to the generated block of size  $(N-1)$  to generate a block of size  $(n-2)$ . The process goes on until the generation of a 1-bit block. All these blocks under consideration together form an equilateral triangle-like shape.

After the formation of such a triangular shape, putting together either the MSBs or the LSBs of all the blocks under consideration in either sequence, the target block is formed. In this regard, the key takes a vital role because only by knowing this key the receiver of the message can understand on how the target block is chosen from the triangular shape. Obviously, this key is to be kept secret. So it is a secret key system.

The encrypted stream of bits is generated by putting together all the target blocks [2, 48, 57].

Section 3.2 of this chapter discusses the scheme followed in this technique through algorithmic presentation as well as diagrammatic representation. An implementation of this technique is described in section 3.3. Section 3.4 shows results after implementing this technique in a certain manner on the same set of files that was also considered for the RPSP technique in chapter 2. An analytical view and the concluding remark on this technique are drawn in section 3.5.

### 3.2 The Scheme

The plaintext to be transmitted is to be converted into a stream of bits. Since the TE technique, like all the other proposed techniques, is a bit-level technique, here also the source stream is to be decomposed into a finite number of blocks, not necessarily of the same length. In fact, as it will be analyzed in section 3.5, an enhancement of security can be done by allowing block sizes to be different because it makes the key length large enough, hereby almost nullifying the chance of cryptanalysis to break the cipher [48].

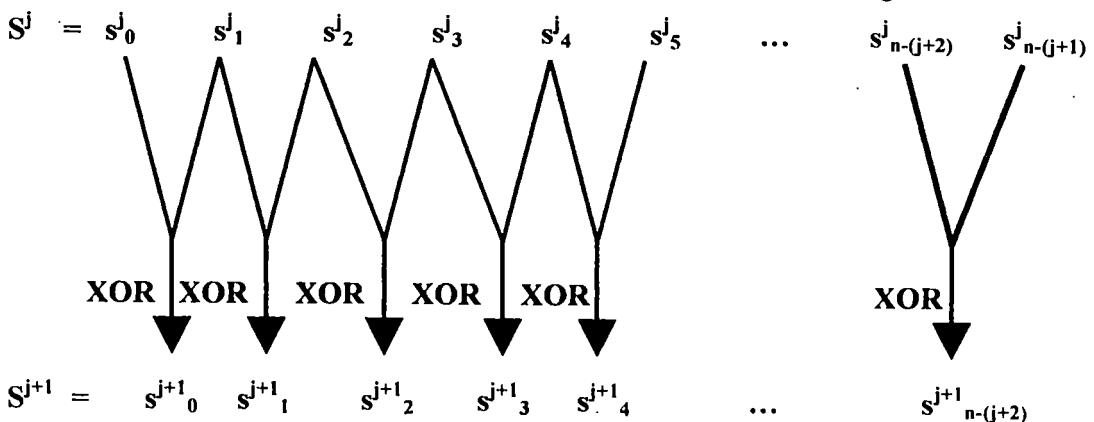
The entire scheme includes several parts in it. First of all is the formation of the triangle, which is shown in section 3.2.1. Section 3.2.2 discusses different options that are available to form the target block from the triangle generated in section 3.2.1. Section 3.2.3 gives the processes for decryption to generate the source block from a target block. Section 3.2.4 gives a simple example to illustrate the scheme.

#### 3.2.1 Formation of Triangle

Consider a block  $S = s_0^0 s_1^0 s_2^0 s_3^0 s_4^0 s_5^0 \dots s_{n-2}^0 s_{n-1}^0$  of size  $n$  bits, where  $s_i^0 = 0$  or 1 for  $0 \leq i \leq (n-1)$ .

Starting from MSB ( $s_0^0$ ) and the next-to-MSB ( $s_1^0$ ), bits are pair-wise XORed, so that the 1<sup>st</sup> intermediate sub-stream  $S^1 = s_0^1 s_1^1 s_2^1 s_3^1 s_4^1 s_5^1 \dots s_{n-2}^1$  is generated consisting of  $(n-1)$  bits, where  $s_j^1 = s_j^0 \oplus s_{j+1}^0$  for  $0 \leq j \leq n-2$ ,  $\oplus$  stands for the exclusive OR operation. This 1<sup>st</sup> intermediate sub-stream  $S^1$  is also then pair-wise XORed to generate  $S^2 = s_0^2 s_1^2 s_2^2 s_3^2 s_4^2 s_5^2 \dots s_{n-3}^2$ , which is the 2<sup>nd</sup> intermediate sub-stream of length  $(n-2)$ . This process continues  $(n-1)$  times to ultimately generate  $S^{n-1} = s_{n-1}^{n-1}$ ,

which is a single bit only. Thus the size of the 1<sup>st</sup> intermediate sub-stream is one bit less than the source sub-stream; the size of each of the intermediate sub-streams starting from the 2<sup>nd</sup> one is one bit less than that of the sub-stream wherefrom it was generated; and finally the size of the final sub-stream in the process is one bit less than the final intermediate sub-stream. Figure 3.2.1.1 shows the generation of an intermediate sub-stream  $S^{j+1} = s^{j+1}_0 s^{j+1}_1 s^{j+1}_2 s^{j+1}_3 s^{j+1}_4 s^{j+1}_5 \dots s^{j+1}_{n-(j+2)}$  from the previous intermediate sub-stream  $S^j = s^j_0 s^j_1 s^j_2 s^j_3 s^j_4 s^j_5 \dots s^j_{n-(j+1)}$ . The formation of the triangular shape for the source sub-stream  $S = s^0_0 s^0_1 s^0_2 s^0_3 s^0_4 s^0_5 \dots s^0_{n-2} s^0_{n-1}$  is shown in figure 3.2.1.2.



**Generation of an Intermediate Sub-Stream in TE**

**Figure 3.2.1.1**

$$\begin{aligned}
 S &= s^0_0 & s^0_1 & s^0_2 & s^0_3 & s^0_4 & s^0_5 & \dots & s^0_{n-2} & s^0_{n-1} \\
 S^1 &= s^1_0 & s^1_1 & s^1_2 & s^1_3 & s^1_4 & \dots & & s^1_{n-2} \\
 S^2 &= s^2_0 & s^2_1 & s^2_2 & s^2_3 & \dots & & & s^2_{n-3} \\
 S^3 &= s^3_0 & s^3_1 & s^3_2 & \dots & & & & s^3_{n-4} \\
 S^4 &= s^4_0 & s^4_1 & \dots & & & & & s^4_{n-5} \\
 S^5 &= s^5_0 & \dots & & & & & & s^5_{n-6} \\
 && \dots & \dots & \dots & \dots & \dots & \dots &
 \end{aligned}$$

$$\begin{aligned}
 S^{n-2} &= & s^{n-2}_0 & s^{n-2}_1 \\
 S^{n-1} &= & s^{n-1}_0
 \end{aligned}$$

**Formation of a Triangle in TE**

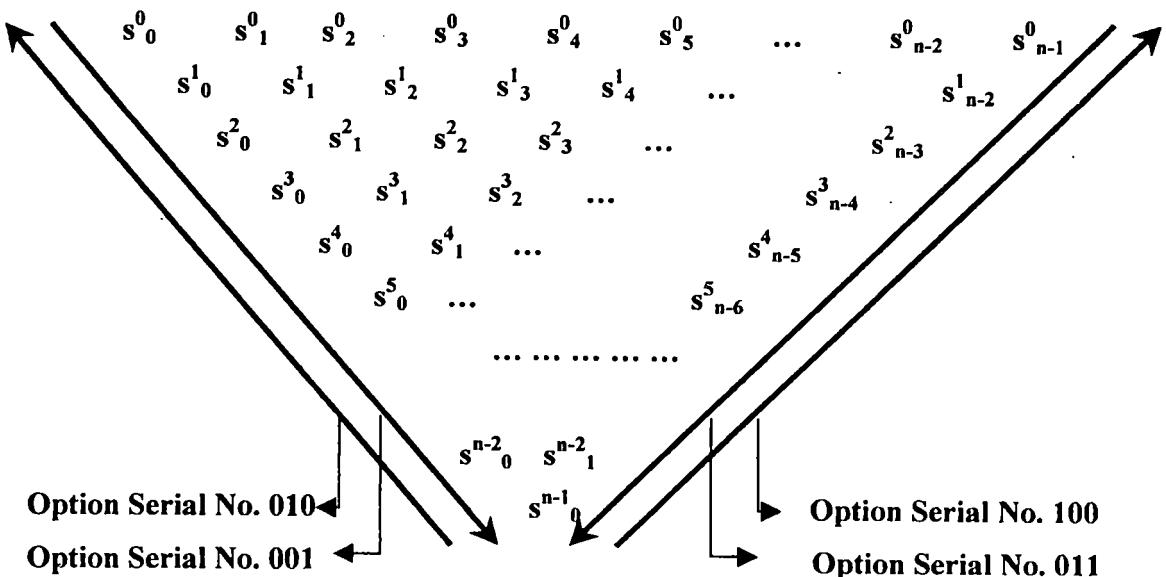
**Figure 3.2.1.2**

### 3.2.2 Options for Forming Target Blocks from Triangle

Corresponding to figure 3.2.1.2, a total of four options are available to form a target block from the source block  $S = s_0^0 s_1^0 s_2^0 s_3^0 s_4^0 s_5^0 \dots s_{n-2}^0 s_{n-1}^0$ . These options are shown in table 3.2.2.1 [48]. Figure 3.2.2.1 shows the same diagrammatically.

**Table 3.2.2.1**  
**Options for choosing Target Block from Triangle**

Option Serial No.	Target Block	Method of Formation
001	$s_0^0 s_1^0 s_2^0 s_3^0 s_4^0 s_5^0 \dots s_{n-2}^0 s_{n-1}^0$	Taking all the MSBs starting from the source block till the last block generated
010	$s_{n-1}^0 s_{n-2}^0 s_{n-3}^0 s_{n-4}^0 s_{n-5}^0 \dots s_0^1 s_0^0$	Taking all the MSBs starting from the last block generated till the source block
011	$s_{n-1}^0 s_{n-2}^1 s_{n-3}^2 s_{n-4}^3 s_{n-5}^4 \dots s_1^{n-2} s_0^{n-1}$	Taking all the LSBs starting from the source block till the last block generated
100	$s_{n-1}^0 s_{n-2}^1 s_{n-3}^2 s_{n-4}^3 s_{n-5}^4 \dots s_1^{n-2} s_0^{n-1}$	Taking all the LSBs starting from the last block generated till the source block



**Figure 3.2.2.1**  
**Diagrammatic Representation of Options for choosing Target Block from Triangle**

### 3.2.3 Generating Source Block from a Target Block

For the purpose of generating the source block from a target block the reference to table 3.2.3.1 is important.

**Table 3.2.3.1**  
**Decryption Reference for Different Target Blocks**

Corresponding Option Serial No.	Decryption Reference	Comment
001	Symmetric	Exactly the same encryption process to be followed
010	Asymmetric	To follow a different technique for decryption
011	Asymmetric	To follow a different technique for decryption
100	Symmetric	Exactly the same encryption process to be followed

As is shown in table 3.2.3.1, corresponding to option 001 and option 100, the processes of generating the source block are the same, which means, forming the triangle and picking bits in the same manner one by one.

Section 3.2.3.1 discusses how the source block is generated from the target block corresponding to the option serial no. 010 and the same for the target block corresponding to the option serial no. 011 is discussed in section 3.2.3.2.

#### 3.2.3.1 Generating Source Block from Target Block $s^{n-1}_0 s^{n-2}_0 s^{n-3}_0 s^{n-4}_0 s^{n-5}_0 \dots s^1_0 s^0_0$ (With Option Serial No. 010)

As shown in table 3.2.2.1, the encrypted sub-stream corresponding to the option serial no. 010 is  $s^{n-1}_0 s^{n-2}_0 s^{n-3}_0 s^{n-4}_0 s^{n-5}_0 \dots s^1_0 s^0_0$ .

To ease the explanation of decryption technique, let us consider,  $e^0_{i-1} = s^{n-i}_0$  for  $1 \leq i \leq n$ , so that the target block corresponding to the option serial no. 2 becomes  $E = e^0_0 e^0_1 e^0_2 e^0_3 e^0_4 \dots e^0_{n-2} e^0_{n-1}$ . Now, following the same approach as mentioned in section 3.2.1, a triangle is to be formed. After the formation of the triangle, for the purpose of decryption, the block  $e^0_{n-1} e^1_{n-2} e^2_{n-3} e^3_{n-4} e^4_{n-5} e^5_{n-6} \dots e^{n-2}_1 e^{n-1}_0$ , i.e., the sub-stream taking all the LSBs of the blocks starting from E to the finally generated 1-bit block  $E^{n-1}$ , are to be taken together and it is to be considered as the decrypted block. Figure 3.2.3.1.1 shows the triangle generated and hence the decrypted block obtained. Here the intermediate blocks are referred to as  $E^1, E^2, \dots, E^{n-2}$  and the final block generated as  $E^{n-1}$ .

$E =$	$e^0_0$	$e^0_1$	$e^0_2$	$e^0_3$	$e^0_4$	$e^0_5$	$\dots$	$e^0_{n-2}$	$e^0_{n-1}$	
$E^1 =$		$e^1_0$	$e^1_1$	$e^1_2$	$e^1_3$	$e^1_4$	$\dots$		$e^1_{n-2}$	
$E^2 =$			$e^2_0$	$e^2_1$	$e^2_2$	$e^2_3$	$\dots$		$e^2_{n-3}$	
$E^3 =$				$e^3_0$	$e^3_1$	$e^3_2$	$\dots$		$e^3_{n-4}$	
$E^4 =$					$e^4_0$	$e^4_1$	$\dots$		$e^4_{n-5}$	
$E^5 =$						$e^5_0$	$\dots$		$e^5_{n-6}$	
							$\dots \dots \dots \dots \dots$			
$E^{n-2} =$						$e^{n-2}_0$	$e^{n-2}_1$			
$E^{n-1} =$							$e^{n-1}_0$			

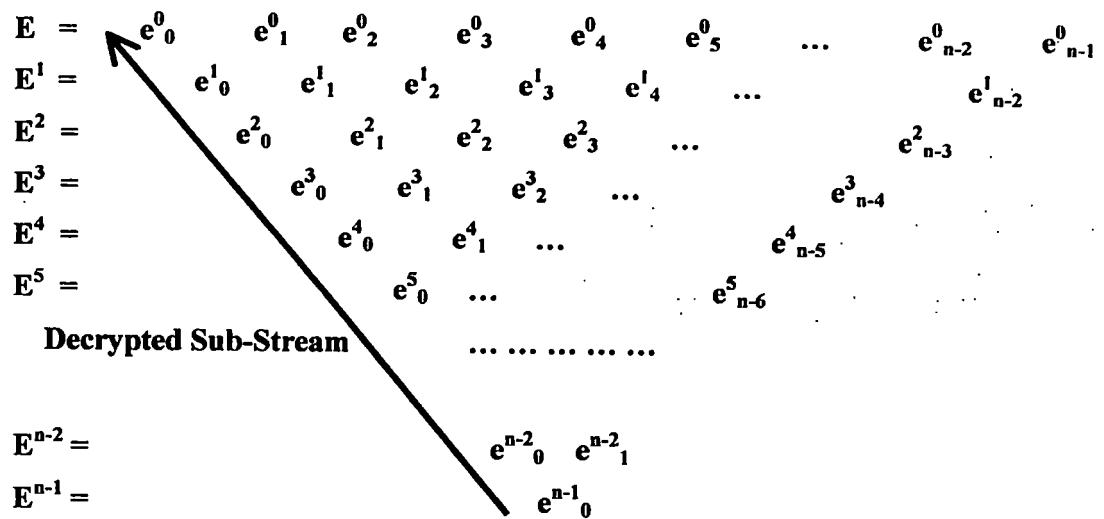
**Figure 3.2.3.1.1**  
**Generation of Source Block from Target Block with Option Serial No. 011**

### 3.2.3.2 Generating Source Block from Target Block

**$s^0_{n-1} s^1_{n-2} s^2_{n-3} s^3_{n-4} s^4_{n-5} \dots s^{n-2}_1 s^{n-1}_0$  (With Option Serial No. 011)**

As shown in table 3.2.2.1, the encrypted block corresponding to the option serial no. 011 is  $s^0_{n-1} s^1_{n-2} s^2_{n-3} s^3_{n-4} s^4_{n-5} \dots s^{n-2}_1 s^{n-1}_0$ .

To ease the explanation of decryption technique, let us consider,  $e^0_{i-1} = s^{i-1}_{n-i}$  for  $1 \leq i \leq n$ , so that the encrypted block becomes  $E = e^0_0 e^0_1 e^0_2 e^0_3 e^0_4 \dots e^0_{n-2} e^0_{n-1}$ . Now, following the same approach as mentioned in section 3.2.1, a triangle is to be formed. After the formation of the triangle, for the purpose of decryption, the block  $e^{n-1}_0 e^{n-2}_0 e^{n-3}_0 e^{n-4}_0 e^{n-5}_0 \dots e^1_0 e^0_0$ , i.e., the block constructed by taking all the MSBs of the blocks starting from the finally generated single-bit block  $E^{n-1}$  to  $E$ , are to be taken together and it is to be considered as the decrypted block. Figure 3.2.3.2.1 shows the triangle generated and hence the decrypted block obtained. Here the intermediate blocks are referred to as  $E^1, E^2, \dots, E^{n-2}$  and the final block generated as  $E^{n-1}$ .



**Figure 3.2.3.2.1**  
**Generation of Source Block from Target Block with Option Serial No. 011**

### 3.2.4 A Sample Example to Illustrate the Scheme

We consider an 8-bit block  $S = 10010101$ . Figure 3.2.4.1 shows the triangle generated using TE.

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

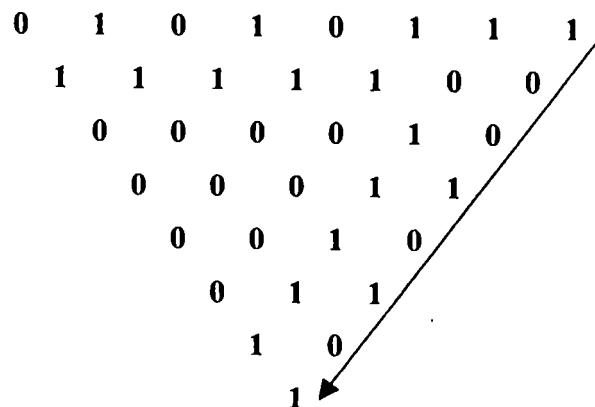
**Figure 3.2.4.1**  
**Formation of Triangle for  $S = 10010101$**

Now, from the triangle shown in figure 3.2.4.1, four types of target blocks, as are shown in table 3.2.4.1, can be generated corresponding to four options mentioned in table 3.2.2.1.

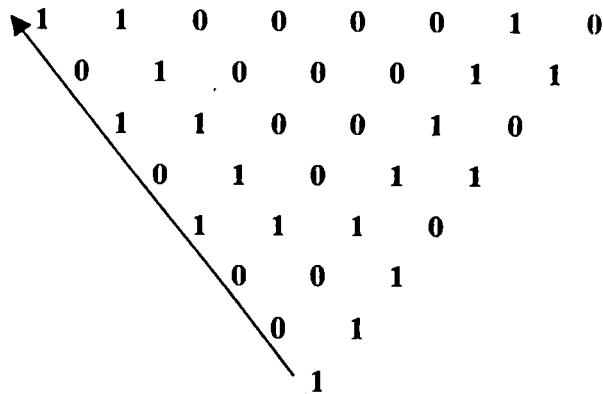
**Table 3.2.4.1**  
**Different Target Blocks generated using TE for S = 10010101**

Source Block S	Target Block Corresponding to Serial No.	Target Block T
10010101	001	11101010
	010	01010111
	011	11000010
	100	01000011

For target blocks  $T_1 = 11101010$  and  $T_4 = 01000011$ , the same approach is to be followed to generate the corresponding source blocks. But blocks  $T_2 = 01010111$  and  $T_3 = 11000010$  require different techniques following sections 3.2.3.1 and 3.2.3.2. Figure 3.2.4.2 and figure 3.2.4.3 respectively show the generations of source blocks from target blocks  $T_2$  and  $T_3$ .



**Figure 3.2.4.2**  
**Generating Source Block S = 10010101 from Target Block T<sub>2</sub> = 01010111**



**Figure 3.2.4.3**  
**Generating Source Block  $S = 10010101$  from Target Block  $T_3 = 11000010$**

### 3.3 Implementation

Consider the following confidential message to be transmitted from one source to its destination point:

**“TERRORIST ATTACK SUSPECTED HOTEL SNOWVIEW DEC 06 ALERT”**

To ease the implementation, say, the following are the rules applied for transmission of such a confidential message:

- The maximum number of characters allowed in the message is 60.
- The maximum length of a block is 32 bits.
- The maximum number of blocks is 20.

In section 8.2.2, in chapter 8, the proposed structure of the corresponding 180-bit key has been proposed.

Table 3.3.1 enlists the considerations to be followed during the process of encryption.

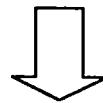
**Table 3.3.1**  
**Different Considerations for Encryption using TE**

Total Number of Blocks: 16							
Size of Block 1	32	Size of Block 5	24	Size of Block 9	32	Size of Block 13	24
Option chosen	001	Option chosen	001	Option chosen	010	Option chosen	001
Size of Block 2	32	Size of Block 6	24	Size of Block 10	32	Size of Block 14	16
Option chosen	001	Option chosen	100	Option chosen	011	Option chosen	100
Size of Block 3	32	Size of Block 7	24	Size of Block 11	32	Size of Block 15	24
Option chosen	100	Option chosen	100	Option chosen	011	Option chosen	100
Size of Block 4	32	Size of Block 8	24	Size of Block 12	16	Size of Block 16	32
Option chosen	010	Option chosen	100	Option chosen	011	Option chosen	100

Following the format of the 180-bit secret key shown in figure 8.2.2.1 in chapter 8, the key will be as shown in figure 3.3.2.

## The Secret Key (Using “/” as Separators)

100000/001/100000/001/100000/100/100000/010/011000/001/011000/100/011000/10  
0/011000/100/100000/010/100000/011/100000/011/010000/011/011000/001/010000/  
100/011000/100/100000/100/000000/000/000000/000/000000/000/000000/000



## The Secret Key (Removing "/")

**Figure 3.3.2**  
**180-bit Secret Key**

Table 3.3.2 converts the message to be transmitted into the corresponding stream of bits. In this table, different characters of the message are given in column wise manner.

**Table 3.3.2**  
**Converting Characters into Bytes**

Character	Byte	Character	Byte	Character	Byte	Character	Byte
T	01010100	K	01001011	E	01000101	<Blank>	00100000
E	01000101	<Blank>	00100000	L	01001100	0	00110000
R	01010010	S	01010011	<Blank>	00100000	6	00110110
R	01010010	U	01010101	S	01010011	<Blank>	00100000
O	01001111	S	01010011	N	01001110	A	01000001
R	01010010	P	01010000	O	01001111	L	01001100
I	01001001	E	01000101	W	01010111	E	01000101
S	01010011	C	01000011	V	01010110	R	01010010
T	01010100	T	01010100	I	01001001	T	01010100
<Blank>	00100000	E	01000101	E	01000101		
A	01000001	D	01000100	W	01010111		
T	01010100	<Blank>	00100000	<Blank>	00100000		
T	01010100	H	01001000	D	01000100		
A	01000001	O	00101111	E	01000101		
C	01000011	T	01010100	C	01000011		

Combining together all the bytes obtained from table 3.3.2, we get the following stream of bits of the length of 432 bits.

01010100/01000101/01010010/01010010/01001111/01010010/01001001/01010011/010111/00100000/01000001/01010100/01010100/01000001/01000011/01001011/001000  
 00/01010011/01010101/01010011/01010000/01000101/01000011/01010100/01000101/  
 01000100/00100000/01001000/00101111/01010100/01000101/01001100/00100000/010  
 10011/01001110/01001111/01010111/01010110/01001001/01000101/01010111/001000  
 00/01000100/01000101/01000011/00100000/00110000/00110110/00100000/01000001/  
 01001100/01000101/01010010/01010100

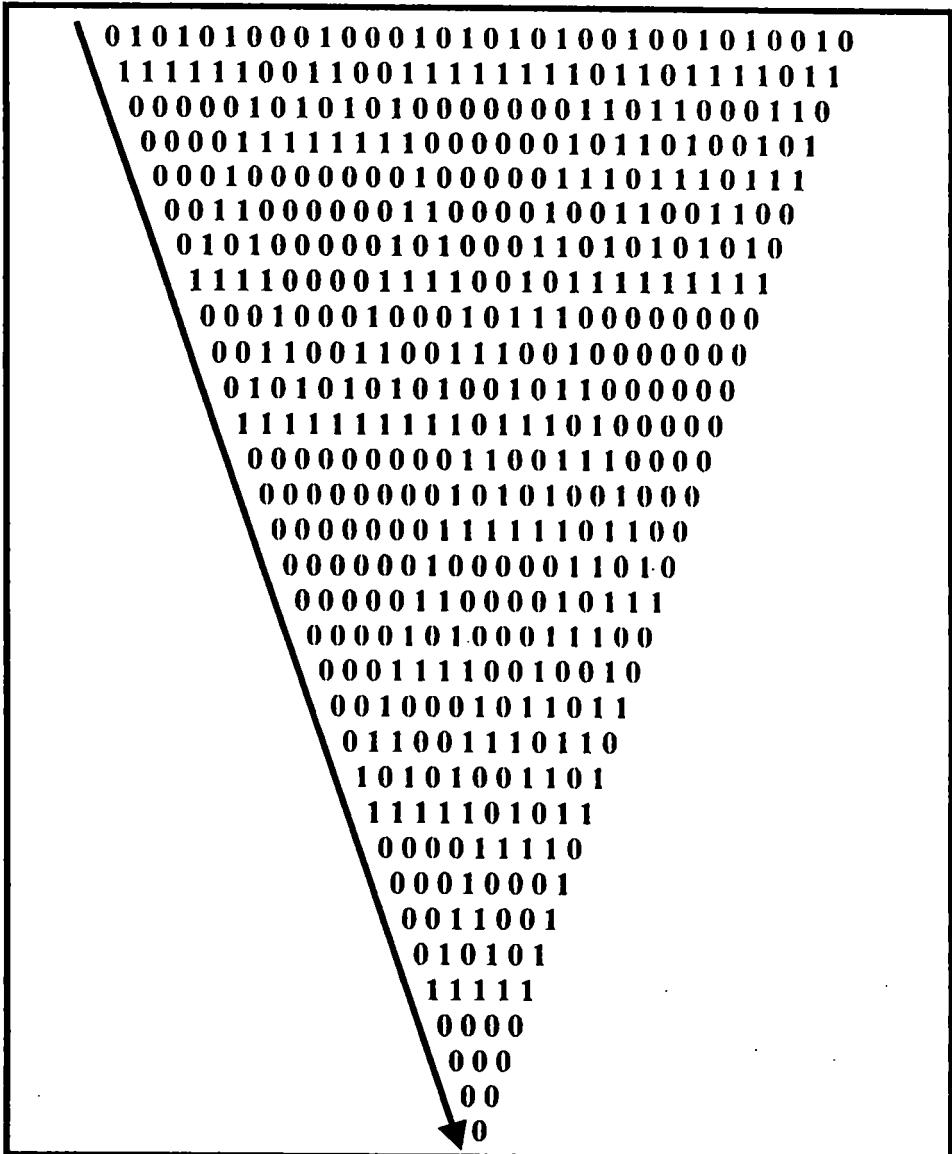
“/” being working as the separator between two consecutive bytes.

Combining the information of table 3.3.1 and table 3.3.2, we obtain table 3.3.3, in which all the blocks to be considered are mentioned.

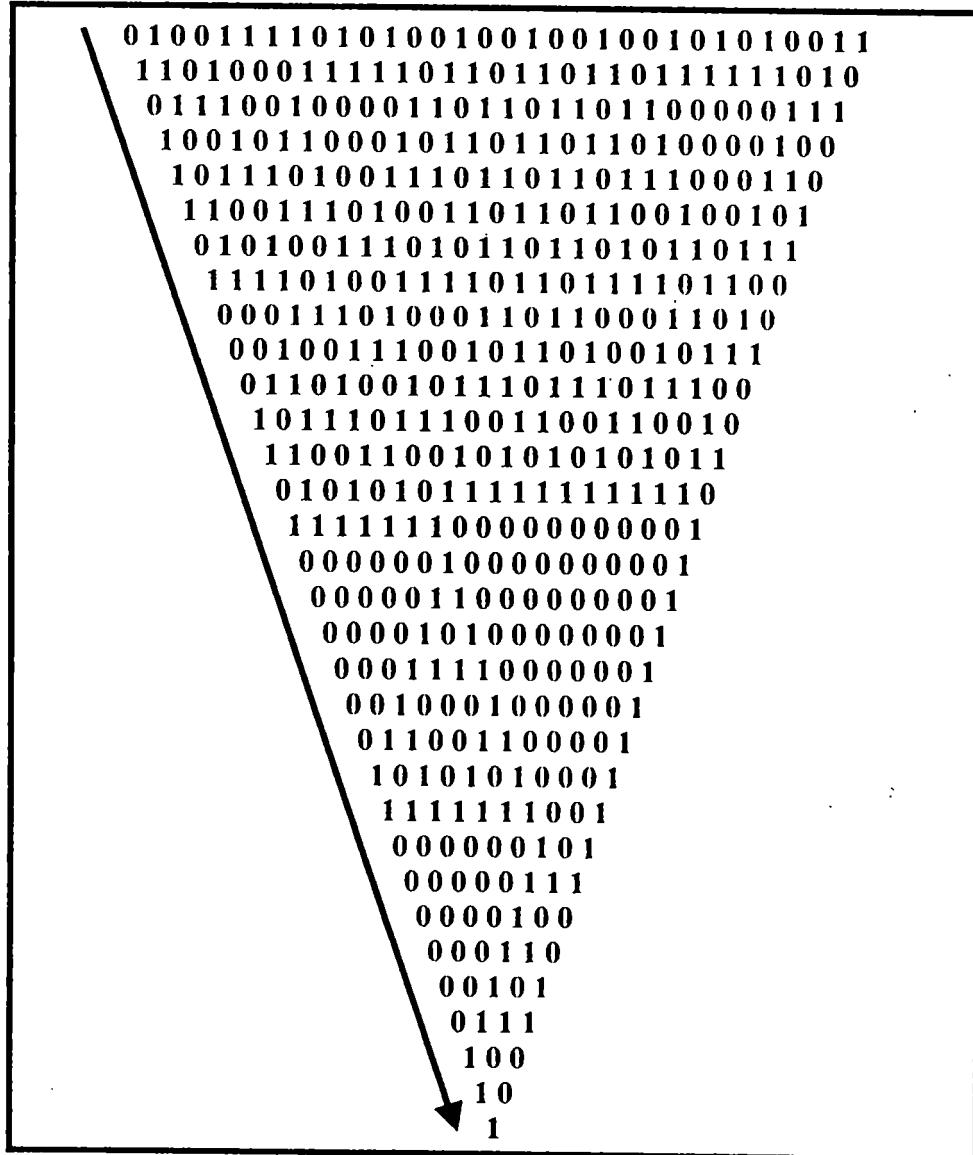
**Table 3.3.3**  
**Different Blocks Considered for Encryption**

<b>Block 1</b>	01010100/01000101/01010010/01010010	<b>Block 9</b>	00101111/01010100/01000101/01001100
<b>Option</b>	001	<b>Option</b>	010
<b>Block 2</b>	01001111/01010010/01001001/01010011	<b>Block 10</b>	00100000/01010011/01001110/01001111
<b>Option</b>	001	<b>Option</b>	011
<b>Block 3</b>	01010011/00100000/01000001/01010100	<b>Block 11</b>	01010111/01010110/01001001/01000101
<b>Option</b>	100	<b>Option</b>	011
<b>Block 4</b>	01010100/0100011/01000011/01001011	<b>Block 12</b>	01010111/00100000
<b>Option</b>	010	<b>Option</b>	011
<b>Block 5</b>	00100000/01010011/01010101	<b>Block 13</b>	01000100/01000101/01000011
<b>Option</b>	001	<b>Option</b>	001
<b>Block 6</b>	01010011/01010000/01000101	<b>Block 14</b>	00100000/00110000
<b>Option</b>	100	<b>Option</b>	100
<b>Block 7</b>	01000011/01010100/01000101	<b>Block 15</b>	00110110/00100000/01000001
<b>Option</b>	100	<b>Option</b>	100
<b>Block 8</b>	01000100/00100000/01001000	<b>Block 16</b>	01001100/01000101/01010010/01010100
<b>Option</b>	100	<b>Option</b>	100

Figure 3.3.3 to figure 3.3.18 show the formations of all the triangles corresponding to block 1 to block 16 respectively. Along with the triangle, each figure also shows the corresponding target block following the option mentioned against each block in table 3.3.3.



**Figure 3.3.3**  
**Construction of Target Block for Block 1 from the Generated Triangle**



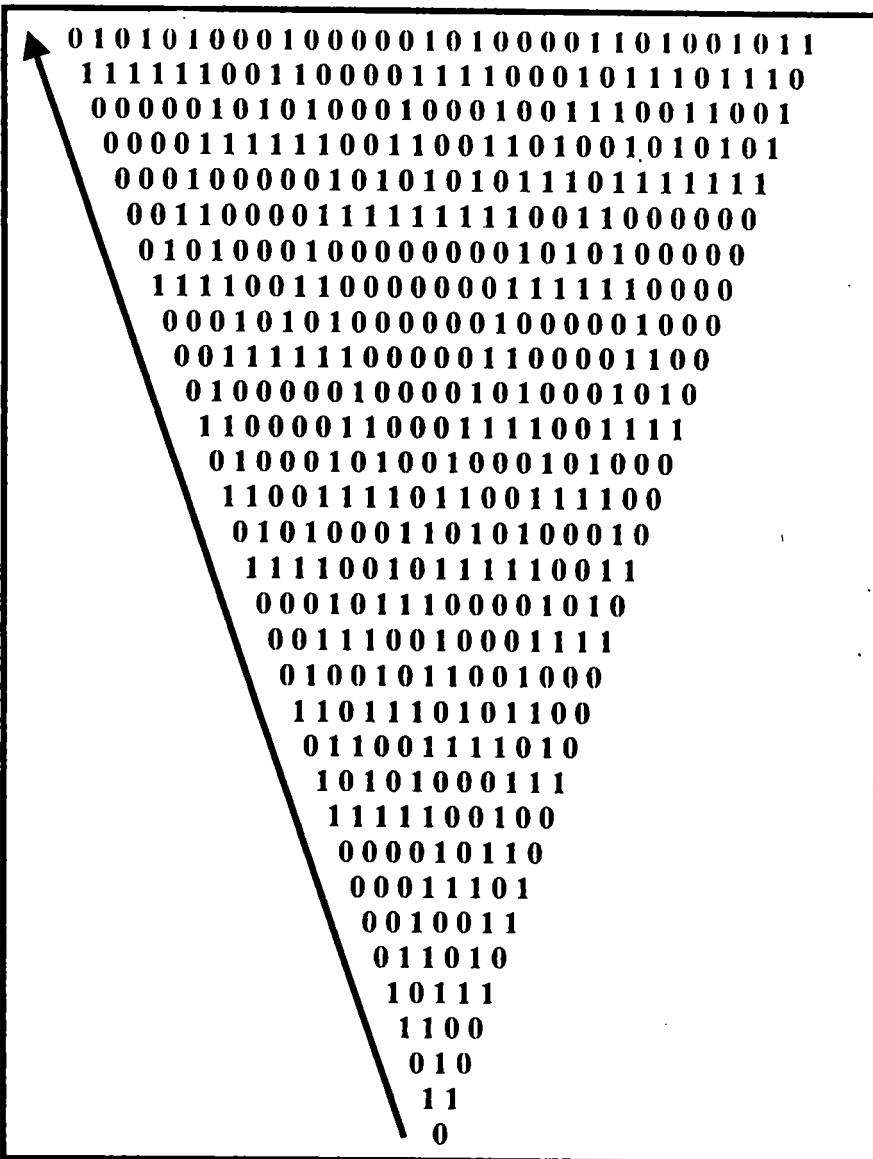
**Figure 3.3.4**  
**Construction of Target Block for Block 2 from the Generated Triangle**

```

01010011001000000100000101010100
1111010101100000110000111111110
000111111010000101000100000001
001000001110001111001100000001
0110000100100100010101000001
101000110110110011111100001
11100101101101010000010001
0010111011011111000011001
011100110110000100010101
10010101101000110011111
1011111011100101010000
11000011001011111000
01000101011100000100
1100111110010000110
010100001011000101
11110001110100111
0001001001110100
001101101001110
01011011101001
11101100111101
0011010100111
01011111010
1110000111
001000100
01100110
1010101
111111
00000
0000
00
0

```

**Figure 3.3.5**  
**Construction of Target Block for Block 3 from the Generated Triangle**



```

01010100010000010100001101001011
1111110011000011110001011101110
000001010100010001001110011001
00001111110011001101001010101
0001000001010101011101111111
001100001111111110011000000
01010001000000001010100000
1111001100000001111110000
000101010000001000001000
001111110000001100001100
0100000100001010001010
110000110001111001111
01000101001000101000
1100111101100111100
010100011010100010
11110010111110011
0001011100001010
001110010001111
01001011001000
1101110101100
011001111010
10101000111
1111100100
000010110
00011101
0010011
011010
10111
1100
010
11
0

```

**Figure 3.3.6**  
**Construction of Target Block for Block 4 from the Generated Triangle**

```

00100000010100110101010101
011000001111010111111111
1010000100011110000000
1110001100100010000000
00100101010011000000
011011110101010000
10110000111111000
11010001000000100
0111001100000110
100101010000101
10111111000111
1100000100100
010000110110
11000101101
0100111011
110100110
01110101
1001111
101000
11100
0010
011
10
1

```

**Figure 3.3.7**  
**Construction of Target Block for Block 5 from the Generated Triangle**

```

010100110101000001000101
11110101111100001100111
0001111000010001010100
00100010001100111110
01100110010101000001
101010101111100001
111111110000010001
00000001000011001
00000001100010101
0000001010011111
00001111010000
0001000111000
001100100100
01010110110
1111101101
0000110111
00010110
0011101
010011
11010
0111
100
10
1

```

**Figure 3.3.8**  
**Construction of Target Block for Block 6 from the Generated Triangle**

```

010000110101010001000101
1100010111111001100111
010011000000101010100
110100100000111111110
01110110000100000001
1001101000110000001
101011100101000001
11110010111100001
0001011100010001
001110010011001
01001011010101
1101110111111
011001100000
10101010000
1111111000
000000100
00000110
0000101
000111
00100
0110
101
11
0

```

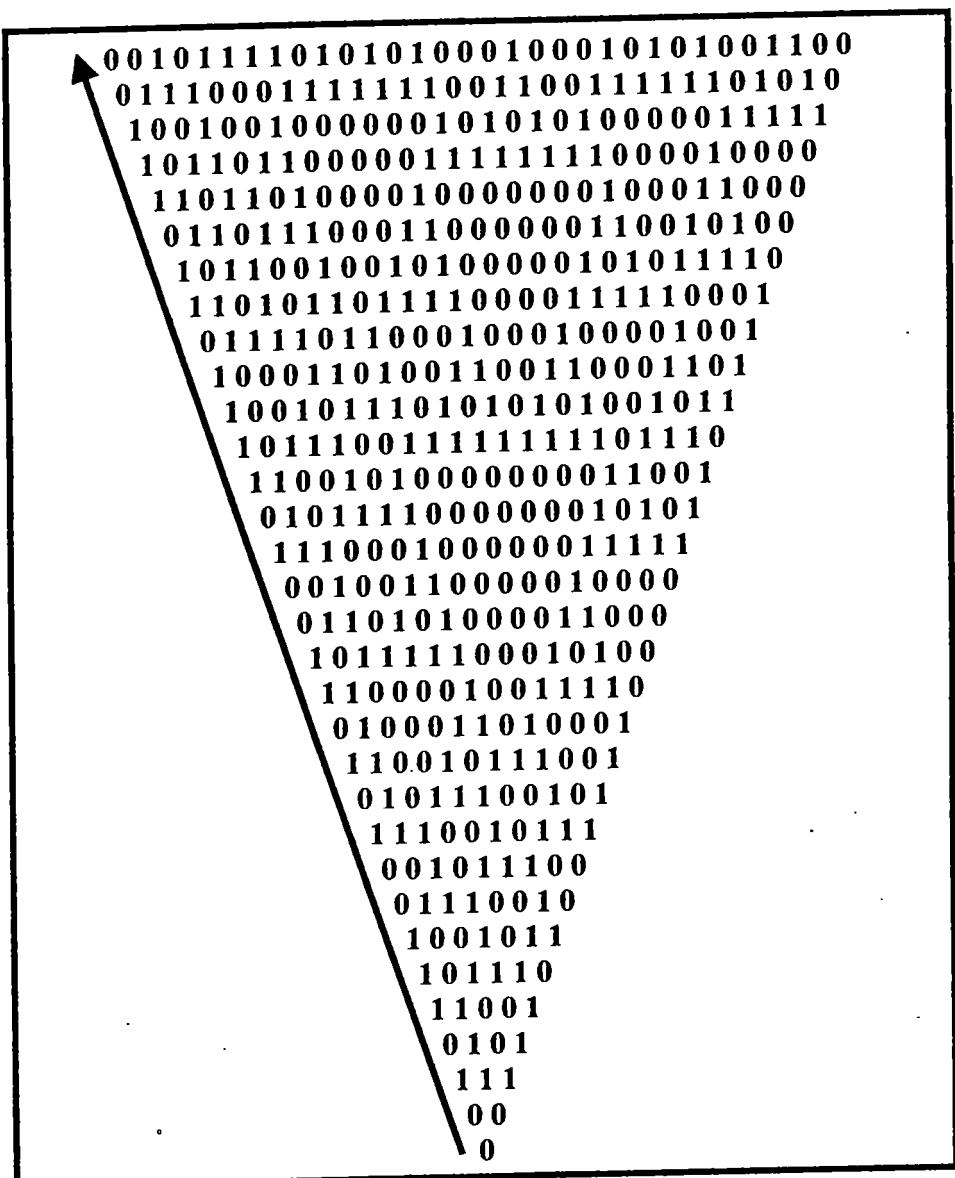
**Figure 3.3.9**  
**Construction of Target Block for Block 7 from the Generated Triangle**

```

010001000010000001001000
11001100011000001101100
0101010010100001011010
111111011110001110111
00000110001001001100
0000101001101101010
000111101011011111
00100011110110000
0110010001101000
101011001011100
11110101110010
0001111001011
001000101110
01100111001
1010100101
1111101111
00001100
0001010
001111
01000
1100
010
11
0

```

**Figure 3.3.10**  
**Construction of Target Block for Block 8 from the Generated Triangle**



**Figure 3.3.11**  
**Construction of Target Block for Block 9 from the Generated Triangle**

```

00100000010100110100111001001111
0110000011110101110100101101000
101000010001111001110111011100
11100011001000101001100110010
00100101011001111010101011
01101111101010001111111110
10110000111110010000000001
1101000100001011000000001
011100110001110100000001
100101010010011100000001
10111111011010010000001
1100000110111011000001
01000010110011010001
11000111010111001
01001001111100101
11011010000010111
0110111000011100
101100100010010
11010110011011
0111101010110
100011111101
10010000011
1011000010
110100011
01110010
1001011
101110
11001
0101
111
00
0

```

**Figure 3.3.12**  
**Construction of Target Block for Block 10 from the Generated Triangle**

```

01010111010101100100100101000101
111110011111010110110111100111
000010100000111101101100010100
00011110000100011011010011110
0010001000110010110111010001
011001100101011101100111001
1010101011110011010100101
1111111100001010111110111
000000010001111100001100
00000011001000010001010
0000010101100011001111
000011111010010101000
0001000011101111100
0011000100110000010
010100110101000011
11110101111100010
0001111000010011
001000100011010
01100110010111
1010101011100
111111110010
00000001011
00000001110
0000001001
00001101
0001011
001110
01001
1101
011
10
1

```

**Figure 3.3.13**  
**Construction of Target Block for Block 11 from the Generated Triangle**

```

0101011100100000
1111110010110000
000010111101000
0001110011100
001001010010
01101111011
1011000110
110100101
01110111
1001100
101010
11111
0000
000
00
0

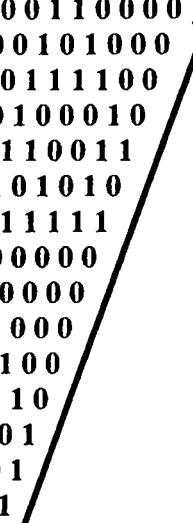
```

**Figure 3.3.14**  
**Construction of Target Block for Block 12 from the Generated Triangle**

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

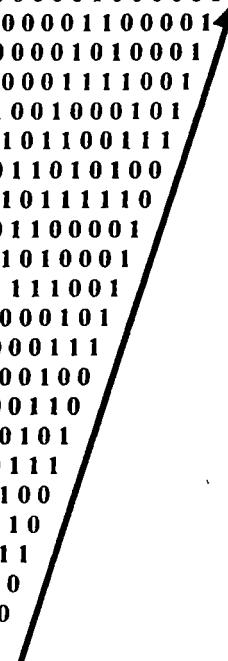
**Figure 3.3.15**  
**Construction of Target Block for Block 13 from the Generated Triangle**

0010000000110000  
0110000001010000  
10100000111100  
1110000100010  
001000110011  
01100101010  
1010111111  
111100000  
00010000  
0011000  
010100  
11110  
0001  
001  
01  
1



**Figure 3.3.16**  
**Construction of Target Block for Block 14 from the Generated Triangle**

00110110001000001000001  
01011010011000001100001  
1110111010100001010001  
001100111110001111001  
01010100001001000101  
1111110001101100111  
000001001011010100  
00001101110111110  
0001011001100001  
001110101010001  
01001111111001  
1101000000101  
011100000111  
10010000100  
1011000110  
110100101  
01110111  
1001100  
101010  
11111  
0000  
000  
00  
0



**Figure 3.3.17**  
**Construction of Target Block for Block 15 from the Generated Triangle**

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

**Figure 3.3.18**  
**Construction of Target Block for Block 16 from the Generated Triangle**

Table 3.3.4 enlists all target blocks formed through figures 3.3.3 to 3.3.18.

**Table 3.3.4**  
**All Target Blocks**

<b>Target Block 1</b>	<b>01000001000100000000011000010000</b>
<b>Target Block 2</b>	<b>01011101000110100000011000000111</b>
<b>Target Block 3</b>	<b>0000011001011100110000111111100</b>
<b>Target Block 4</b>	<b>01011000011010001010100010000010</b>
<b>Target Block 5</b>	<b>001100110111010101110011</b>
<b>Target Block 6</b>	<b>1001011011000001111110011</b>
<b>Target Block 7</b>	<b>011001100000111111110011</b>
<b>Target Block 8</b>	<b>010001001110100001001000</b>
<b>Target Block 9</b>	<b>00101110010101000101111011011100</b>
<b>Target Block 10</b>	<b>100010111111110010110101011100</b>
<b>Target Block 11</b>	<b>11001111001000101010010111011101</b>
<b>Target Block 12</b>	<b>0000010110010000</b>
<b>Target Block 13</b>	<b>010100000000000100000111</b>
<b>Target Block 14</b>	<b>1111000001010000</b>
<b>Target Block 15</b>	<b>000010011001111100111111</b>
<b>Target Block 16</b>	<b>00010001000011110110011011111100</b>

By combining all target blocks shown in table 3.3.4, we obtain the following stream of bits as the encrypted stream:

010000010001000000000110000100000101110100011010000001100000011100000110  
 0101110011000011111110001011000011010001010100010000110011001101110101  
 0111001110010110110000111110011011001100000111111100110100010011101000  
 01001000001011100101011001011110110111001000101111111110010110101011100  
 1100111100100010100101110111010000010110010000010100000000000100000111  
 1111000001010000000010011001111100111110010001000011110110011011111100

Using “/” as the separator between two consecutive bytes, we present the encrypted stream like the following:

01000001/00010000/00000110/00010000/01011101/00011010/00000110/00000111/000  
 00110/01011100/11000011/11111100/01011000/01101000/10101000/10000010/001100  
 11/01110101/01110011/10010110/11000011/11110011/01100110/00001111/11110011/  
 01000100/11101000/01001000/00101110/01010110/01011110/11011100/10001011/111  
 1111/00101101/01011100/11001111/00100010/10100101/11011101/00000101/100100  
 00/01010000/00000001/00000111/11110000/01010000/00001001/10011111/00111111/  
 00010001/00001111/01100110/11111100

Table 3.3.5 converts bytes in the encrypted stream into characters.

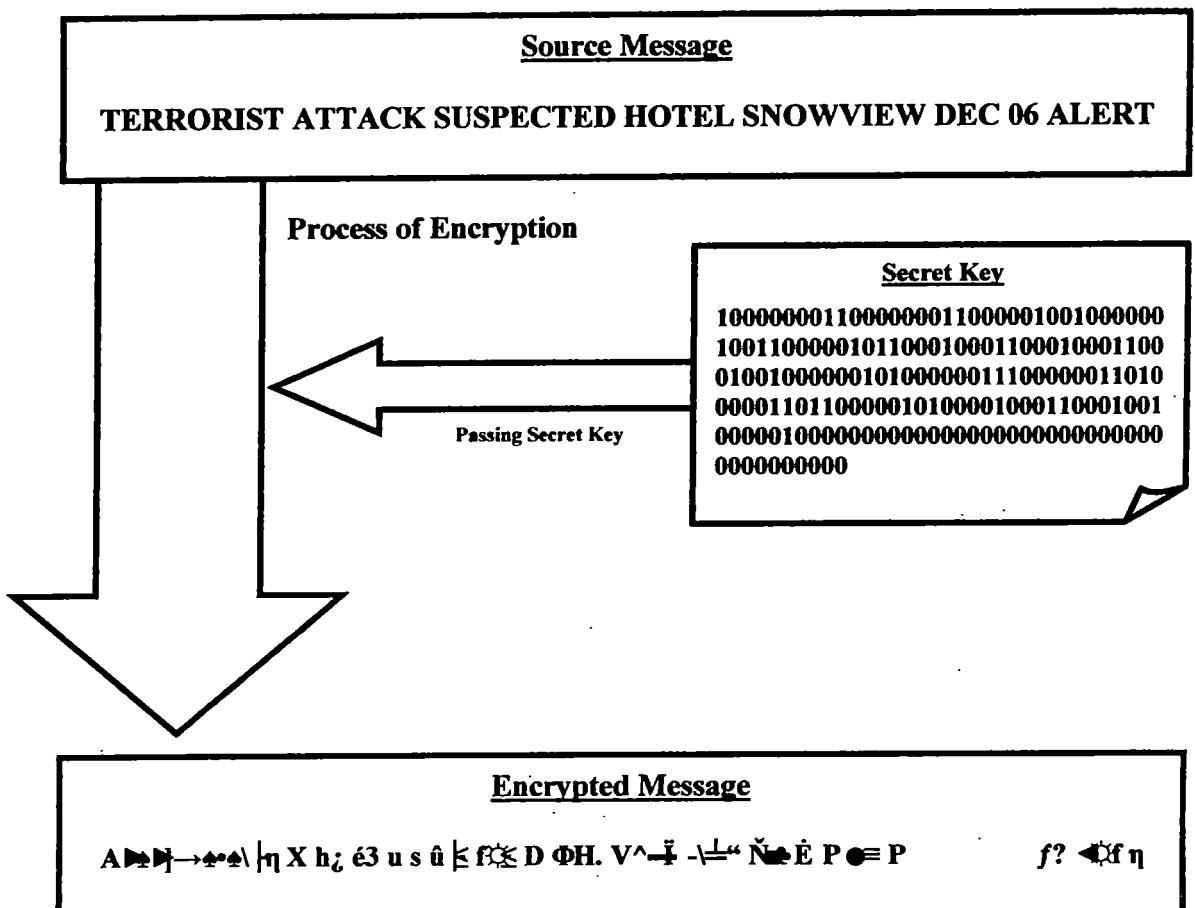
**Table 3.3.5**  
**Converting Bytes into Characters**

Byte	Character	Byte	Character	Byte	Character	Byte	Character
01000001	A	10000010	é	01011110	^	11110000	≡
00010000	►	00110011	3	11011100	—	01010000	P
00000110	♠	01110101	u	10001011	ī	00001001	<Tab>
00010000	►	01110011	s	11111111	<Blank>	10011111	f
01011101	]	10010110	û	00101101	-	00111111	?
00011010	→	11000011		01011100	\	00010001	◀
00000110	♣	11110011	≤	11001111	±	00001111	☀
00000111	•	01100110	f	00100010	“	01100110	f
00000110	♠	00001111	⊗	10100101	Ñ	11111100	η
01011100	\	11110011	≤	11011101	■		
11000011		01000100	D	00000101	♣		
11111100	η	11101000	Φ	10010000	È		
01011000	X	01001000	H	01010000	P		
01101000	h	00101110	.	00000001	⊕		
10101000	ζ	01010110	V	00000111	.		

All characters from table 3.3.5 are combined sequentially to obtain the following:

“A►►→♣♦|η X hζ é3 u s û ≤ f⊗ D ΦH. V^—± -±“ Ñ⊕È P ⊕ P f? ◀f η”

This is the ciphertext to be transmitted corresponding to the plaintext taken for the implementation purpose. Figure 3.3.19 shows it in a pictorial manner.



**Figure 3.3.19**  
**Generating Encrypted Message from Source using 180-Bit Secret Key**

### 3.4 Results

For the purpose of implementation, a unique size of 64 bits has been considered, and for each such block, an asymmetric technique has been implemented.

Section 3.4.1 presents report of the encryption/decryption times and the Chi Square values. Section 3.4.2 presents result of frequency distribution tests. A comparative result with the RSA system is shown in section 3.4.3.

#### 3.4.1 Result of Encryption/Decryption Time and Chi Square Value

Section 3.4.1.1 presents result for the .EXE files, section 3.4.1.2 presents result for the .COM files, section 3.4.1.3 presents result for the .DLL files, section 3.4.1.4

presents results for the .SYS files, and section 3.4.1.5 presents results for the .CPP files. For all the sections, the tabular representation of results include the encryption/decryption time, the numbers of operations required during encryption as well as decryption, and the chi square value [44, 48, 55, 56].

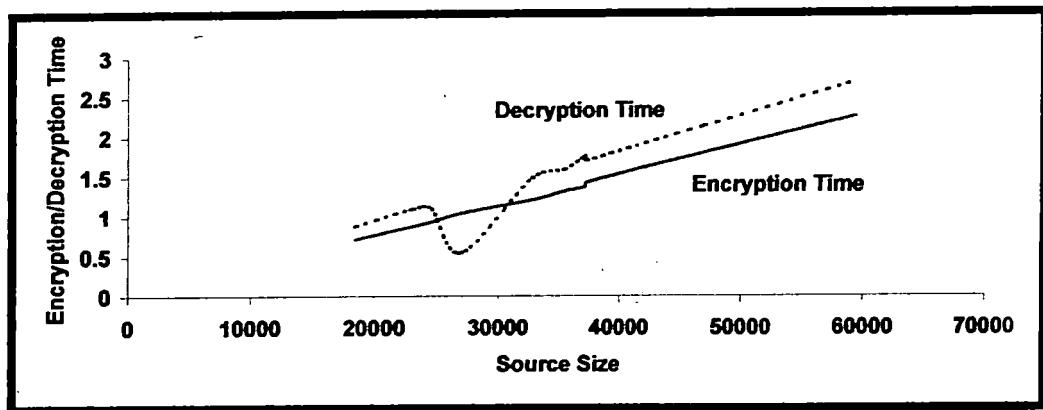
### 3.4.1.1 Result for **EXE** Files

Table 3.4.1.1.1 shows results. Ten files have been considered. The size of the files varies from 11611 bytes to 59398 bytes. The encryption time by using the proposed TE technique ranges from 0.714286 seconds to 5.439560 seconds. Using the TE technique for decryption, the decryption time ranges from 0.549451 to 2.692307 seconds. At the time of encryption, the total number of operations ranges from 74303488 to 448829696. During the process of decryption, the total number of operations ranges from 74041856 to 416387328. The Chi Square value between the source and the corresponding encrypted file ranges from 17993 to 158628 with the degree of freedom ranging from 248 to 255.

**Table 3.4.1.1.1**  
**Result for .EXE Files for TE Technique**

Source File	Encryption Time	Decryption Time	No. of Operations (During Encryption)	No. of Operations (During Decryption)	Chi Square Value	Degree of Freedom
<i>TLIB.EXE</i>	1.428571	1.703297	76004096	76004096	128674	255
<i>MAKER.EXE</i>	2.252747	2.692307	197401344	197401344	158628	255
<i>UNZIP.EXE</i>	0.879121	1.098901	244495104	244495104	17993	255
<i>RPPO.EXE</i>	1.318681	1.593407	316443904	316836352	55351	255
<i>PRIME.EXE</i>	1.373626	1.8242	393756160	392709632	62986	255
<i>TCDEF.EXE</i>	1.043956	0.549451	448829696	416387328	33539	254
<i>TRIANGLE.EXE</i>	5.439560	1.703297	74303488	74041856	57961	255
<i>PING.EXE</i>	0.934066	1.098901	124536832	124275200	79747	248
<i>NETSTAT.EXE</i>	1.208791	1.483516	191514624	191252992	110332	255
<i>CLIPBRD.EXE</i>	0.714286	0.879121	229189632	228928000	65560	255

The graphical relationship between the source file size and the encryption/decryption time for .EXE files is shown in figure 3.4.1.1.1. It is observed in the graph that there exists a tendency that the encryption/decryption time varies linearly with the source file size, although there also exist a few exceptions.



**Figure 3.4.1.1.1**  
**Relationship between Source Size and Encryption/Decryption Time for .EXE Files in TE Technique**

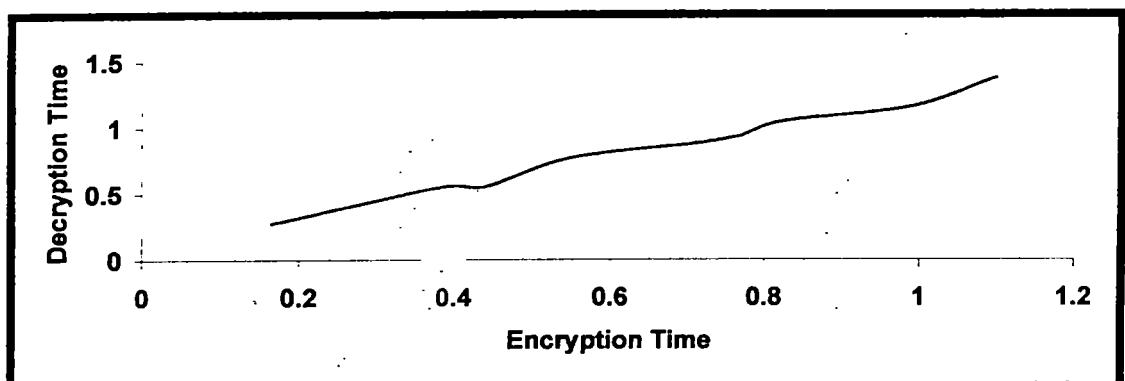
### 3.4.1.2 Result for COM Files

Table 3.4.1.2.1 shows results. Ten files have been considered. The size of the files varies from 5239 bytes to 29271 bytes. The encryption time by using the proposed TE technique ranges from 0.164835 seconds to 0.989011 seconds. Using the TE technique for decryption, the decryption time ranges from 0.274725 to 1.373626 seconds. At the time of encryption, the total number of operations ranges from 40160512 to 361052160. During the process of decryption, the total number of operations also ranges from 40160512 to 361052160. The Chi Square value between the source and the corresponding encrypted file ranges from 6451 to 63314 with the degree of freedom ranging from 230 to 255.

**Table 3.4.1.2.1**  
**Result for .COM Files for TE Technique**

Source File	Encryption Time	Decryption Time	No. of Operations (During Encryption)	No. of Operations (During Decryption)	Chi Square Value	Degree of Freedom
<i>EMSTEST.COM</i>	0.769231	0.934066	40160512	40160512	33211	255
<i>THELP.COM</i>	0.439560	0.549451	62791680	62791680	28716	250
<i>WIN.COM</i>	0.989011	1.153846	113417472	113417472	47792	252
<i>KEYB.COM</i>	0.769231	0.934066	154101248	154101248	48939	255
<i>CHOICE.COM</i>	0.164835	0.274725	164697344	164697344	9254	232
<i>DISKCOPY.COM</i>	0.824176	1.043956	209567232	209567232	51815	254
<i>DOSKEY.COM</i>	0.549451	0.769231	241224704	241224704	39119	253
<i>MODE.COM</i>	1.098901	1.373626	301007616	301007616	63314	255
<i>MORE.COM</i>	0.384615	0.549451	322330624	322330624	6451	230
<i>SYS.COM</i>	0.714286	0.879121	361052160	361052160	43706	254

Figure 3.4.1.2.1 shows how the decryption time changes with the encryption time for .COM files. This graph establishes the fact that the time complexity during the process of encryption varies almost linearly with that during the process of decryption.



**Table 3.4.1.2.1**  
**Relationship between Encryption Time and Decryption Time for .COM Files in TE Technique**

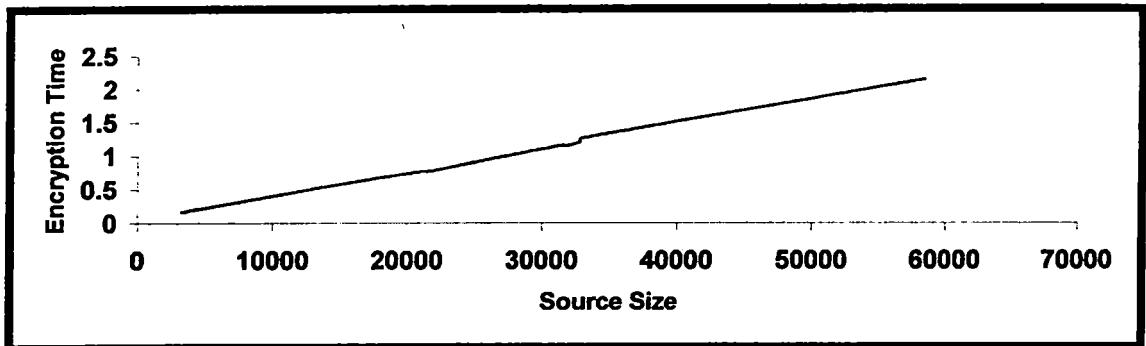
### 3.4.1.3 Result for **DLL** Files

Table 3.4.1.3.1 shows results. Ten files have been considered. The size of the files varies from 3216 bytes to 58368 bytes. The encryption time by using the proposed TE technique ranges from 0.164835 seconds to 2.142857 seconds. Using the TE technique for decryption, the decryption time ranges from 0.164835 seconds to 2.637362 seconds. At the time of encryption, the total number of operations ranges from 66977792 to 526403584. During the process of decryption, the total number of operations also ranges from 66977792 to 526403584. The Chi Square value between the source and the corresponding encrypted file ranges from 8383 to 205511 with the degree of freedom ranging from 217 to 255.

**Table 3.4.1.3.1**  
**Result for .DLL Files for TE Technique**

Source File	Encryption Time	Decryption Time	No. of Operations (During Encryption)	No. of Operations (During Decryption)	Chi Square Value	Degree of Freedom
<i>SNMPAPI.DLL</i>	1.208791	1.483516	66977792	66977792	70479	253
<i>KPSHARP.DLL</i>	1.153846	1.483516	131862528	131862528	203671	254
<i>WINSOCK.DLL</i>	0.769231	0.989011	175816704	175816704	105418	252
<i>SPWHPT.DLL</i>	1.263736	1.428571	242794496	242794496	182841	255
<i>HIDCI.DLL</i>	0.164835	0.164835	249335296	249335296	8383	217
<i>PFPICK.DLL</i>	2.142857	2.637362	368639488	368639488	153199	255
<i>NDDEAPI.DLL</i>	0.549451	0.659341	397288192	397288192	56969	249
<i>NDDENB.DLL</i>	0.439560	0.494505	419657728	419657728	60992	251
<i>ICCCODES.DLL</i>	0.769231	0.989011	462565376	462565376	168116	252
<i>KPSCALE.DLL</i>	1.153846	1.428571	526403584	526403584	205511	255

- The linear relationship between the source size and the encryption time for .DLL files is shown in figure 3.4.1.3.1.



**Table 3.4.1.3.1**  
**Linear Relationship between Source Size and Encryption Time for**  
**.DLL Files in TE Technique**

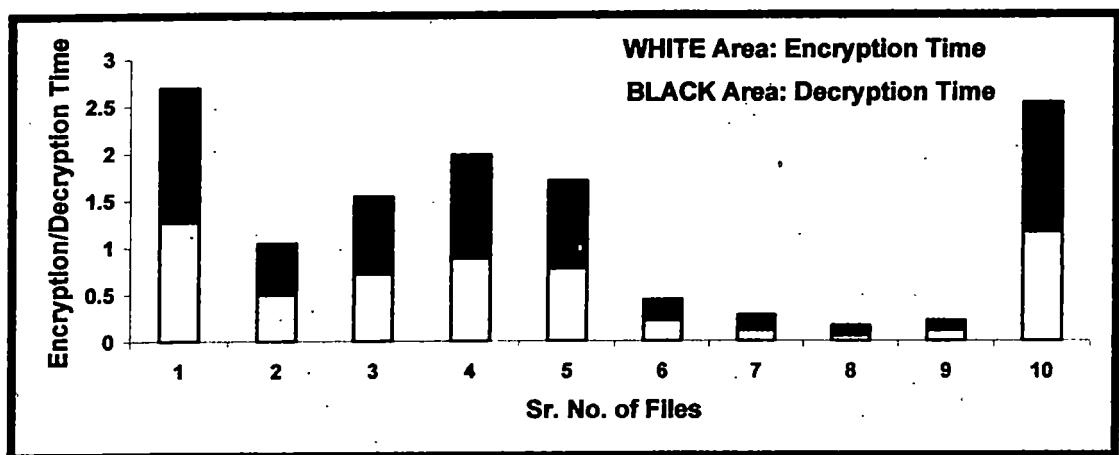
#### 3.4.1.4 Result for SYS Files

Table 3.4.1.4.1 shows results. Ten files have been considered. The size of the files varies from 1105 bytes to 33191 bytes. The encryption time by using the proposed TE technique ranges from 0.054945 seconds to 1.263736 seconds. Using the TE technique for decryption, the decryption time ranges from 0.109890 seconds to 1.428571 seconds. At the time of encryption, the total number of operations ranges from 38590720 to 222648832. During the process of decryption, the total number of operations also ranges from 38590720 to 222648832. The Chi Square value between the source and the corresponding encrypted file ranges from 859 to 126367 with the degree of freedom ranging from 165 to 255.

**Table 3.4.1.4.1**  
**Result for .SYS Files for TE Technique**

Source File	Encryption Time	Decryption Time	No. of Operations (During Encryption)	No. of Operations (During Decryption)	Chi Square Value	Degree of Freedom
HIMEM.SYS	1.263736	1.428571	67762688	67762688	63553	255
RAMDRIVE.SYS	0.494505	0.549451	93533440	93533440	16452	241
USBD.SYS	0.714286	0.824176	38590720	3859072	84593	255
CMD640X.SYS	0.879121	1.098901	88824064	88824064	65706	255
CMD640X2.SYS	0.769231	0.934066	131470080	131470080	58201	255
REDBOOK.SYS	0.219780	0.219780	142981888	142981888	21052	230
IFSHLP.SYS	0.109890	0.164835	150438400	150438400	6488	237
ASPI2HLP.SYS	0.054945	0.109890	152662272	152662272	859	165
DBLBUFF.SYS	0.109890	0.109890	157894912	157894912	3190	215
CCPORT.SYS	1.153846	1.373626	222648832	222648832	126367	255

Figure 3.4.1.4.1 graphically compares the encryption time and the decryption time for .SYS files, in which the white part stands for the encryption time, and the black portion stands for the decryption time. For each file, the corresponding pillar stands for the total time required for the encryption and the decryption purposes. Out of each of the pillars, almost half is occupied by the white part and the remaining by the black part, which establishes the fact that the encryption time and the decryption time are almost equal to each other.



**Figure 3.4.1.4.1**  
**Comparison between Encryption Time and Decryption Time for .SYS Files in TE Technique**

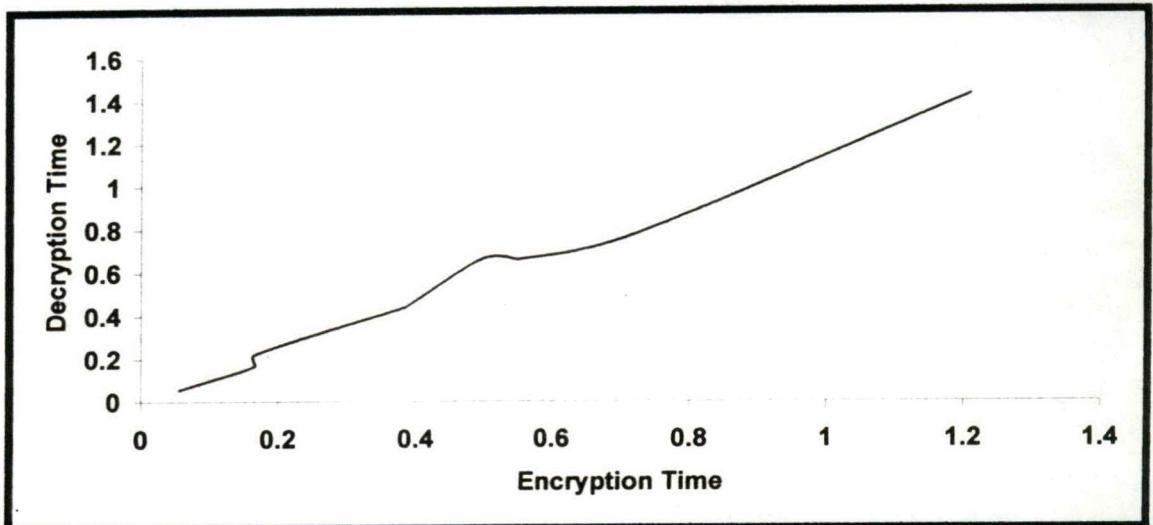
### 3.4.1.5 Result for CPP Files

Table 3.4.1.5.1 shows results. Ten files have been considered. The size of the files varies from 4071 bytes to 14557 bytes. The encryption time by using the proposed TE technique ranges from 0.054945 seconds to 0.164835 seconds. Using the TE technique for decryption, the decryption time ranges from 0.164835 seconds to 1.428571 seconds. At the time of encryption, the total number of operations ranges from 34142976 to 246980608. During the process of decryption, the total number of operations also ranges from 34142976 to 246980608. The Chi Square value between the source and the corresponding encrypted file ranges from 1305 to 144422 with the degree of freedom ranging from 69 to 90.

**Table 3.4.1.5.1  
Result for .CPP Files for TE Technique**

Source File	Encryption Time	Decryption Time	No. of Operations (During Encryption)	No. of Operations (During Decryption)	Chi Square Value	Degree of Freedom
<i>BRICKS.CPP</i>	0.714286	0.769231	34142976	34142976	68626	88
<i>PROJECT.CPP</i>	1.208791	1.428571	99812608	99812608	144422	90
<i>ARITH.CPP</i>	0.384615	0.439560	119304192	119304192	15466	77
<i>START.CPP</i>	0.549451	0.659341	148999424	148999424	47550	88
<i>CHARTCOM.CPP</i>	0.494505	0.659341	177778944	177778944	45205	84
<i>BITIO.CPP</i>	0.164835	0.164835	186020352	186020352	8387	70
<i>MAINC.CPP</i>	0.164835	0.219780	195439104	195439104	7916	83
<i>TTEST.CPP</i>	0.054945	0.054945	197924608	197924608	1305	69
<i>DO.CPP</i>	0.549451	0.659341	227489024	227489024	48061	88
<i>CAL.CPP</i>	0.384615	0.439560	246980608	246980608	17336	77

The linear relationship between the encryption time and the decryption time for .CPP files is shown in figure 3.4.1.5.1

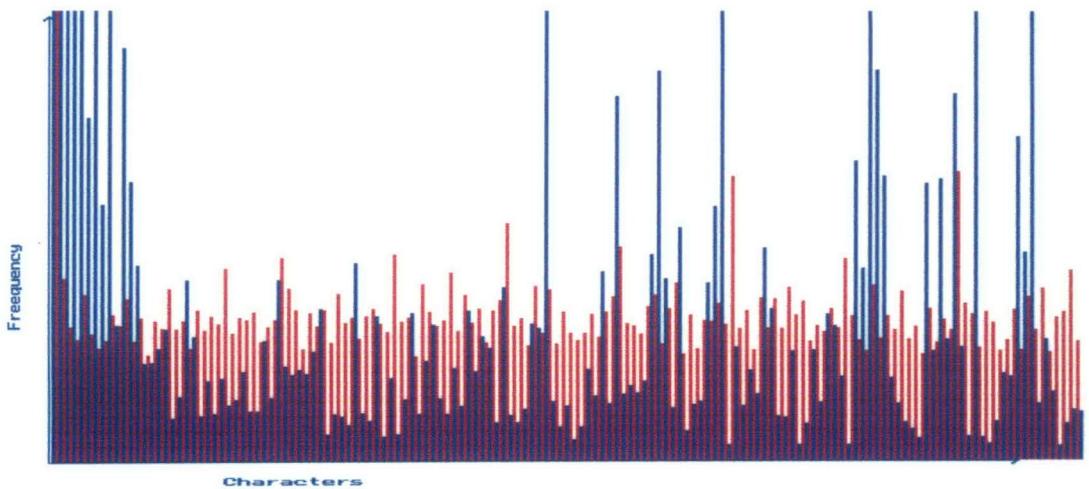


**Figure 3.4.1.5.1**  
**Linear Relationship between Encryption Time and Decryption Time for .CPP Files**

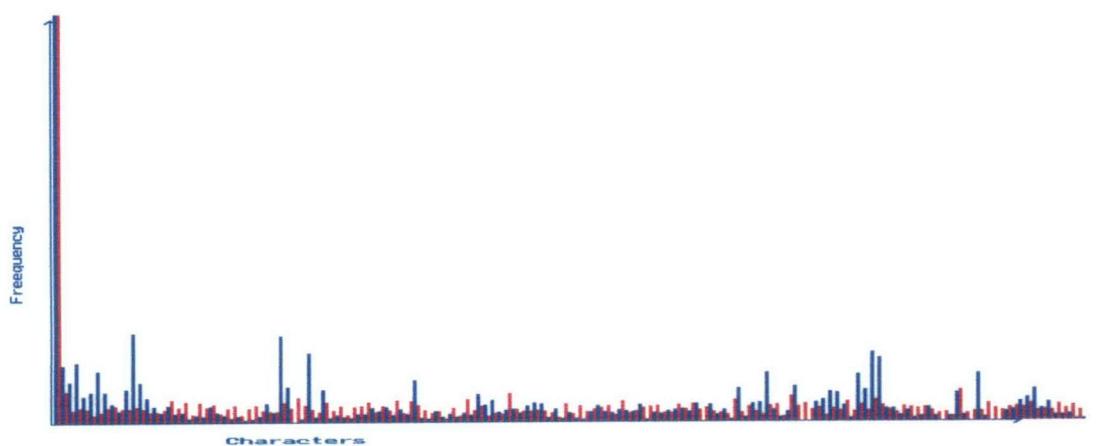
### 3.4.2 Result of Frequency Distribution Tests

For each sample file, the frequency of each character is computed and it is compared with that in the corresponding encrypted file. It is seen for all cases that the characters in the encrypted files are well distributed. This shows that the technique proposed here is quite compatible with available techniques. The bars in red color represent frequencies of characters in the encrypted file and those in blue color represent frequencies of characters in the source file. The frequencies of characters in encrypted files are evenly distributed. It can be said that the source and the corresponding encrypted file are heterogeneous in nature. Therefore it can be interpreted that through the proposed technique, a good quality of encryption is obtained.

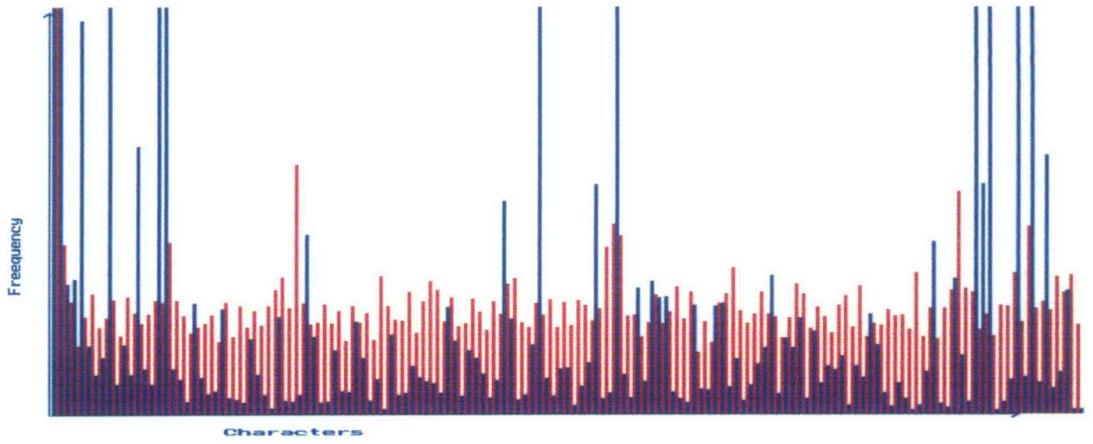
Figure 3.4.2.1 to figure 3.4.2.5 shows five of such results obtained through the frequency distribution tests, taking one from each category of files.



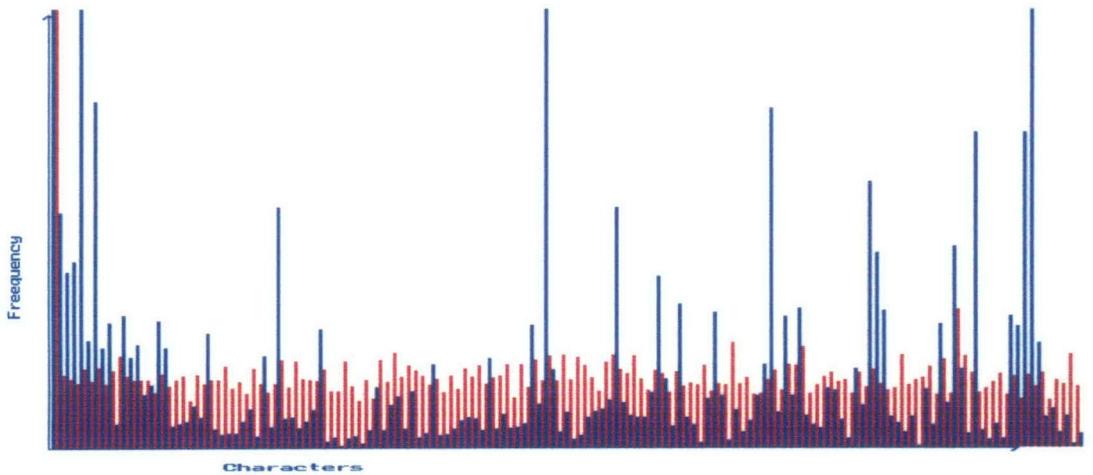
**Figure 3.4.2.1**  
**Frequency Distribution of Characters between**  
**TRIANGLE.EXE and Encrypted FOX1.EXE**



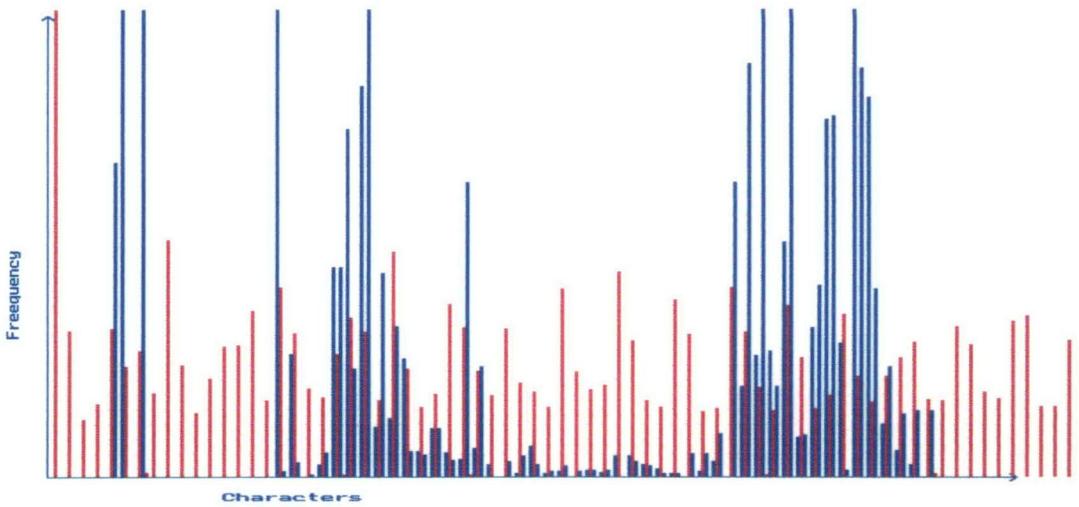
**Figure 3.4.2.2**  
**Frequency Distribution of Characters between  
MORE.COM and Encrypted FOX1.COM**



**Figure 3.4.2.3**  
**Frequency Distribution of Characters between**  
**KPSCALE.DLL and Encrypted FOX1.DLL**



**Figure 3.4.2.4**  
**Frequency Distribution of Characters between**  
**CMD640X2.SYS and Encrypted FOX1.SYS**



**Figure 3.4.2.5**  
**Frequency Distribution of Characters between**  
**DO.CPP and Encrypted FOX1.CPP**

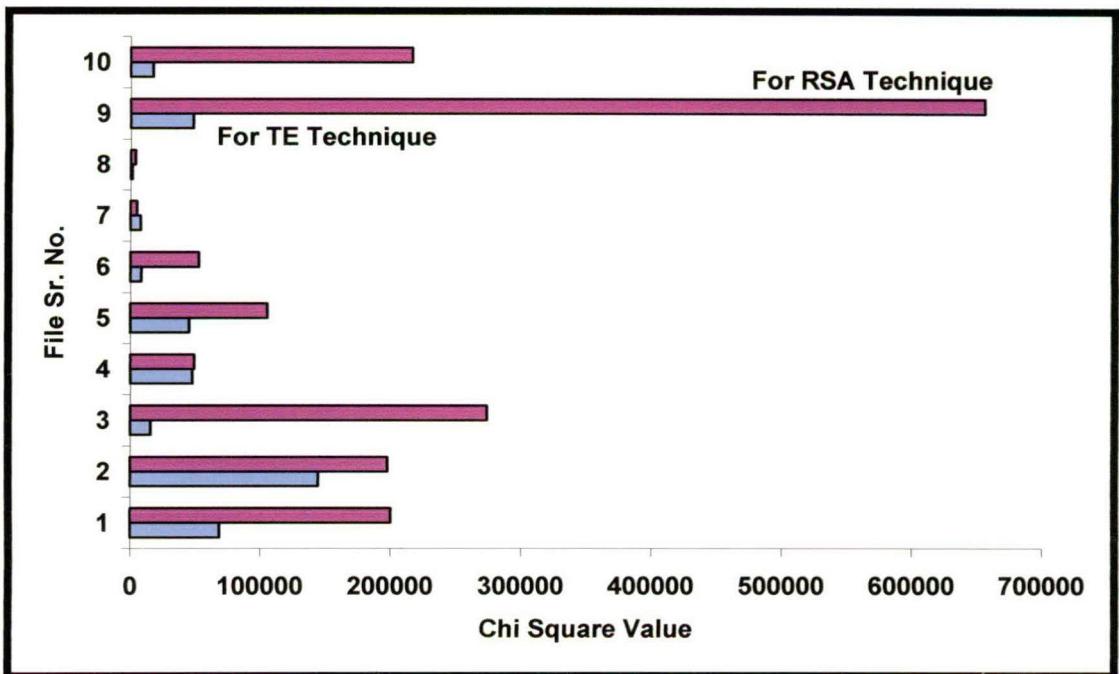
### 3.4.3 Comparison of TE with RSA Technique

On the basis of the Chi Square values between .CPP sample files and the corresponding encrypted files, the proposed TE technique has been compared with the existing RSA system. Ten sample files of different sizes are taken. Chi Square values are computed for both TE and RSA techniques. Both techniques show very high values in Chi Square tests. This proves the high degree of non-homogeneity between source files and corresponding encrypted files [7, 8]. Table 3.4.3.1 enlists these comparative results.

**Table 3.4.3.1**  
**Comparison of Chi Square Values between TE and RSA**

Source File (1)	Source Size	Encrypted File using TE Technique (2)	Encrypted File using RSA Technique (3)	Chi Square Value between (1) and (2)	Chi Square Value between (1) and (3)	Degree of Freedom
BRICKS.CPP	16723	FOX1.CPP	CPP1.CPP	68626	200221	88
PROJECT.CPP	32150	FOX2.CPP	CPP2.CPP	144422	197728	90
ARITH.CPP	9558	FOX3.CPP	CPP3.CPP	15466	273982	77
START.CPP	14557	FOX4.CPP	CPP4.CPP	47550	49242	88
CHARTCOM.CPP	14080	FOX5.CPP	CPP5.CPP	45205	105384	84
BITIO.CPP	4071	FOX6.CPP	CPP6.CPP	8387	52529	70
MAINC.CPP	4663	FOX7.CPP	CPP7.CPP	7916	4964	83
TTEST.CPP	1257	FOX8.CPP	CPP8.CPP	1305	3652	69
DO.CPP	14481	FOX9.CPP	CPP9.CPP	48061	655734	88
CAL.CPP	9540	FOX10.CPP	CPP10.CPP	17336	216498	77

Figure 3.4.3.1 graphically shows the comparison.



**Figure 3.4.3.1**  
**Graphical Comparison of Results of Chi Square Tests for Proposed TE and Existing RSA**

### **3.5 Analysis and Conclusion including Comparison with RPSP**

On the basis of the practical implementation on a sample set of 50 real files, it is observed that the process of encryption/decryption using the proposed TE techniques requires less time on the average than that using the proposed RPSP technique, presented in chapter 2. The average Chi Square value is observed to be higher in case of the TE technique than the RPSP technique [48, 52]. Table 3.5.1 show this comparative result. In chapter 10, graphical comparisons have been drawn.

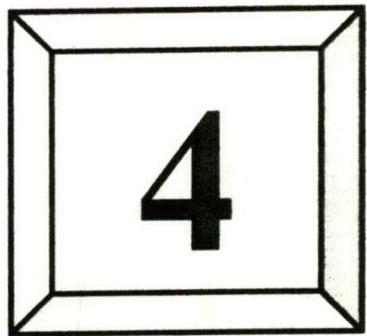
**Table 3.5.1  
Average Encryption/Decryption Time and Chi Square Value obtained in  
RPSP and TE Techniques**

<b>Proposed Technique</b>	<b>Average Encryption Time</b>	<b>Average Decryption Time</b>	<b>Average Chi Square Value</b>	<b>Average Degree of Freedom</b>
<b>RPSP</b>	<b>8.75713800</b>	<b>8.73955200</b>	<b>10701.70</b>	
<b>TE</b>	<b>0.86703290</b>	<b>0.94175818</b>	<b>64188.04</b>	<b>214</b>

The TE policy of encryption is a strong scheme due to the existence of the multiple numbers of options for encrypting each block. Choosing varying block lengths and varying options for different blocks help in requiring a long key space. One such key structure strictly following a set of encryption protocol is proposed in section 8.2.2 in chapter 8.

The encryption/decryption time largely depends on the size of the source file, and the encrypted files are highly non-homogeneous with the respective source files with the encrypted characters are well distributed to almost nullify the chance through cryptanalysis to break the cipher.

By applying flexibility in choosing sizes of blocks and options for each of the blocks, the policy of the TE encryption is capable of producing the information security of a highly satisfactory level. Also, it helps in enhancing performance while taking part in the cascaded approach of encryption.



**Encryption Through  
Recursive Paired Parity Operation  
(RPPO)**

<u>Contents</u>	<u>Pages</u>
<b>4.1      Introduction</b>	<b>132</b>
<b>4.2      The Scheme</b>	<b>133</b>
<b>4.3      Implementation</b>	<b>136</b>
<b>4.4      Results</b>	<b>143</b>
<b>4.5      Analysis and Conclusion with RPSP and TE</b>	<b>156</b>

#### **4.1 Introduction**

Like RPSP, described in chapter 2, the Recursive Paired Parity Operation or the RPPO is also a secret-key cipher system and it also generates a cycle to regenerate the source block. Here also during the process of forming the cycle, any intermediate block can be considered as the encrypted block. After running the same technique for a finite number of more iterations, the source block is regenerated. This is under the part of decryption. In RPSP, one generating function was used to re-orient the positions of different bits in each of the iterations in forming the cycle. In RPPO, the bits are not re-oriented only in their positions but a special Boolean operation is performed on the source and the subsequent blocks of bits. The operation, called the Recursive Paired Parity Operation, is such that after a finite number of iterations, the source block is regenerated. In RPPO, the number of iterations required to complete the cycle follows a certain mathematical policy, while in RPSP this number was not as per a fixed policy.

After decomposing the source stream of bits into a finite number of blocks, the RPPO technique can be applied on each block. Depending on the size of a block, it is fixed that after how many iterations the source block will be regenerated. Accordingly, any intermediate block can be considered as the corresponding encrypted block. It is a wise strategy to take different blocks of varying sizes, so that the key space becomes large enough to almost nullify the chance of breaking the cipher through cryptanalysis. The same type of strategy was also advised in chapter 2 for the RPSP technique.

The technique does not cause any storage overhead. The implementation is well proven with the positive outcome [45, 46, 49].

Section 4.2 of this chapter discusses the entire scheme of this technique with simple examples. Like the RPSP technique, since the entire scheme is the combination of the encryption and the decryption processes, this section also includes how one part of the scheme can be used for the encryption and how the remaining part can be used for the decryption. Section 4.3 shows a simple implementation of the technique, where the same text message as in section 2.3 has been considered for the purpose of transmission using the encryption. Section 4.4 gives the results obtained after implementing the RPPO technique on the same set of real-life files of different categories. Section 4.5 is an analytical presentation of the technique with the concluding remark, where the RPPO

technique has been analyzed from different perspectives and also the mathematical policy that exists in finding the number of iterations required to complete a cycle has been presented.

#### 4.2 The Scheme

The technique, like all other techniques described in the thesis, considers the plaintext as a stream of finite number of bits  $N$ , and is divided into a finite number of blocks, each also containing a finite number of bits  $n$ , where  $1 \leq n \leq N$  [45].

Let  $P = s_0^0 s_1^0 s_2^0 s_3^0 s_4^0 \dots s_{n-1}^0$  is a block of size  $n$  in the plaintext. Then the first intermediate block  $I_1 = s_0^1 s_1^1 s_2^1 s_3^1 s_4^1 \dots s_{n-1}^1$  can be generated from  $P$  in the following way:

$$s_0^1 = s_0^0$$

$$s_i^1 = s_{i-1}^1 \oplus s_i^0, 1 \leq i \leq (n-1); \oplus \text{ stands for the exclusive OR operation.}$$

Now, in the same way, the second intermediate block  $I_2 = s_0^2 s_1^2 s_2^2 s_3^2 s_4^2 \dots s_{n-1}^2$  of the same size ( $n$ ) can be generated by:

$$s_0^2 = s_0^1$$

$$s_i^2 = s_{i-1}^2 \oplus s_i^1, 1 \leq i \leq (n-1).$$

After this process continues for a finite number of iterations, which depends on the value of  $n$ , the source block  $P$  is regenerated.

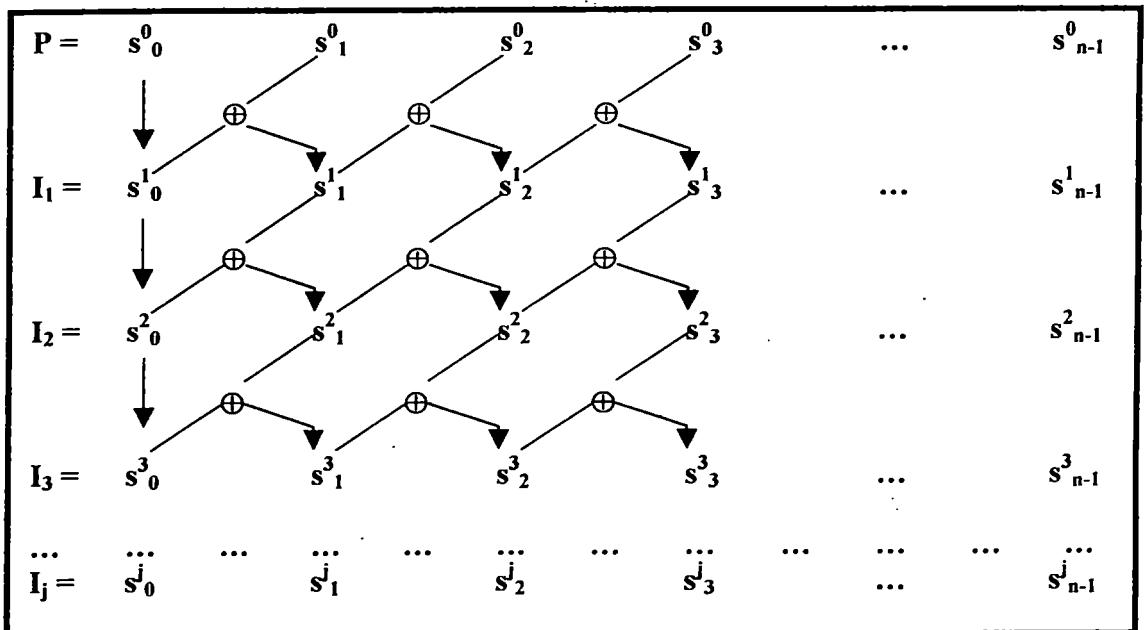
If the number of iterations required to regenerate the source block is assumed to be  $I$ , the generation of any intermediate or the final block can be generalized as follows:

$$s_0^j = s_0^{j-1}$$

$$s_i^j = s_{i-1}^j \oplus s_i^{j-1}, 1 \leq i \leq (n-1); \text{ where } 1 \leq j \leq I.$$

In this generalized formulation system, the final block, which in turn is the source block, is generated when  $j = I$ .

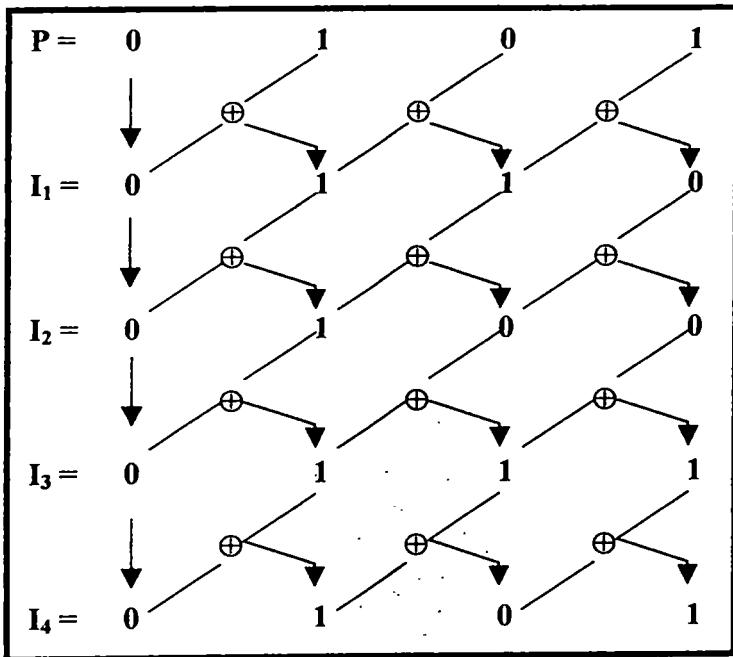
Figure 4.2.1 pictorially represents this technique.



**Figure 4.2.I**  
**Pictorial Representation of the RPPO Technique**

#### 4.2.1 Example

To illustrate the technique, let  $P = 0101$  be a 4-bit source block. Figure 4.2.1.1 shows the generation of the cycle for this sample block. Here it requires 4 iterations to regenerate the source block.



**Figure 4.2.1.1**  
**Pictorial Representation of the RPPO Technique for Source Block  $P = 0101$**

In this way, for different blocks in the plaintext corresponding cycles are formed. If the blocks are taken of the same size, the number of iterations required in forming the cycles will be equal and hence that number of iterations will be required to complete the cycle for the entire stream of bits.

With respect to one single block of bits, any intermediate block during the process of forming the cycle can be considered as the encrypted block. If the total number of iterations required to complete the cycle is  $P$  and the  $i^{\text{th}}$  block is considered to be the encrypted block, then a number of  $(P - i)$  more iterations will be required to decrypt the encrypted block, i.e., to regenerate the source block.

Now, if the process of encryption is considered for the entire stream of bits, then it depends on how the blocks have been formed. Out of the entire stream of bits, different blocks can be formed in two ways:

1. Blocks with equal size
2. Blocks with different sizes.

In the case of blocks with equal length, if for all blocks, intermediate blocks after a fixed number of iterations are considered as the corresponding encrypted blocks. then that very number of iterations will be required for encrypting the entire stream of bits. The key of the scheme will be quite simple, consisting of only two information, one being the fixed block size and the other being the fixed number of iterations for all the blocks used during the encryption. On the other hand, for different source blocks different intermediate blocks may be considered as the corresponding encrypted blocks. For example, the policy may be something like that out of three source blocks  $B_1$ ,  $B_2$ ,  $B_3$  in a source block of bits, the 4<sup>th</sup>, the 7<sup>th</sup> and the 5<sup>th</sup> intermediate blocks respectively are being considered as the encrypted blocks. In such a case, the key of the scheme will become much more complex, which in turn will ensure better security.

In the case of blocks with varying lengths, different blocks will require different numbers of iteration to form the corresponding cycle. So, the LCM value, say,  $P$ . of all these numbers will give the actual number of iterations required to form the cycle for the entire stream. Now, if  $i$  number of iterations are performed to encrypt the entire stream. then a number of  $(P - i)$  more iterations will be required to decrypt the encrypted stream.

#### 4.3 Implementation

In this section, let us consider the same plaintext ( $P$ ): Data Encryption to encrypt it using the RPPO technique. The corresponding stream of bits ( $S$ ) of length 120 bits is as follows:

01000100/01100001/01110100/01100001/11111111/01000101/01101110/01100011/  
01110010/01111001/01110000/01110100/01101001/01101111/01101110

Here “/” is used as the separator between successive bytes.

Blocks can be chosen in any manner. Here we choose blocks to be of varying sizes. Say, following are the different blocks constructed from  $S$ :

$$S_1 = 0100010001 \text{ (10 bits)}$$

$$S_2 = 10000101110100 \text{ (14 bits)}$$

$$S_3 = 0110000111111111 \text{ (16 bits)}$$

$$S_4 = 010001010110111001100011 \text{ (24 bits)}$$

$$S_5 = 01110010 \text{ (8 bits)}$$

$S_6 = 01111001011100000111010001101001$  (32 bits)

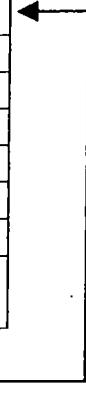
$S_7 = 0110111101101110$  (16 bits)

Tables 4.3.1 to 4.3.7 show the formation of cycles for blocks  $S_1$ ,  $S_2$ ,  $S_3$ ,  $S_4$ ,  $S_5$ ,  $S_6$  and  $S_7$  respectively. Now, for each of the blocks, an arbitrary intermediate block, as indicated in each table, is considered as the encrypted block.

**Table 4.3.1**  
**Formation of Cycle for Block  $S_1$  for RPPO Technique**

<b>Source Block</b>	0100010001
<b>Block (<math>I_{11}</math>) after iteration 1</b>	0111100001
<b>Block (<math>I_{12}</math>) after iteration 2</b>	0101000001
<b>Block (<math>I_{13}</math>) after iteration 3</b>	0110000001
<b>Block (<math>I_{14}</math>) after iteration 4</b>	0100000001
<b>Block (<math>I_{15}</math>) after iteration 5</b>	0111111110
<b>Block (<math>I_{16}</math>) after iteration 6</b>	0101010100
<b>Block (<math>I_{17}</math>) after iteration 7</b>	0110011000
<b>Block (<math>I_{18}</math>) after iteration 8</b>	0100010000
<b>Block (<math>I_{19}</math>) after iteration 9</b>	0111100000
<b>Block (<math>I_{110}</math>) after iteration 10</b>	0101000000
<b>Block (<math>I_{111}</math>) after iteration 11</b>	0110000000
<b>Block (<math>I_{112}</math>) after iteration 12</b>	0100000000
<b>Block (<math>I_{113}</math>) after iteration 13</b>	0111111111
<b>Block (<math>I_{114}</math>) after iteration 14</b>	0101010101
<b>Block (<math>I_{115}</math>) after iteration 15</b>	0110011001
<b>Block (<math>I_{116}</math>) after iteration 16</b> (Source Block)	0100010001

Encrypted Block



**Table 4.3.2**  
**Formation of Cycle for Block S<sub>2</sub> for RPPO Technique**

<b>Source Block</b>	10000101110100
<b>Block (I<sub>21</sub>) after iteration 1</b>	11111001011000
<b>Block (I<sub>22</sub>) after iteration 2</b>	10101110010000
<b>Block (I<sub>23</sub>) after iteration 3</b>	11001011100000
<b>Block (I<sub>24</sub>) after iteration 4</b>	10001101000000
<b>Block (I<sub>25</sub>) after iteration 5</b>	11110110000000
<b>Block (I<sub>26</sub>) after iteration 6</b>	10100100000000
<b>Block (I<sub>27</sub>) after iteration 7</b>	11000111111111
<b>Block (I<sub>28</sub>) after iteration 8</b>	10000101010101
<b>Block (I<sub>29</sub>) after iteration 9</b>	11111001100110
<b>Block (I<sub>210</sub>) after iteration 10</b>	10101110111011
<b>Block (I<sub>211</sub>) after iteration 11</b>	11001011010010
<b>Block (I<sub>212</sub>) after iteration 12</b>	10001101100011
<b>Block (I<sub>213</sub>) after iteration 13</b>	11110110111101
<b>Block (I<sub>214</sub>) after iteration 14</b>	10100100101001
<b>Block (I<sub>215</sub>) after iteration 15</b>	11000111001110
<b>Block (I<sub>216</sub>) after iteration 16</b> <b>(Source Block)</b>	10000101110100

Encrypted Block

**Table 4.3.3**  
**Formation of Cycle for Block S<sub>3</sub> for RPPO Technique**

<b>Source Block</b>	0110000111111111
<b>Block (I<sub>31</sub>) after iteration 1</b>	0100000101010101
<b>Block (I<sub>32</sub>) after iteration 2</b>	0111111001100110
<b>Block (I<sub>33</sub>) after iteration 3</b>	0101010001000100
<b>Block (I<sub>34</sub>) after iteration 4</b>	0110011110000111
<b>Block (I<sub>35</sub>) after iteration 5</b>	0100010100000101
<b>Block (I<sub>36</sub>) after iteration 6</b>	0111100111111001
<b>Block (I<sub>37</sub>) after iteration 7</b>	0101000101010001
<b>Block (I<sub>38</sub>) after iteration 8</b>	0110000110011110
<b>Block (I<sub>39</sub>) after iteration 9</b>	0100000100010100
<b>Block (I<sub>310</sub>) after iteration 10</b>	0111111000011000
<b>Block (I<sub>311</sub>) after iteration 11</b>	0101010000010000
<b>Block (I<sub>312</sub>) after iteration 12</b>	0110011111100000
<b>Block (I<sub>313</sub>) after iteration 13</b>	0100010101000000
<b>Block (I<sub>314</sub>) after iteration 14</b>	0111100110000000
<b>Block (I<sub>315</sub>) after iteration 15</b>	0101000100000000
<b>Block (I<sub>316</sub>) after iteration 16</b> <b>(Source Block)</b>	0110000111111111

Encrypted Block

**Table 4.3.4**  
**Formation of Cycle for Block S<sub>4</sub> for RPPO Technique**

<b>Source Block</b>	010001010110111001100011
<b>Block (L<sub>41</sub>) after iteration 1</b>	011110011011010001000010
<b>Block (L<sub>42</sub>) after iteration 2</b>	01010001001001110000011
<b>Block (L<sub>43</sub>) after iteration 3</b>	011000011100010100000010
<b>Block (L<sub>44</sub>) after iteration 4</b>	010000010111100111111100
<b>Block (L<sub>45</sub>) after iteration 5</b>	011111100101000101010111
<b>Block (L<sub>46</sub>) after iteration 6</b>	010101000110000110011010
<b>Block (L<sub>47</sub>) after iteration 7</b>	011001111011111011101100
<b>Block (L<sub>48</sub>) after iteration 8</b>	010001010010101101001000
<b>Block (L<sub>49</sub>) after iteration 9</b>	011110011100110110001111
<b>Block (L<sub>410</sub>) after iteration 10</b>	010100010111011011110101
<b>Block (L<sub>411</sub>) after iteration 11</b>	011000011010010010100110
<b>Block (L<sub>412</sub>) after iteration 12</b>	010000010011100011000100
<b>Block (L<sub>413</sub>) after iteration 13</b>	011111100010111101111000
<b>Block (L<sub>414</sub>) after iteration 14</b>	010101000011010110101111
<b>Block (L<sub>415</sub>) after iteration 15</b>	011001111101100100110101
<b>Block (L<sub>416</sub>) after iteration 16</b>	010001010110111000100110
<b>Block (L<sub>417</sub>) after iteration 17</b>	011110011011010000111011
<b>Block (L<sub>418</sub>) after iteration 18</b>	01010001001001111010010
<b>Block (L<sub>419</sub>) after iteration 19</b>	011000011100010101100011
<b>Block (L<sub>420</sub>) after iteration 20</b>	010000010111100110111101
<b>Block (L<sub>421</sub>) after iteration 21</b>	011111100101000100101001
<b>Block (L<sub>422</sub>) after iteration 22</b>	010101000110000111001110
<b>Block (L<sub>423</sub>) after iteration 23</b>	011001111011111010001011
<b>Block (L<sub>424</sub>) after iteration 24</b>	010001010010101100001101
<b>Block (L<sub>425</sub>) after iteration 25</b>	011110011100110111110110
<b>Block (L<sub>426</sub>) after iteration 26</b>	010100010111011010100100
<b>Block (L<sub>427</sub>) after iteration 27</b>	011000011010010011000111
<b>Block (L<sub>428</sub>) after iteration 28</b>	010000010111100010000101
<b>Block (L<sub>429</sub>) after iteration 29</b>	011111100010111100000110
<b>Block (L<sub>430</sub>) after iteration 30</b>	010101000011010111110111
<b>Block (L<sub>431</sub>) after iteration 31</b>	011001111101100101010010
<b>Block (L<sub>432</sub>) after iteration 32</b>	010001010110111001100011

**Encrypted Block**

**Table 4.3.5**  
**Formation of Cycle for Block S<sub>5</sub> for RPPO Technique**

<b>Source Block</b>	01110010
<b>Block (I<sub>51</sub>) after iteration 1</b>	01011100
<b>Block (I<sub>52</sub>) after iteration 2</b>	01101000
<b>Block (I<sub>53</sub>) after iteration 3</b>	01001111
<b>Block (I<sub>54</sub>) after iteration 4</b>	01110101
<b>Block (I<sub>55</sub>) after iteration 5</b>	01011001
<b>Block (I<sub>56</sub>) after iteration 6</b>	01101110
<b>Block (I<sub>57</sub>) after iteration 7</b>	01001011
<b>Block (I<sub>58</sub>) after iteration 8</b>	01110010

**Encrypted Block**



**Table 4.3.6**  
**Formation of Cycle for Block S<sub>6</sub> for RPPO Technique**

<b>Source Block</b>	01111001011100000111010001101001
<b>Block (I<sub>61</sub>) after iteration 1</b>	01010001101000000101100001001110
<b>Block (I<sub>62</sub>) after iteration 2</b>	011000010011111100100001110100
<b>Block (I<sub>63</sub>) after iteration 3</b>	0100000111010101000111110100111
<b>Block (I<sub>64</sub>) after iteration 4</b>	01111110100110011110101011000101
<b>Block (I<sub>65</sub>) after iteration 5</b>	01010100111011101011001101111001
<b>Block (I<sub>66</sub>) after iteration 6</b>	01100111010010110010001001010001
<b>Block (I<sub>67</sub>) after iteration 7</b>	01000101100011011100001110011110
<b>Block (I<sub>68</sub>) after iteration 8</b>	01111001000010010111110100010100
<b>Block (I<sub>69</sub>) after iteration 9</b>	01010001111100011010100111100111
<b>Block (I<sub>610</sub>) after iteration 10</b>	01100001010111101100111010111010
<b>Block (I<sub>611</sub>) after iteration 11</b>	01000001100101001000101100101100
<b>Block (I<sub>612</sub>) after iteration 12</b>	01111110111001110000110111001000
<b>Block (I<sub>613</sub>) after iteration 13</b>	01010100101110100000100101110000
<b>Block (I<sub>614</sub>) after iteration 14</b>	01100111001011000000111001011111
<b>Block (I<sub>615</sub>) after iteration 15</b>	01000101110010000000101110010101
<b>Block (I<sub>616</sub>) after iteration 16</b>	01111001011100000000110100011001
<b>Block (I<sub>617</sub>) after iteration 17</b>	0101000110100000000100111101110
<b>Block (I<sub>618</sub>) after iteration 18</b>	0110000100111111111000101001011
<b>Block (I<sub>619</sub>) after iteration 19</b>	0100000111010101010111001110010
<b>Block (I<sub>620</sub>) after iteration 20</b>	01111110100110011001010001011100
<b>Block (I<sub>621</sub>) after iteration 21</b>	01010100111011101110011110010111
<b>Block (I<sub>622</sub>) after iteration 22</b>	01100111010010110100010100011010
<b>Block (I<sub>623</sub>) after iteration 23</b>	01000101100011011000011000010011
<b>Block (I<sub>624</sub>) after iteration 24</b>	01111001000010010000010000011101
<b>Block (I<sub>625</sub>) after iteration 25</b>	0101000111110001111100000010110
<b>Block (I<sub>626</sub>) after iteration 26</b>	0110000101011110101011111100100
<b>Block (I<sub>627</sub>) after iteration 27</b>	01000001100101001100101010111000
<b>Block (I<sub>628</sub>) after iteration 28</b>	01111110111001110111001100101111
<b>Block (I<sub>629</sub>) after iteration 29</b>	01010100101110100101110111001010
<b>Block (I<sub>630</sub>) after iteration 30</b>	01100111001011000110100101110011
<b>Block (I<sub>631</sub>) after iteration 31</b>	01000101110010000100111001011101
<b>Block (I<sub>632</sub>) after iteration 32</b>	01111001011100000111010001101001

**Encrypted Block**

**Table 4.3.7**  
**Formation of Cycle for Block S<sub>7</sub> for RPPO Technique**

<b>Source Block</b>	0110111101101110
<b>Block (I<sub>71</sub>) after iteration 1</b>	0100101001001011
<b>Block (I<sub>72</sub>) after iteration 2</b>	0111001110001101
<b>Block (I<sub>73</sub>) after iteration 3</b>	0101110100001001
<b>Block (I<sub>74</sub>) after iteration 4</b>	0110100111110001
<b>Block (I<sub>75</sub>) after iteration 5</b>	0100111000100001
<b>Block (I<sub>76</sub>) after iteration 6</b>	0111010011000001
<b>Block (I<sub>77</sub>) after iteration 7</b>	0101100010000001
<b>Block (I<sub>78</sub>) after iteration 8</b>	0110111100000001
<b>Block (I<sub>79</sub>) after iteration 9</b>	0100101000000001
<b>Block (I<sub>710</sub>) after iteration 10</b>	0111001111111110
<b>Block (I<sub>711</sub>) after iteration 11</b>	0101110101010100
<b>Block (I<sub>712</sub>) after iteration 12</b>	0110100110011000
<b>Block (I<sub>713</sub>) after iteration 13</b>	0100111011101111
<b>Block (I<sub>714</sub>) after iteration 14</b>	0111010010110101
<b>Block (I<sub>715</sub>) after iteration 15</b>	0101100011011001
<b>Block (I<sub>716</sub>) after iteration 16 (Source Block)</b>	0110111101101110

#### Encrypted Block

As indicated in tables 4.3.1 to 4.3.7, intermediate blocks I<sub>19</sub> (0111100000), I<sub>214</sub> (10100100101001), I<sub>314</sub> (0111100110000000), I<sub>43</sub> (011000011100010100000010), I<sub>57</sub> (01001011), I<sub>625</sub> (010100011110001111100000010110) and I<sub>77</sub> (0101100010000001) are considered as the encrypted blocks, so that these blocks form the encrypted stream as follows:

0111100000/10100100101001/0111100110000000/011000011100010100000010/01001  
 011/0101000111110001111100000010110/0101100010000001, “/” being used as only the separator.

The encrypted stream can be rewritten as the series of bytes as follows:

011.11000/00101001/00101001/01111001/10000000/01100001/11000101/00000010/010  
 01011/01010001/11110001/11111000/00010110/01011000/10000001.

Converting the bytes into the corresponding characters, the following text is obtained as the encrypted text, which is to be transmitted/stored:

$$C = x))y \rightarrow KQ \pm \square X \rightarrow$$

Now, since while encrypting in this case, the source stream is decomposed into sub-streams in a different way than what was done in section 2.3 for the same example, the process of decryption also in this case is much more complicated.

After converting the ciphertext C into a stream of bits, the technique of decomposition into several blocks of bits should follow the same way the source was decomposed. Then for each block the necessary number of iterations is to be performed to get the corresponding source block. For example, to get the source block corresponding to the encrypted block  $I_{19}$ , the same iterations are to be applied  $(16 - 9) = 7$  times because as per the mathematical policy a total of 16 iterations are required to complete the cycle, and as was shown in table 4.3.1, the encrypted block  $I_{19}$  was obtained after a total of 9 iterations. After obtaining all source blocks in this way, they are grouped together to form what would be the source stream of bits, from which the plaintext is achieved.

#### 4.4 Results

Section 4.4.1 shows results of the encryption/decryption time, the number of operations for encryption and decryption, and the chi square value, section 4.4.2 depicts pictorial result of the frequency distribution tests, section 4.4.3 presents results of the comparison with the RSA system.

##### 4.4.1 Result of Encryption/Decryption Time, Total Number of Operations, Chi Square Value

To experiment with the same set of sample files considered earlier, the technique of RPPO has been applied in a cascaded way with block sizes of  $2^n$ , n increasing from 3 to 8. This means that first on the source file, the RPPO encryption technique is applied for blocks with the unique length of 8 bits. On the generated stream of bits, the same technique is applied with blocks with the unique length of 16 bits, and this process continues till the generation of stream of bits for blocks of the unique length of 256 bits. In each case, intermediate blocks generated after only one iteration are considered as target blocks, so that the process of decryption requires much more time and involves much more number of operations than the process of encryption. [36, 44, 46, 55, 56]

Section 4.4.1.1 shows the result on *EXE* files, section 4.4.1.2 shows the result on *COM* files, section 4.4.1.3 shows the result on *DLL* files, section 4.4.1.4 shows the result on *SYS* files and section 4.4.1.5 shows the result on *CPP* files.

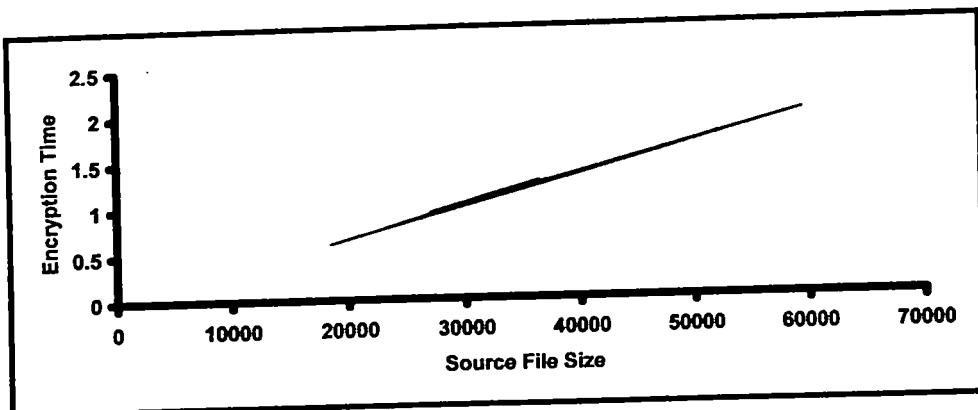
#### **4.4.1.1 Result for *EXE* Files**

Table 4.4.1.1.1 gives the result of implementing the technique on *EXE* files. Ten files have been considered. Their sizes range from 18432 bytes to 59398 bytes. The encryption time ranges from 0.604396 seconds to 1.978022 seconds. The decryption time ranges from 3.956044 seconds to 20.274725 seconds. The number of operations during the process of encryption ranges from 5562057 to 4256157816, whereas the same during the process of decryption ranges from 725704903 to 4630879800. The Chi Square value is observed to be between 20479 and 202973 with the degree of freedom ranging from 248 to 255.

**Table 4.4.1.1.1**  
**Result for *EXE* Files for RPPO Technique**

Source File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	No. of Operations During Encryption	No. of Operations During Decryption	Chi Square Value	Degree of Freedom
<i>TLIB.EXE</i>	37220	1.263736	12.692307	5879407	762483595	146690	255
<i>MAKER.EXE</i>	59398	1.978022	20.274725	899004121	2106446259	166659	255
<i>UNZIP.EXE</i>	23044	0.769231	7.912087	2312980138	2781386689	20479	255
<i>RPPO.EXE</i>	35425	1.208791	12.087912	5562057	725704903	56083	255
<i>PRIME.EXE</i>	37152	1.263736	12.692307	852691357	1608069906	66283	255
<i>TCDEF.EXE</i>	26983	0.934066	3.956044	1739245468	1976720300	43640	254
<i>TRIANGLE.EXE</i>	36385	1.263736	12.362637	2022115101	2758695665	57049	255
<i>PING.EXE</i>	24576	0.824176	8.351648	2886377400	3386006712	142814	248
<i>NETSTAT.EXE</i>	32768	1.098901	11.153846	3475136184	4141308600	202973	255
<i>CLIPBRD.EXE</i>	18432	0.604396	6.318681	4256157816	4630879800	90561	255

A part of the table is diagrammatically represented in figure 4.4.1.1.1, where one graphical relationship is established between the source size and the encryption time for *EXE* files. From the figure, it can be interpreted that there is a tendency that the encryption time changes almost linearly with the size of the source file.



**Figure 4.4.1.1.1**  
**Relationship between Source Size and Encryption Time for**  
**.EXE Files in RPPO Technique**

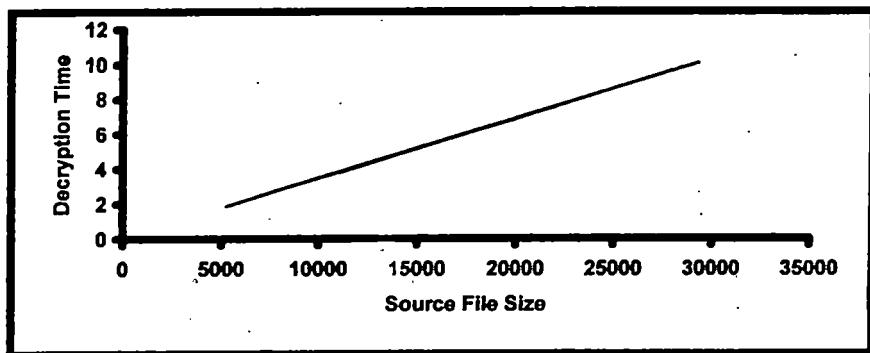
#### 4.4.1.2 Result for COM Files

Table 4.4.1.2.1 gives the result of implementing the technique on different *COM* files. Ten files have been considered. Their sizes range from 5239 bytes to 29271 bytes. The encryption time ranges from 0.219780 seconds to 1.043956 seconds. The decryption time ranges from 1.868132 seconds to 10.054945 seconds. The number of operations during the process of encryption ranges from 827008 to 1972543267, whereas the same during the process of decryption ranges from 107002375 to 2357808445. The Chi Square value is observed to be between 8801 and 80497 with the degree of freedom ranging from 230 to 255.

**Table 4.4.1.2.1**  
**Result for COM Files for RPPO Technique**

Source File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	No. of Operations During Encryption	No. of Operations During Decryption	Chi Square Value	Degree of Freedom
EMSTEST.COM	19664	0.714286	6.758242	3106170	402678963	40182	255
THELP.COM	11072	0.494505	3.791209	471601446	696694860	29624	250
WIN.COM	24791	0.879121	8.516483	738433117	1242100033	53529	252
KEYB.COM	19927	0.769231	6.868132	1329918277	1734700225	61875	255
CHOICE.COM	5239	0.219780	1.868132	827008	107002375	11607	232
DISKCOPY.COM	21975	0.769231	7.527472	128353285	574771009	80497	254
DOSKEY.COM	15495	0.549451	5.329670	652269631	967144870	37393	253
MODE.COM	29271	1.043956	10.054945	1024683457	1619428633	80065	255
MORE.COM	10471	0.384615	3.626374	1721057212	1933794688	8801	230
SYS.COM	18967	0.659341	6.538461	1972543267	2357808445	47097	254

Figure 4.4.1.2.1 is constructed to establish the relationship between the source size and the decryption time for COM files. As it is observed from the figure, there exists a linear relationship between the source file size and the encryption time.



**Figure 4.4.1.2.1**  
**Relationship between Source Size and Decryption Time for .COM Files in RPPO Technique**

#### 4.4.1.3 Result for DLL Files

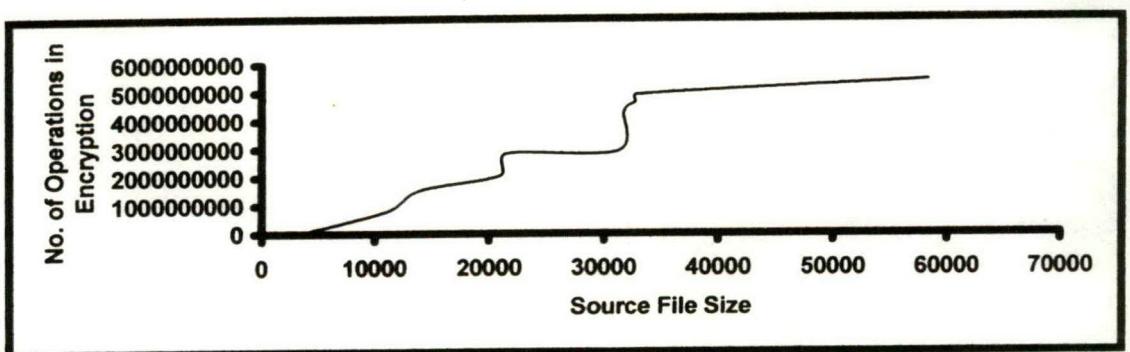
Table 4.4.1.3.1 gives the result of implementing the technique on different DLL files. Ten files have been considered. Their sizes range from 3216 bytes to 58368 bytes. The encryption time ranges from 0.164835 seconds to 2.032967 seconds. The decryption

time ranges from 1.098901 seconds to 19.945055 seconds. The number of operations during the process of encryption ranges from 5176320 to 5415873840, whereas the same during the process of decryption ranges from 5176320 to 6050819424. The Chi Square value is observed to be between 8907 and 534891 with the degree of freedom ranging from 217 to 255.

**Table 4.4.1.3.1**  
**Result for *DLL* Files for RPPO Technique**

Source File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	No. of Operations During Encryption	No. of Operations During Decryption	Chi Square Value	Degree of Freedom
<i>SNMPAPI.DLL</i>	32768	1.153846	11.153846	5176320	5176320	99714	253
<i>KPSHARP.DLL</i>	31744	1.098901	10.879121	788300832	1433655360	291423	254
<i>WINSOCK.DLL</i>	21504	0.714286	7.362637	1545491808	1982667456	101308	252
<i>SPWHPT.DLL</i>	32792	1.153846	11.263736	2061306257	2727632447	320131	255
<i>HIDCI.DLL</i>	3216	0.164835	1.098901	2840151804	2905337271	8907	217
<i>PFPICK.DLL</i>	58368	2.032967	19.945055	2925543648	4112163264	201908	255
<i>NDDEAPI.DLL</i>	14032	0.494505	4.835165	4313768490	4598842899	78677	249
<i>NDDENB.DLL</i>	10976	0.384615	3.791209	4648510953	4871652690	56741	251
<i>ICCCODES.DLL</i>	20992	0.769231	7.197802	4912463472	5339230176	534891	252
<i>KPSCALE.DLL</i>	31232	1.043956	10.714285	5415873840	6050819424	242761	255

The relationship between the source size and the number of operations during encryption for *DLL* files is shown in figure 4.4.1.3.1, from which it is found that there is a tendency that the number of operations increases with the size of the source file considered for encryption, but the relationship is not exactly linear.



**Figure 4.4.1.3.1**  
**Relationship between Source Size and No. of Operations during Encryption for .DLL Files in RPPO Technique**

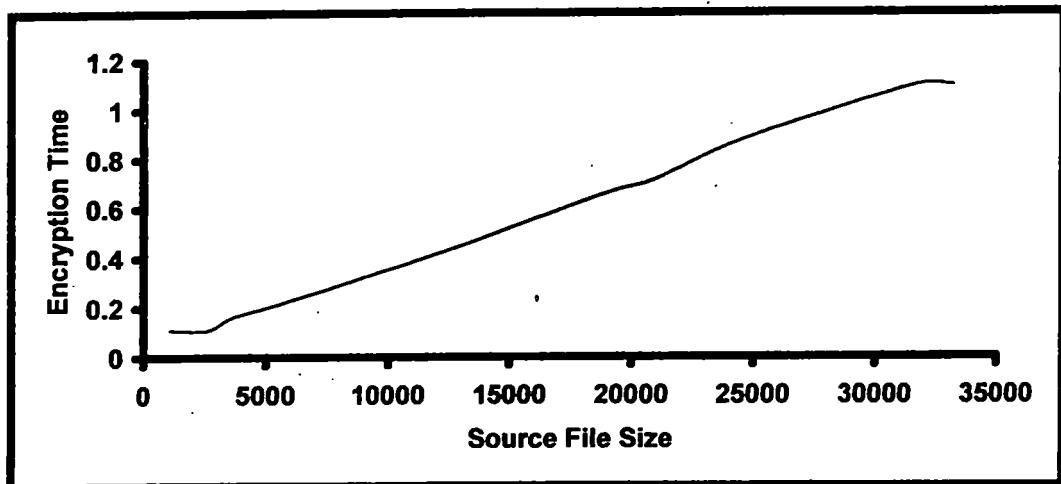
#### 4.4.1.4 Result for SYS Files

Table 4.4.1.4.1 gives the result of implementing the technique on different SYS files. Ten files have been considered. Their sizes range from 1105 bytes to 33191 bytes. The encryption time ranges from 0.109890 seconds to 1.098901 seconds. The decryption time ranges from 0.384615 seconds to 11.318681 seconds. The number of operations during the process of encryption ranges from 412456 to 2859362712, whereas the same during the process of decryption ranges from 53241936 to 2881611334. The Chi Square value is observed to be between 1532 and 170454 with the degree of freedom ranging from 165 to 255.

**Table 4.4.1.4.1**  
**Result for SYS Files for RPPO Technique**

Source File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	No. of Operations During Encryption	No. of Operations During Decryption	Chi Square Value	Degree of Freedom
HIMEM.SYS	33191	1.098901	11.318681	5242678	679877044	115511	255
RAMDRIVE.SYS	12663	0.439560	4.340659	795242728	1052347783	22506	241
USBD.SYS	18912	0.659341	6.428571	1098576273	1483056642	134840	255
CMD640X.SYS	24626	0.879121	8.406593	1551551070	2051961071	56728	255
CMD640X2.SYS	20901	0.714286	7.142857	2139380983	2564200130	54934	255
REDBOOK.SYS	5664	0.219780	1.978022	2636482815	2751631758	29329	230
IFSHLP.SYS	3708	0.164835	1.263736	2771565576	2846537706	7791	237
ASPI2HLP.SYS	1105	0.109890	0.384615	2859362712	2881611334	1532	165
DBLBUFF.SYS	2614	0.109890	0.934066	412456	53241936	4536	215
CCPORT.SYS	31680	1.098901	10.824176	5004450	649057860	170454	255

The linear relationship between the source size and the decryption time for SYS files is shown in figure 4.4.1.4.1. The figure establishes the fact that the decryption time varies almost linearly with the size of the source file.



**Figure 4.4.1.4.1**  
**Relationship between Source Size and Decryption Time for  
 .SYS Files in RPPO Technique**

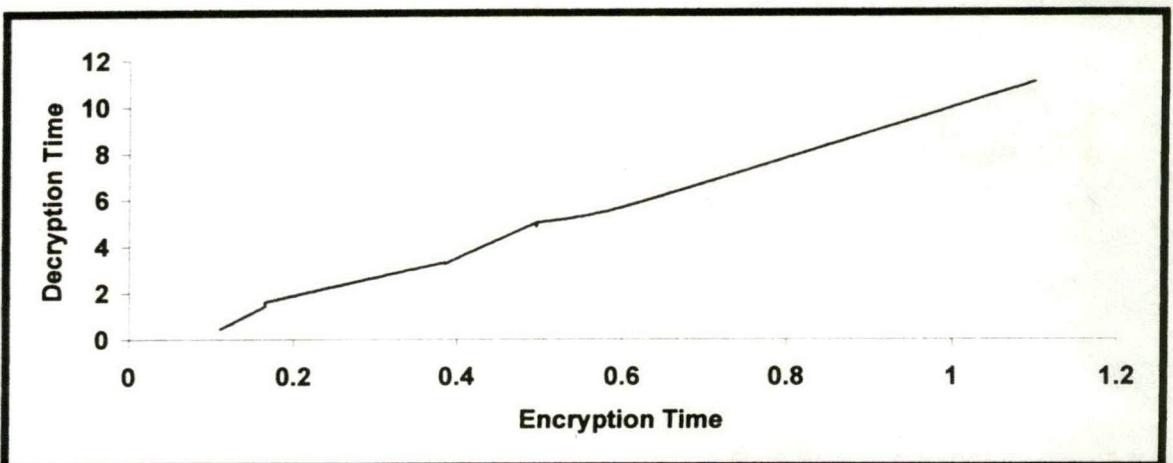
#### 4.4.1.5 Result for *CPP* Files

Table 4.4.5.1 gives the result of implementing the technique on different *CPP* files. Ten files have been considered. Their sizes range from 1257 bytes to 32150 bytes. The encryption time ranges from 0.109890 seconds to 1.098901 seconds. The decryption time ranges from 0.439560 seconds to 11.043956 seconds. The number of operations during the process of encryption ranges from 2641311 to 2665493488, whereas the same during the process of decryption ranges from 342363288 to 2859364141. The Chi Square value is observed to be between 1644 and 74726 with the degree of freedom ranging from 69 to 90.

**Table 4.4.1.5.1**  
**Result for *CPP* Files for RPPO Technique**

Source File	Source Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In Seconds)	No. of Operations During Encryption	No. of Operations During Decryption	Chi Square Value	Degree of Freedom
<i>BRICKS.CPP</i>	16723	0.604396	5.714285	2641311	342363288	53583	88
<i>PROJECT.CPP</i>	32150	1.098901	11.043956	404559709	1057855146	74726	90
<i>ARITH.CPP</i>	9558	0.384615	3.296703	1169177503	1363178286	18910	77
<i>START.CPP</i>	14557	0.494505	5.000000	1398114495	1693626175	31930	88
<i>CHARTCOM.CPP</i>	14080	0.494505	4.835165	1745558760	2031804720	39848	84
<i>BITIO.CPP</i>	4071	0.164835	1.428571	2080545508	2163171184	10608	70
<i>MAINC.CPP</i>	4663	0.164835	1.593407	2177797378	2272262683	9920	83
<i>TTEST.CPP</i>	1257	0.109890	0.439560	2288373428	2313769485	1644	69
<i>DO.CPP</i>	14481	0.494505	5.000000	2320339542	2614521826	31359	88
<i>CAL.CPP</i>	9540	0.384615	3.241758	2665493488	2859364141	16496	77

Figure 4.4.5.1 shows how the decryption time almost linearly changes with the encryption time for a given source size for *CPP* files.

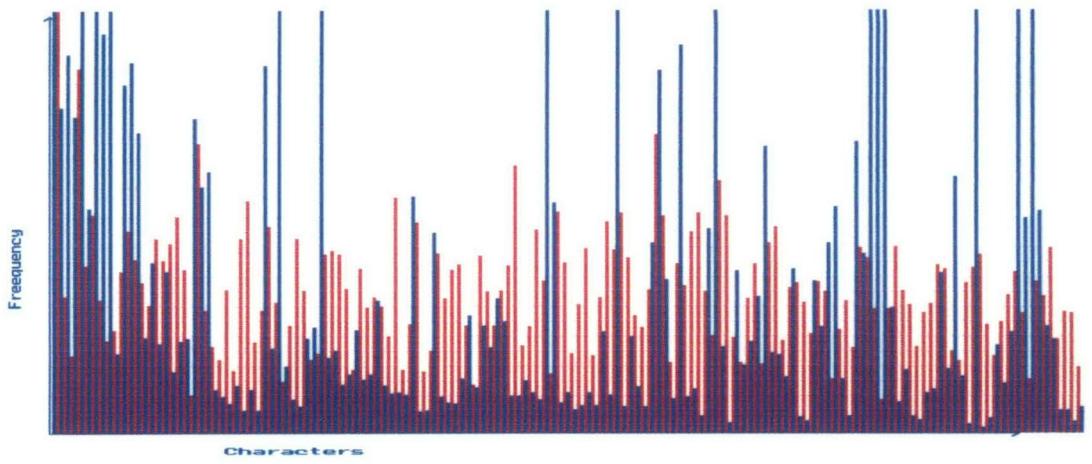


**Figure 4.4.1.5.1**  
**Graphical Relationship between Encryption Time and Decryption Time for .CPP Files in RPPO Technique**

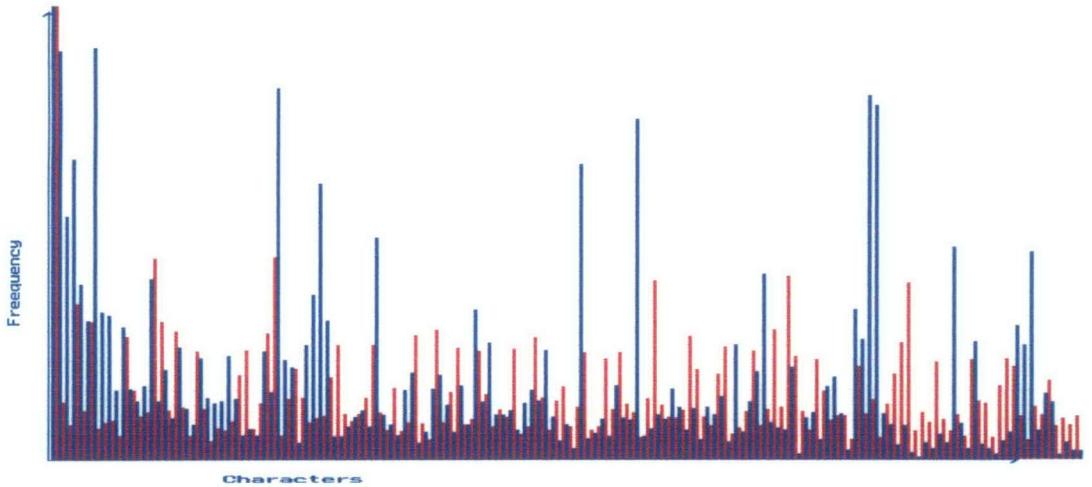
#### 4.4.2 Result of Frequency Distribution Tests

For all sample files, frequency distribution tests for all characters have been performed. It is seen for all cases that the characters in the encrypted files are well distributed, which indicates that the technique proposed here is quite compatible with existing techniques. The red bars represent frequencies of characters in the encrypted file and those in blue color represent frequencies of characters in the source file. The frequencies of characters in encrypted files are evenly distributed. Therefore the source and the corresponding encrypted file are heterogeneous in nature. Hence it can be interpreted that through the proposed technique, a good quality of encryption is obtained.

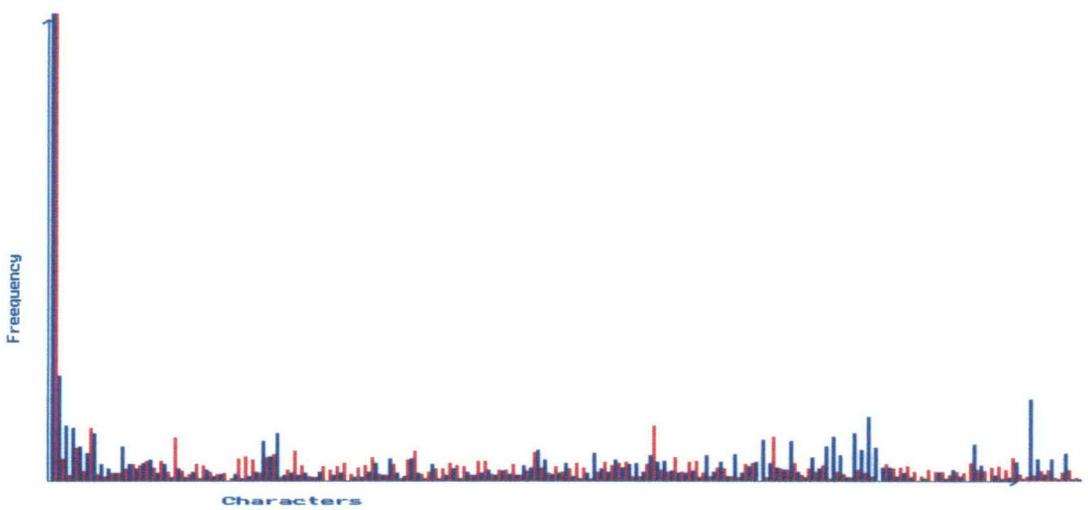
Each of the figures from 4.4.2.1 to 4.4.2.5 exhibits frequency distribution for different files, one from each category.



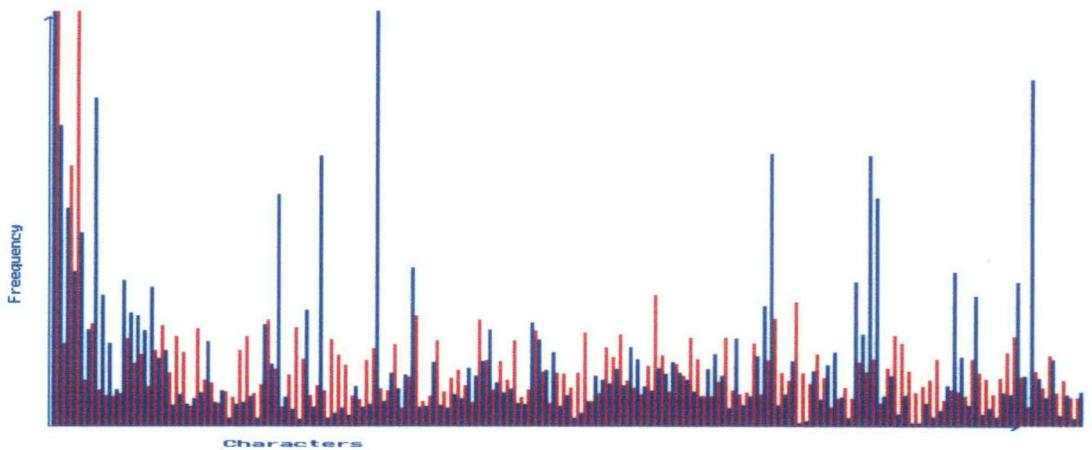
**Figure 4.4.2.1**  
**Frequency Distribution between TLIB.EXE and its Encrypted File for RPPO Technique**



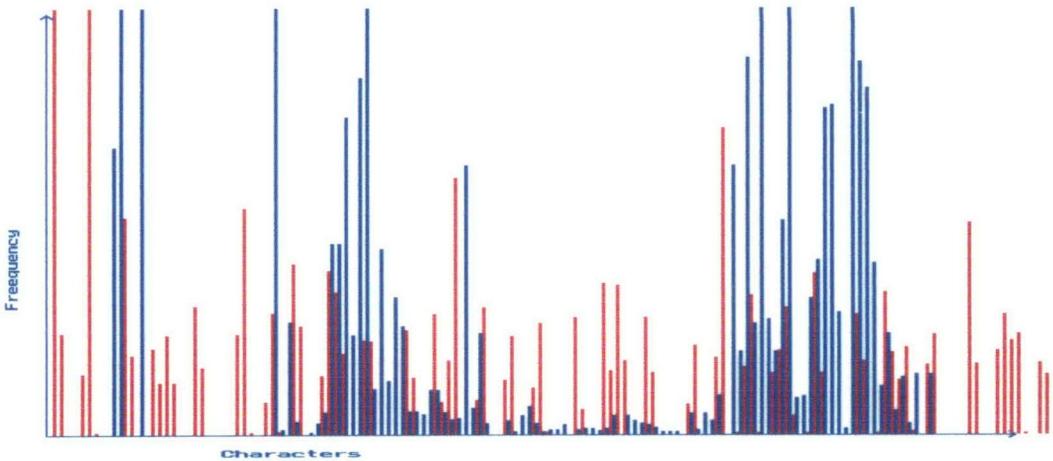
**Figure 4.4.2.2**  
**Frequency Distribution between KEYB.COM and its Encrypted File for RPPO Technique**



**Figure 4.4.2.3**  
**Frequency Distribution between HIDCI.DLL and its Encrypted File for RPPO Technique**



**Figure 4.4.2.4**  
**Frequency Distribution between HIMEM.SYS and its Encrypted File for RPPO Technique**



**Figure 4.4.2.5**  
**Frequency Distribution between START.CPP and its Encrypted File for RPPO Technique**

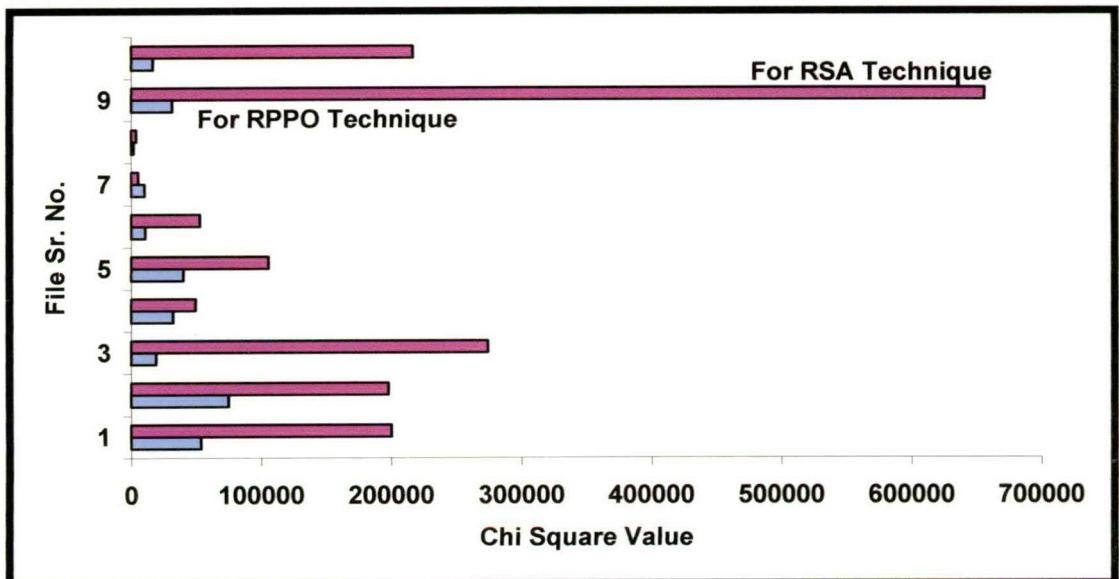
#### 4.4.3 Comparison with RSA Technique

The comparison between the proposed RPPO technique and the existing RSA system has been performed in terms of chi square values between .CPP sample files and respective encrypted files implementing both the techniques. Table 4.4.3.1 enlists the result. The same ten sample files have been considered. The Chi Square value between each of these files and the corresponding encrypted file using the proposed RPPO technique ranges from 1644 to 74726, whereas the same between each of the source files and the corresponding encrypted file using the existing RSA technique ranges from 3652 to 655734. The degree of freedom for these files ranges from 69 to 90 [40, 41].

**Table 4.4.3.1**  
**Comparison of Chi Square Values between RPPO and RSA**

Source File (1)	Source Size	Encrypted File using RPPO Technique (2)	Encrypted File using RSA Technique (3)	Chi Square Value between (1) and (2)	Chi Square Value between (1) and (3)	Degree of Freedom
BRICKS.CPP	16723	FOX1.CPP	CPP1.CPP	53583	200221	88
PROJECT.CPP	32150	FOX2.CPP	CPP2.CPP	74726	197728	90
ARITH.CPP	9558	FOX3.CPP	CPP3.CPP	18910	273982	77
START.CPP	14557	FOX4.CPP	CPP4.CPP	31930	49242	88
CHARTCOM.CPP	14080	FOX5.CPP	CPP5.CPP	39848	105384	84
BITIO.CPP	4071	FOX6.CPP	CPP6.CPP	10608	52529	70
MAINC.CPP	4663	FOX7.CPP	CPP7.CPP	9920	4964	83
TTEST.CPP	1257	FOX8.CPP	CPP8.CPP	1644	3652	69
DO.CPP	14481	FOX9.CPP	CPP9.CPP	31359	655734	88
CAL.CPP	9540	FOX10.CPP	CPP10.CPP	16496	216498	77

Figure 4.4.3.1 presents the comparison graphically. Here red horizontal bars stand for Chi Square values for the RSA technique and blue horizontal bars stand for those for the proposed RPPO technique.



**Figure 4.4.3.1**  
**Comparison between Proposed RPPO Technique and Existing RSA Technique**

## **4.5 Analysis and Conclusion including Comparison with RPSP, TE**

On the basis of the practical implementation, the proposed RPPO technique has been assessed in comparison with the RPSP and the TE techniques, discussed in the last two chapters, in section 4.5.1. The total number of iterations required to complete the cycle, i.e., to regenerate the source block, follows a certain mathematical policy. Here first of all this policy has been presented in section 4.5.2. Section 4.5.3 presents the proof of the finiteness in regenerating the source block. Section 4.5.4 analyzes the results obtained for different sample files from different perspectives, and also draws a conclusion on the technique.

### **4.5.1 Comparative Analysis with RPSP and TE Techniques**

The average of all the Chi Square values between all fifty sample files and their corresponding encrypted files, encrypted using the RPPO technique, is observed to be the maximum so far. Table 4.5.1.1 shows these results. From the table, it is observed that this value is 85350.94, comparing to 64188.04, obtained for the TE technique, and 10701.70, obtained for the RPSP technique. The average encryption time, 0.73186806 seconds, is the lowest, and the average decryption time, 7.03076904 seconds, is lesser than that for the RPSP technique but much more than that for the TE technique [48, 52]

In chapter 10, graphical comparisons have been drawn.

**Table 4.5.1.1  
Average Encryption/Decryption Time and Chi Square Value obtained in  
RPSP, TE, RPPO Techniques**

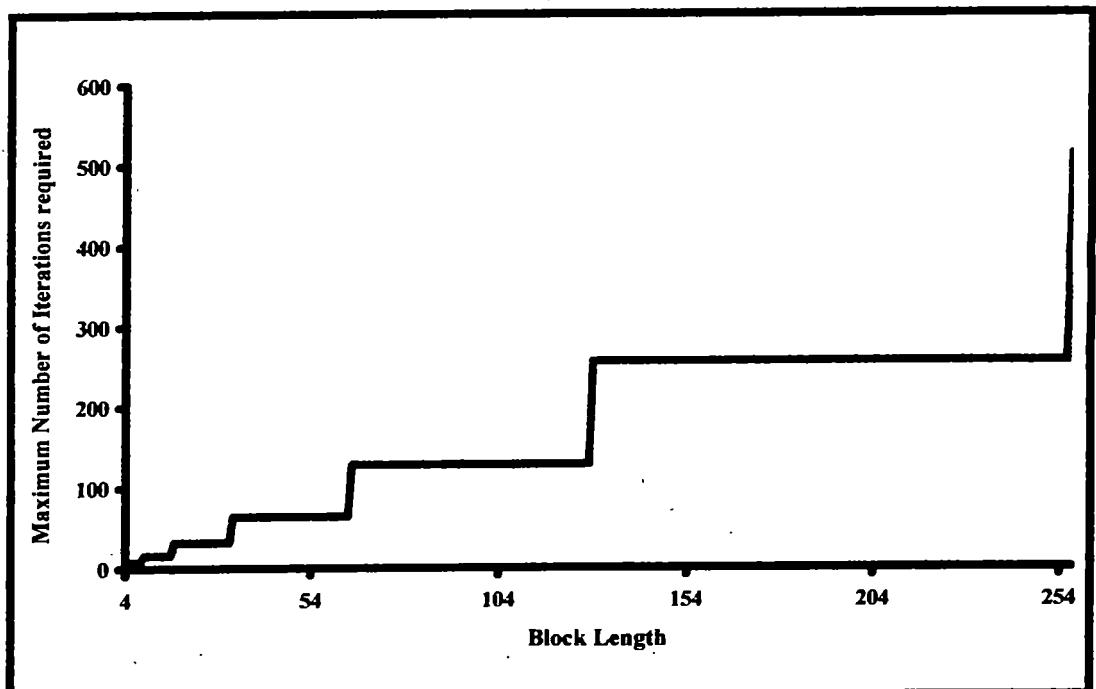
<b>Proposed Technique</b>	<b>Average Encryption Time</b>	<b>Average Decryption Time</b>	<b>Average Chi Square Value</b>	<b>Average Degree of Freedom</b>
<b>RPSP</b>	<b>8.75713800</b>	<b>8.73955200</b>	<b>10701.70</b>	<b>214</b>
<b>TE</b>	<b>0.86703290</b>	<b>0.94175818</b>	<b>64188.04</b>	
<b>RPPO</b>	<b>0.73186806</b>	<b>7.03076904</b>	<b>85350.94</b>	

### **4.5.2 Formation of Cycle**

The maximum number of iterations,  $I_{max}$ , required to complete a cycle for a certain block of bits of any length, say,  $N$ , can be evaluated using the following mathematical policy [46, 49]:

$$\begin{aligned}
 I_{\max} &= N, \text{ if } N = 2^p, p \text{ being a finite positive integer;} \\
 &= 2^{p+1}, \text{ if } 2^p < N < 2^{p+1}, p \text{ being a finite positive integer.}
 \end{aligned}$$

Table 4.5.2.1 shows values of  $I_{\max}$  corresponding to different block lengths. In table 4.5.2.1, the value of  $N$  has been considered in the range of 4 to 259. When  $N$  is 4, the value of  $I_{\max}$  is 4. For  $N$  ranging from 5 to 8,  $I_{\max}$  is 8. For  $N$  ranging from 9 to 16,  $I_{\max}$  is 16. In the range of values of 17 to 32 of  $N$ ,  $I_{\max}$  is 32. The value of  $I_{\max}$  is 64 as  $N$  ranges from 33 to 64. For  $N$  ranging from 65 to 128,  $I_{\max}$  is 128. If  $N$  is in the range of 129 to 256,  $I_{\max}$  is 256. Finally, in the range of 257 to 259 of  $N$ , the value of  $I_{\max}$  is 512.



**Figure 4.5.2.1**  
**Diagrammatic Representation between  $N$  and  $I_{\max}$**

Figure 4.5.2.1 depicts the diagrammatic relationship between  $N$  and  $I_{\max}$ . The figure indicates that the value of  $I_{\max}$  does not vary linearly with  $N$ .  $I_{\max}$  is non-decreasing with the increment of  $N$ .

**Table 4.5.2.1**  
**Values of  $I_{max}$  for Different Block Lengths (N)**

N	$I_{max}$	N	$I_{max}$	N	$I_{max}$	N	$I_{max}$	N	$I_{max}$	N	$I_{max}$	N	$I_{max}$	N	$I_{max}$	N	$I_{max}$
4	4	36	64	68	128	100	128	132	256	164	256	196	256	228	256		
5	8	37	64	69	128	101	128	133	256	165	256	197	256	229	256		
6	8	38	64	70	128	102	128	134	256	166	256	198	256	230	256		
7	8	39	64	71	128	103	128	135	256	167	256	199	256	231	256		
8	8	40	64	72	128	104	128	136	256	168	256	200	256	232	256		
9	16	41	64	73	128	105	128	137	256	169	256	201	256	233	256		
10	16	42	64	74	128	106	128	138	256	170	256	202	256	234	256		
11	16	43	64	75	128	107	128	139	256	171	256	203	256	235	256		
12	16	44	64	76	128	108	128	140	256	172	256	204	256	236	256		
13	16	45	64	77	128	109	128	141	256	173	256	205	256	237	256		
14	16	46	64	78	128	110	128	142	256	174	256	206	256	238	256		
15	16	47	64	79	128	111	128	143	256	175	256	207	256	239	256		
16	16	48	64	80	128	112	128	144	256	176	256	208	256	240	256		
17	32	49	64	81	128	113	128	145	256	177	256	209	256	241	256		
18	32	50	64	82	128	114	128	146	256	178	256	210	256	242	256		
19	32	51	64	83	128	115	128	147	256	179	256	211	256	243	256		
20	32	52	64	84	128	116	128	148	256	180	256	212	256	244	256		
21	32	53	64	85	128	117	128	149	256	181	256	213	256	245	256		
22	32	54	64	86	128	118	128	150	256	182	256	214	256	246	256		
23	32	55	64	87	128	119	128	151	256	183	256	215	256	247	256		
24	32	56	64	88	128	120	128	152	256	184	256	216	256	248	256		
25	32	57	64	89	128	121	128	153	256	185	256	217	256	249	256		
26	32	58	64	90	128	122	128	154	256	186	256	218	256	250	256		
27	32	59	64	91	128	123	128	155	256	187	256	219	256	351	256		
28	32	60	64	92	128	124	128	156	256	188	256	220	256	252	256		
29	32	61	64	93	128	125	128	157	256	189	256	221	256	253	256		
30	32	62	64	94	128	126	128	158	256	190	256	222	256	254	256		
31	32	63	64	95	128	127	128	159	256	191	256	223	256	255	256		
32	32	64	64	96	128	128	128	160	256	192	256	224	256	256	256		
33	64	65	128	97	128	129	256	161	256	193	256	225	256	257	512		
34	64	66	128	98	128	130	256	162	256	194	256	226	256	258	512		
35	64	67	128	99	128	131	256	163	256	195	256	227	256	259	512		

### 4.5.3 Proof of the Finiteness in Re-generating Source Block

This section only presents the proof through the empirical evidences.

#### 4.5.3.1 Proof for Block Size of 2 Bits

Consider a 2-bit block  $P = AB$ . Then the block after the first iteration is  $Y^2_1 = A [A \oplus B]$ . Accordingly, the block after the second iteration is  $Y^2_2 = A [A \oplus (A \oplus B)] = AB$ , which is the source block itself. Hence after 2 iterations, the source block is regenerated.

#### 4.5.3.2 Proof for Block Size of 3 Bits

Consider a 3-bit block  $Q = ABC$ , by adding an extra bit (C) to the block P considered in case 1.

Then after the first iteration, the block  $Y^3_1 = A [A \oplus B] [(A \oplus B) \oplus C]$  is obtained.

Through the second iteration, the block  $Y^3_2 = A [A \oplus (A \oplus B)] [(A \oplus (A \oplus B)) \oplus (A \oplus B) \oplus C]$  is generated.

Through the third iteration, the block  $Y^3_3 = A [A \oplus (A \oplus (A \oplus B))] [(A \oplus (A \oplus (A \oplus B))) \oplus ((A \oplus (A \oplus B)) \oplus (A \oplus B) \oplus C)]$  is generated.

Finally, through the fourth iteration, the block  $Y^3_4 = A [A \oplus (A \oplus (A \oplus (A \oplus B)))] [A \oplus (A \oplus (A \oplus (A \oplus B))) \oplus (A \oplus (A \oplus (A \oplus B))) \oplus ((A \oplus (A \oplus B)) \oplus (A \oplus B) \oplus C)]$  is generated, which is nothing but ABC, the source block Q.

Hence after 4 iterations the source block is regenerated.

#### 4.5.3.3 Proof for Block Size of 4 Bits

Consider a 4-bit block  $R = ABCD$ , by adding an extra bit (D) to the block Q considered in case 2.

Then the block generated after the first iteration is  $Y^4_1 = A [A \oplus B] [(A \oplus B) \oplus C] [((A \oplus B) \oplus C) \oplus D]$ .

The block generated after the second iteration is  $Y_2^4 = A [A \oplus (A \oplus B)] [(A \oplus (A \oplus B)) \oplus ((A \oplus B) \oplus C)] [((A \oplus (A \oplus B)) \oplus ((A \oplus B) \oplus C)) \oplus (((A \oplus B) \oplus C) \oplus D)].$

The block generated after the third iteration is  $Y_3^4 = A [A \oplus (A \oplus (A \oplus B))] [(A \oplus (A \oplus (A \oplus B))) \oplus ((A \oplus (A \oplus B)) \oplus ((A \oplus B) \oplus C))] [((A \oplus (A \oplus (A \oplus B))) \oplus ((A \oplus (A \oplus B)) \oplus ((A \oplus B) \oplus C))) \oplus (((A \oplus (A \oplus B)) \oplus ((A \oplus B) \oplus C)) \oplus (((A \oplus B) \oplus C) \oplus D))].$

Finally, the block generated after the fourth iteration is  $Y_4^4 = A [A \oplus (A \oplus (A \oplus (A \oplus B)))] [(A \oplus (A \oplus (A \oplus (A \oplus B)))) \oplus ((A \oplus (A \oplus (A \oplus B))) \oplus ((A \oplus (A \oplus B)) \oplus ((A \oplus B) \oplus C)))] [((A \oplus (A \oplus (A \oplus (A \oplus B)))) \oplus ((A \oplus (A \oplus (A \oplus B))) \oplus ((A \oplus (A \oplus B)) \oplus ((A \oplus B) \oplus C)))) \oplus (((A \oplus (A \oplus (A \oplus B))) \oplus ((A \oplus (A \oplus B)) \oplus ((A \oplus B) \oplus C))) \oplus (((A \oplus (A \oplus B)) \oplus ((A \oplus B) \oplus C)) \oplus (((A \oplus B) \oplus C) \oplus D)))]$ , which is nothing but ABCD, the source block R.

Hence after 4 iterations the source block is regenerated.

In this manner, the finiteness of the number of iterations to regenerate the source block can be proved for the source block of any length.

#### 4.5.4 A Conclusive Analysis of Different Results Obtained

The encryption/decryption time is related to the size of the source file, and, being a bit-level application, it does not depend on the type of the file. On the other hand, the chi square value is entirely file-dependent.

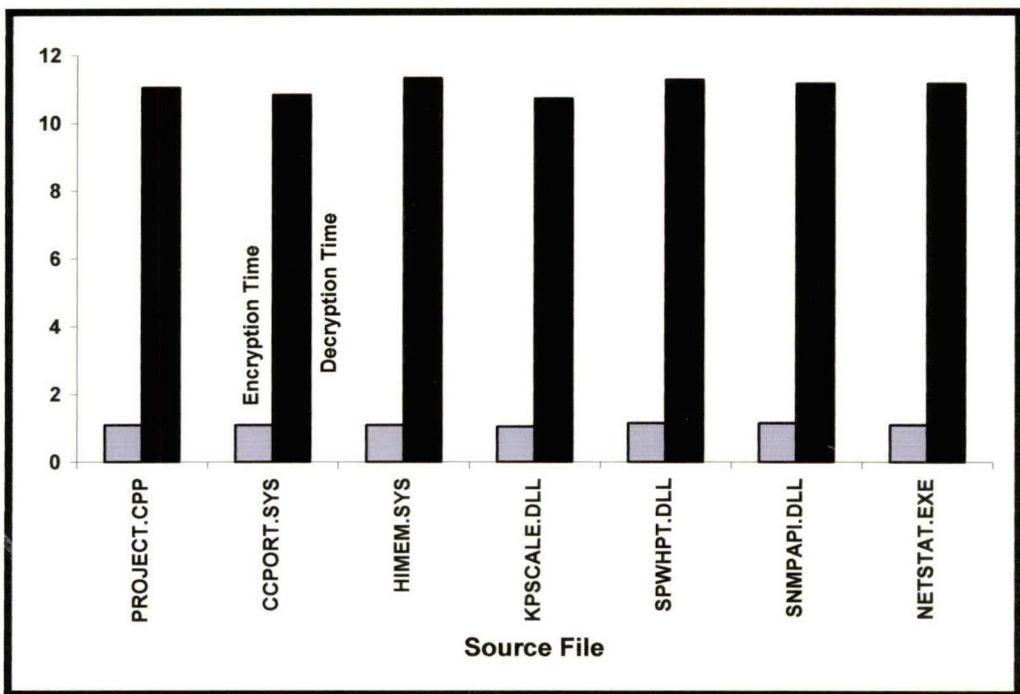
Table 4.5.4.1 considers results for some files with almost same sizes but of different types. Here seven files have been considered. Their sizes range from 31232 bytes to 33191 bytes. It is observed from the table that encryption times are highly compatible to each other, ranging from 1.043956 seconds to 1.153846. Decryption times are also lying in the small range of 10.714285 seconds to 11.318681 seconds. But different Chi Square values are in the much bigger range of 74726 to 320131.

**Table 4.5.4.1**  
**Result of Encryption/Decryption Time and Chi Square value for**  
**Different Types of Files of Almost Same Sizes**

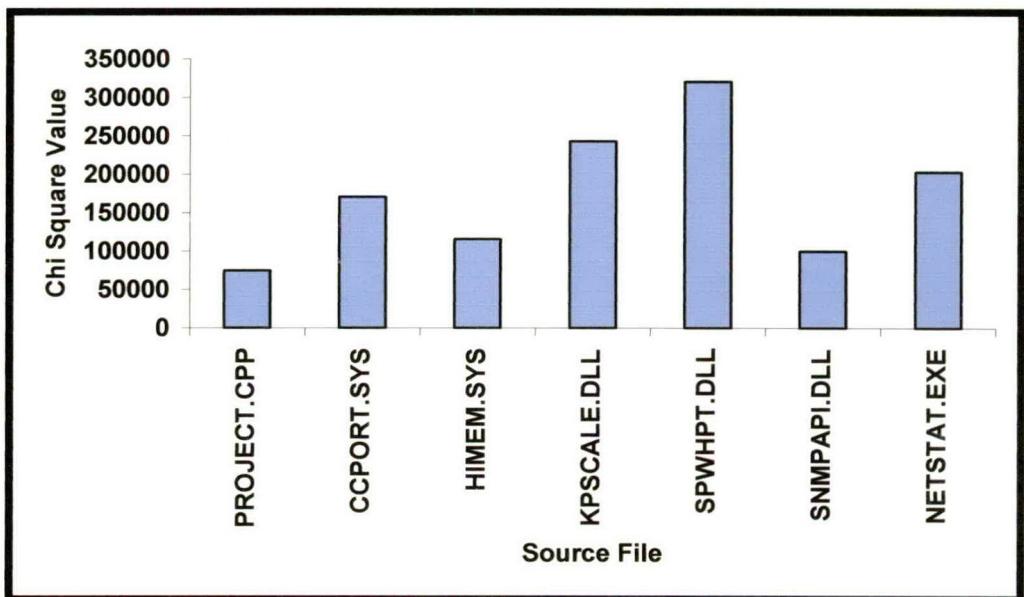
File Name	File Size (In Bytes)	Encryption Time (In Seconds)	Decryption Time (In seconds)	Chi Square Value	Degree of Freedom
<i>PROJECT.CPP</i>	32150	1.098901	11.043956	74726	90
<i>CCPORT.SYS</i>	31680	1.098901	10.824176	170454	255
<i>HIMEM.SYS</i>	33191	1.098901	11.318681	115511	255
<i>KPSCALE.DLL</i>	31232	1.043956	10.714285	242761	255
<i>SPWHPT.DLL</i>	32792	1.153846	11.263736	320131	255
<i>SNMPAPI.DLL</i>	32768	1.153846	11.153846	99714	253
<i>NETSTAT.EXE</i>	32768	1.098901	11.153846	202973	255

Figure 4.5.4.1 exhibits the sameness of encryption/decryption times for the files considered in table 4.5.4.1. Here gray vertical pillars stand for different encryption times and black vertical pillars stand for different decryption times. From the figure, it is observed that gray pillars are almost of the same height and the same is true for black pillars also. This indicates that whatever be the types of files considered, being of almost same size, encryption/decryption times do not differ too much.

Figure 4.5.4.2 establishes the file-dependency of the chi square value. Here it is observed that although files are of almost similar sizes, different vertical pillars are of different lengths. This indicates that Chi Square values are related to contents of files [44].

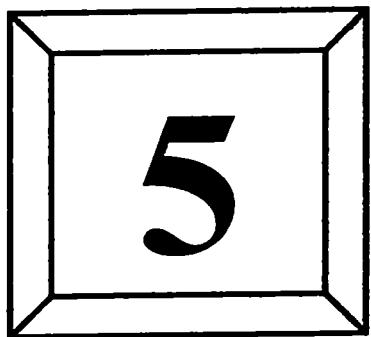


**Figure 4.5.4.1**  
**The Sameness in Encryption/Decryption Time for Files of Almost Similar Sizes**



**Figure 4.5.3.2**  
**The File-Dependency in Chi Square Value for Files of Almost Similar Sizes**

Results obtained from the frequency distribution tests indicate that the encrypted characters are well distributed. Figure 4.4.3.1 suggests that the performance of the RPPO technique in terms of the chi square value is not as good as of the existing RSA technique. But the real strength of this proposed technique, like the other proposed techniques, lies on the formation of a long key space, which can be achieved by constructing blocks of varying sizes, and by arbitrary choice of a block as the encrypted block from any of the intermediate blocks generated during the formation of the cycle. The RPPO encryption policy is expected to ensure a highly satisfactory performance mainly due to the level of flexibility it offers in encrypting a file.



**Encryption Through  
Recursive Positional Modulo-2  
Substitution (RPMS) Technique**

<u>Contents</u>	<u>Pages</u>
<b>5.1      Introduction</b>	<b>166</b>
<b>5.2      The Scheme</b>	<b>167</b>
<b>5.3      Implementation</b>	<b>174</b>
<b>5.4      Results</b>	<b>178</b>
<b>5.5      Analysis and Conclusion including Comparison with RPSP, TE, RPPO</b>	<b>193</b>

## 5.1 Introduction

Unlike the RPSP and the RPPO techniques, the Recursive Positional Modulo-2 Substitution (RPMS) technique is designed in such a manner that neither any cycle is formed nor the process of decryption is the same as that of the encryption.

Unlike the RPSP technique, here there is no any positional orientation of bits. In fact, through a generating function a new block is generated and the function is such that, unlike the generating function used in the RPSP technique, if an attempt is made to form a cycle it may require different number of iterations for two blocks of the same length. The generating function in the RPSP technique is related only with different bit-positions in a block, not with bits, so that any block of a fixed size requires the same number of iterations to form the cycle. But the generating function in the RPMS technique is directly related with different bits.

Unlike the RPPO technique discussed in chapter 4 and the TE technique discussed in chapter 3, here in the RPMS technique there is no application of Boolean algebra during encryption as well as decryption. During encryption, the decimal equivalent of the block of bits under consideration is one integral value from which the recursive modulo-2 operation starts. The modulo-2 operation is performed to check if the integral value is even or odd. Then the position of that integral value in the series of natural even or odd numbers is evaluated. The same process is started again with this positional value. Recursively this process is carried out to a finite number of times, which is exactly the length of the source block. After each modulo-2 operation, 0 or 1 is pushed to the output stream in MSB-to-LSB direction, depending on the fact whether the integral value is even or odd respectively. To generate the source code during decryption, bits in the target block are to be considered along LSB-to-MSB direction, where obviously 0 stands for even and 1 stands for odd. Following the same logic in reverse manner we are to reach to the MSB, after which we get an integral value, the binary equivalent of which is the source block [38, 42, 46, 47, 50, 54].

Section 5.2 is a detailed discussion on the scheme. Since the technique is an asymmetric one, the techniques of the scheme are implemented on a sample plaintext in section 5.3. Section 5.4 shows the results of applying this technique on the same set of

files that were considered earlier. The analysis of the RPMS technique from different angles on the basis of the results presented in section 5.4 is done in section 5.5.

## 5.2 The Scheme

Since the technique proposed is an asymmetric one, the scheme for encryption and that for decryption are being discussed separately along with relevant examples in sections 5.2.1 and 5.2.2.

### 5.2.1 The Encryption

A stream of bits is considered as the plaintext. Like in other proposed techniques, the plaintext is divided into a finite number of blocks, each having a finite fixed number of bits. The RPMS is then applied for each of the blocks.

For each block  $S = s_0 s_1 s_2 s_3 s_4 \dots s_{L-1}$  of length L bits, the following technique 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).

**Step 1:** Corresponding to the source block  $S = s_0 s_1 s_2 s_3 s_4 \dots s_{L-1}$ , evaluate the equivalent decimal integer,  $D_L$ .

**Step 2:** Apply step 3 and step 4 exactly L number of times, for the values of the variable P ranging from 0 to  $(L-1)$  increasing by 1 after each execution of the loop.

**Step 3:** Apply modulo-2 operation on  $D_{L-P}$  to check if  $D_{L-P}$  is even or odd.

**Step 4:** If  $D_{L-P}$  is found to be even, compute  $D_{L-P-1} = D_{L-P} / 2$ , where  $D_{L-P}$  is its position in the series of natural even numbers. Assign  $t_P = 0$ .

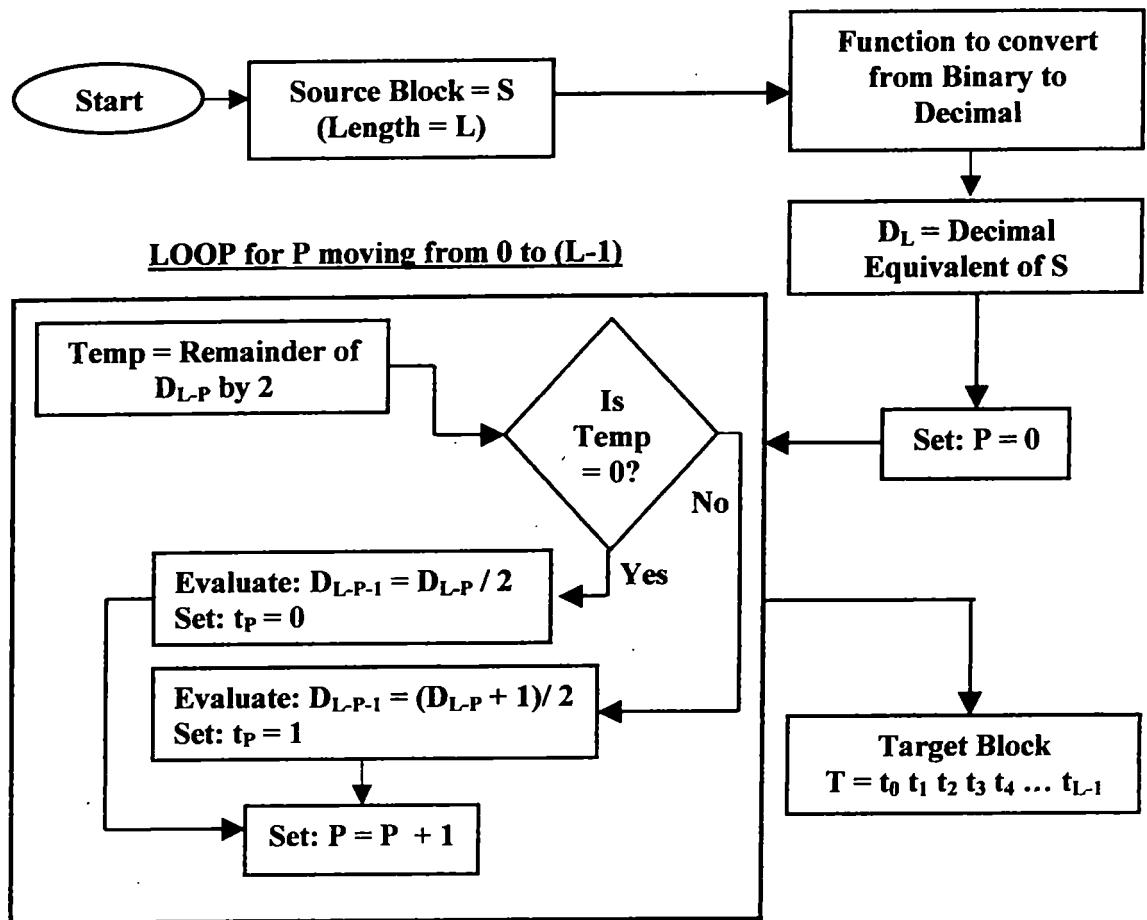
If  $D_{L-P}$  is found to be odd, compute  $D_{L-P-1} = (D_{L-P} + 1) / 2$ , where  $D_{L-P-1}$  is its position in the series of natural odd numbers. Assign  $t_P = 1$ .

**Step 5:** With the values of all the  $t_P$ 's being available, p ranging from 0 to  $(L-1)$ ,  $T = t_0 t_1 t_2 t_3 t_4 \dots t_{L-1}$  constructs the target block corresponding to  $S = s_0 s_1 s_2 s_3 s_4 \dots s_{L-1}$ .

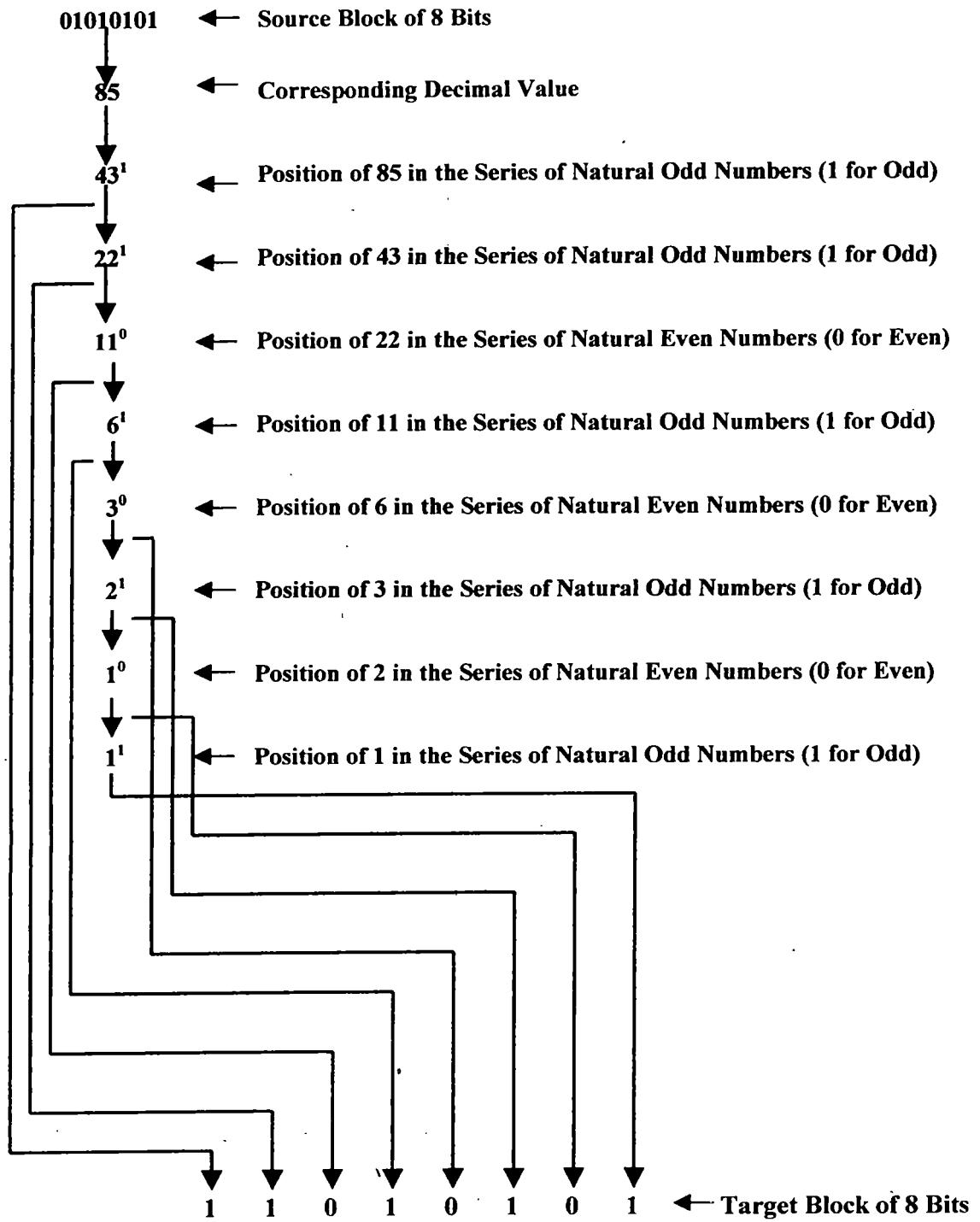
Figure 5.2.1.1 shows the pseudocode for this approach and the same has been presented through a flow diagram in figure 5.2.1.2.

```
Evaluate:  $D_L$ , the decimal equivalent, corresponding to the source  
block  $S = s_0 s_1 s_2 s_3 s_4 \dots s_{L-1}$ .  
Set:  $P = 0$ .  
LOOP:  
    Evaluate: Temp = Remainder of  $D_{L-P} / 2$ .  
    If Temp = 0  
        Evaluate:  $D_{L-P-1} = D_{L-P} / 2$ .  
        Set:  $t_p = 0$ .  
    Else  
        If Temp = 1  
            Evaluate:  $D_{L-P-1} = (D_{L-P} + 1) / 2$ .  
            Set:  $t_p = 1$ .  
        Set:  $P = P + 1$ .  
        If ( $P > (L - 1)$ )  
            Exit.  
ENDLOOP
```

**Figure 5.2.1.1**  
**Pseudocode of the RPMS Scheme to encrypt a Source Block**



**Figure 5.2.1.2**  
**Flow Diagram of the RPMS Scheme to generate Target Block**



**Figure 5.2.1.3**  
**Formation of Target Block for Source Block 01010101 for RPMS Technique**

Consider a source block of bits  $S=01010101$  of length  $L=8$ . Now, following the technique shown in figures 5.2.1.1 and 5.2.1.2 the encrypted stream will be 11010101

and the corresponding flow diagram is shown in figure 5.2.1.3. In this figure, in any intermediate step after getting the position of an odd or even number in the series of natural numbers, either 1 or 0 is written on top of that position for indicating odd and even respectively. This bit may be termed as the **modulo-2 check bit**. Hence the encrypted block is  $T=11010101$ .

After generating all the target blocks, these are to be composed together in the same way the source stream was decomposed into different source blocks. As the result of this, the target stream of bits is obtained, the text corresponding to which is the encrypted message.

### 5.2.2 The Decryption

The initial step of work to be performed in the process of decryption is the same for all the proposed techniques.

In this initial stage, the encrypted message is to be converted into the corresponding original (source) stream of bits and then this stream is to be decomposed into a finite set of blocks, each consisting of a finite set of bits. Now, during this process of decomposition, the way by which the source stream was decomposed during encryption is to be followed, so that corresponding to each block, which is effectively the target block, the source block can be generated.

For each target block  $T = t_0 t_1 t_2 t_3 t_4 \dots t_{L-1}$  of length  $L$  bits, the following scheme is followed in a stepwise manner to generate the source block  $S = s_0 s_1 s_2 s_3 s_4 \dots s_{L-1}$  of the same length ( $L$ ).

**Step 1:** Set  $P = L - 1$  and  $T = 1$ .

**Step 2:** Repeat step 3 and step 4 for the value of  $P$  ranging from  $(L-1)$  to 0.

**Step 3:** If  $t_P = 0$

$T = T^{\text{th}}$  even number in the series of natural even numbers;

If  $t_P = 1$

$T = T^{\text{th}}$  odd number in the series of natural even numbers.

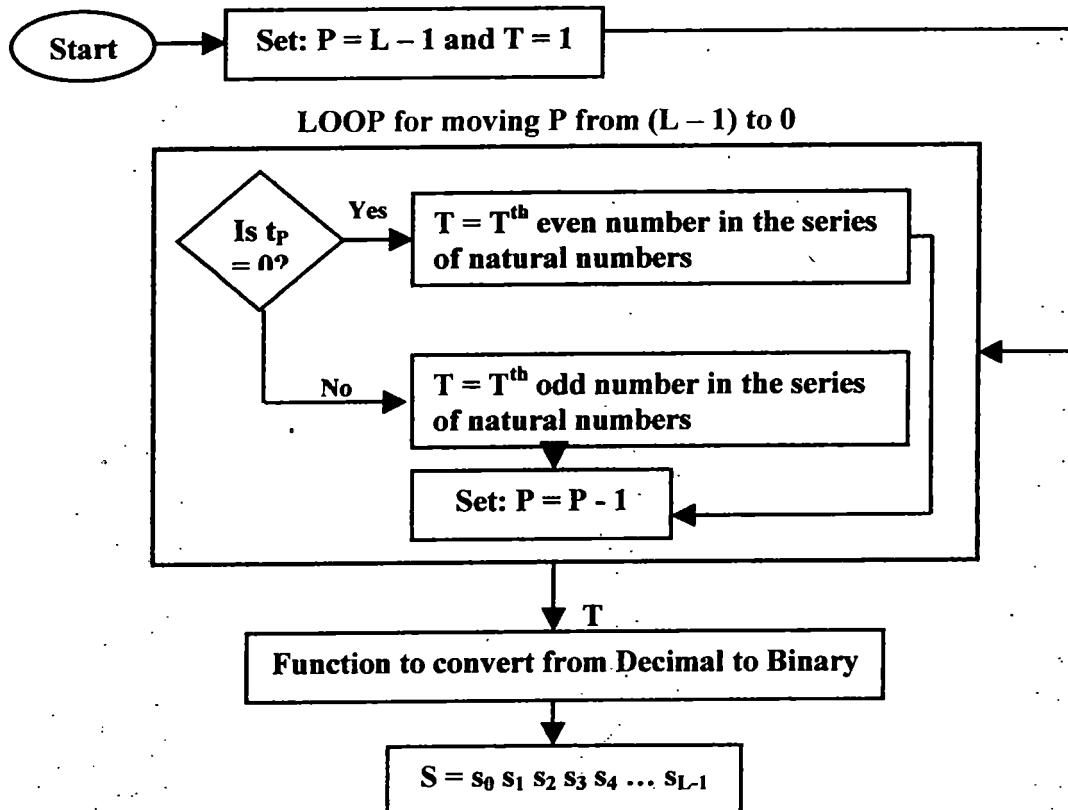
- Step 4:** Set  $P = P - 1$ .
- Step 5:** Convert  $T$  into the corresponding stream of bits  $S = s_0 s_1 s_2 s_3 s_4 \dots s_{L-1}$ , which is the source block.

```

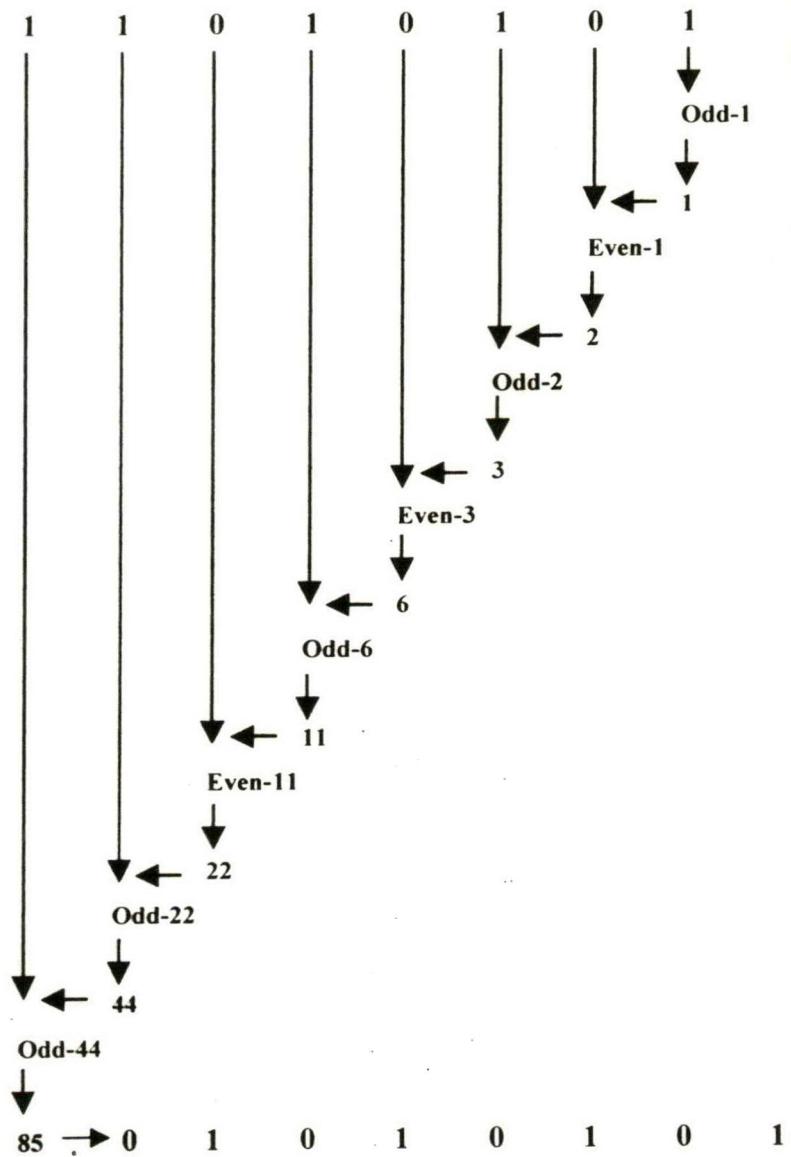
Set:  $P = L - 1$  and  $T = 1$ .
LOOP:
  If  $t_P = 0$ 
    Evaluate:  $T = T^{\text{th}}$  even number in the series of natural numbers.
  Else
    If  $t_P = 1$ 
      Evaluate:  $T = T^{\text{th}}$  odd number in the series of natural numbers.
  Set:  $P = P - 1$ .
  If  $P < 0$ 
    Exit.
ENDLOOP
Evaluate:  $S = s_0 s_1 s_2 s_3 s_4 \dots s_{L-1}$ , which is the binary equivalent of  $T$ .

```

**Figure 5.2.1.4**  
Pseudocode of RPMS Technique to decrypt a Target Block



**Figure 5.2.1.5**  
Flow Diagram of the RPMS Scheme to decrypt Target Block



**Figure 5.2.1.6**

**Formation of Source Block decrypting Target Block 11010101 in the RPMS Scheme**

Figure 5.2.1.4 shows the approach written in the form of a pseudocode and the corresponding flow diagram is shown in figure 5.2.1.5.

Figure 5.2.1.6 diagrammatically shows the generation of the source block  $S=01010101$  by decrypting the target block  $T=11010101$ . Here the process starts with the LSB, which is “1”. Since “1” stands for “odd”, the first odd number is to be considered first, which is 1. Then the previous-to-LSB bit is to be considered, which is here “0”. Now, “0” stands for “even”. Therefore the 1<sup>st</sup> even number is to be considered, which is

2. Consider the previous bit, which is “1”. As “1” stands for “odd”, 2<sup>nd</sup> odd number is to be considered, which is 3. Approaching in this way for the remaining bits till the MSB, the number “85” is obtained, for which the 8-bit block is “01010101”.

### **5.3 Implementation**

In this section, we consider a separate plaintext (P) as: “Local Area Network”. The technique being an asymmetric one, the task of encryption and that of decryption are being discussed separately. Section 5.3.1 shows how the plaintext P is to be encrypted using this technique and section 5.3.2 describes the process of decryption.

#### **5.3.1 The Process of Encryption**

Consider the message string “Local Area Network”. To construct the stream of bits table 5.3.1.1 is used, in which all the characters and their corresponding ASCII representations are enlisted.

**Table 5.3.1.1**  
**Character-to-Byte Conversion for the Text “Local Area Network”**

Character	Byte
L	01001100
o	01101111
c	01100011
a	01100001
l	01101100
<Blank>	00100000
A	01000001
r	01110010
e	01100101
a	01100001
<Blank>	00100000
N	01001110
e	01100101
t	01110100
w	01110111
o	01101111
r	01110010
K	01101011

Putting together these bytes in the original sequence, we get the source stream of bits as the following:

S=01001100/01101111/01100011/01100001/01101100/00100000/01000001/01110010/01100101/01100001/00100000/01001110/01100101/01110100/01110111/01101111/01110010/01101011.

Now, S is decomposed into a set of 5 blocks, each of the first four being of size 32 bits and the last one being of 16 bits. During this process of decomposition, S is scanned along the MSB-to-LSB direction and extract required number of bits for different block.

like the first 32 bits in this direction being for the block  $S_1$ , the next 32 bits being for the block  $S_2$ , and so on. In this way the blocks are generated as follows:

$$S_1=01001100011011110110001101100001, S_2=01101100001000000100000101110010,$$

$$S_3=01100101011000010010000001001110, S_4=01100101011101000111011101101111,$$

$$S_5=0111001001101011.$$

This way of decomposition is to be intimated as the key by the sender of the message to the receiver of the same through a secret channel. More about this has been discussed in section 5.5.

For the block  $S_1$ , corresponding to which the decimal value is  $(1282368353)_{10}$ , the process of encryption is shown below:

$$\begin{aligned} 1282368353 &\rightarrow 641184177^1 \rightarrow 320592089^1 \rightarrow 160296045^1 \rightarrow 80148023^1 \rightarrow 40074012^1 \\ &\rightarrow 20037006^0 \rightarrow 100018503^0 \rightarrow 5009252^1 \rightarrow 2504626^0 \rightarrow 1252313^0 \rightarrow 626157^1 \rightarrow \\ &313079^1 \rightarrow 156540^1 \rightarrow 78720^0 \rightarrow 39135^0 \rightarrow 19568^1 \rightarrow 9784^0 \rightarrow 4892^0 \rightarrow 2446^0 \rightarrow 1223^0 \\ &\rightarrow 612^1 \rightarrow 306^0 \rightarrow 153^0 \rightarrow 77^1 \rightarrow 39^1 \rightarrow 20^1 \rightarrow 10^0 \rightarrow 5^0 \rightarrow 3^1 \rightarrow 2^1 \rightarrow 1^0 \rightarrow 1^1. \end{aligned}$$

From this we generate the target block  $T_1$  corresponding to  $S_1$  as:  
 $T_1=11111001001110010000100111001101.$

Applying the similar process, we generate target blocks  $T_1, T_2, T_3, T_4$  and  $T_5$  as follows corresponding to source blocks  $S_1, S_2, S_3, S_4$  and  $S_5$  respectively.

$$T_2=011100010111101111101111001001$$

$$T_3=0100110111110110111100101011001$$

$$T_4=10001001000100011101000101011001$$

$$T_5=1110100110110001.$$

Now, combining target blocks in the same sequence, we get the target stream of bits  $T$  as the following:

$$T=11111001/00111001/00001001/11001101/01110001/01111101/111110$$

$$11/11001001/01001101/11111011/01111001/01011001/10001001/000100$$

$$01/11010001/01011001/11101001/10110001.$$

This stream ( $T$ ) of bits, in the form of a stream of characters, is transmitted as the encrypted message ( $C$ ), which looks like the following:

•9°=q}√M√yYé►Y@

### 5.3.2 The Process of Decryption

At the destination point, this encrypted message or the ciphertext C reaches and for the purpose of decryption the receiver has only the secret key. Now, by that secret key, the suggested format of which is discussed in section 5.5, the receiver gets the information on different block lengths. Using that secret key, all the blocks  $T_1, T_2, T_3, T_4$  and  $T_5$  are formed as follows:

$$T_1=11111001001110010000100111001101$$

$$T_2=011100010111101111101111001001$$

$$T_3=0100110111110110111100101011001$$

$$T_4=10001001000100011101000101011001$$

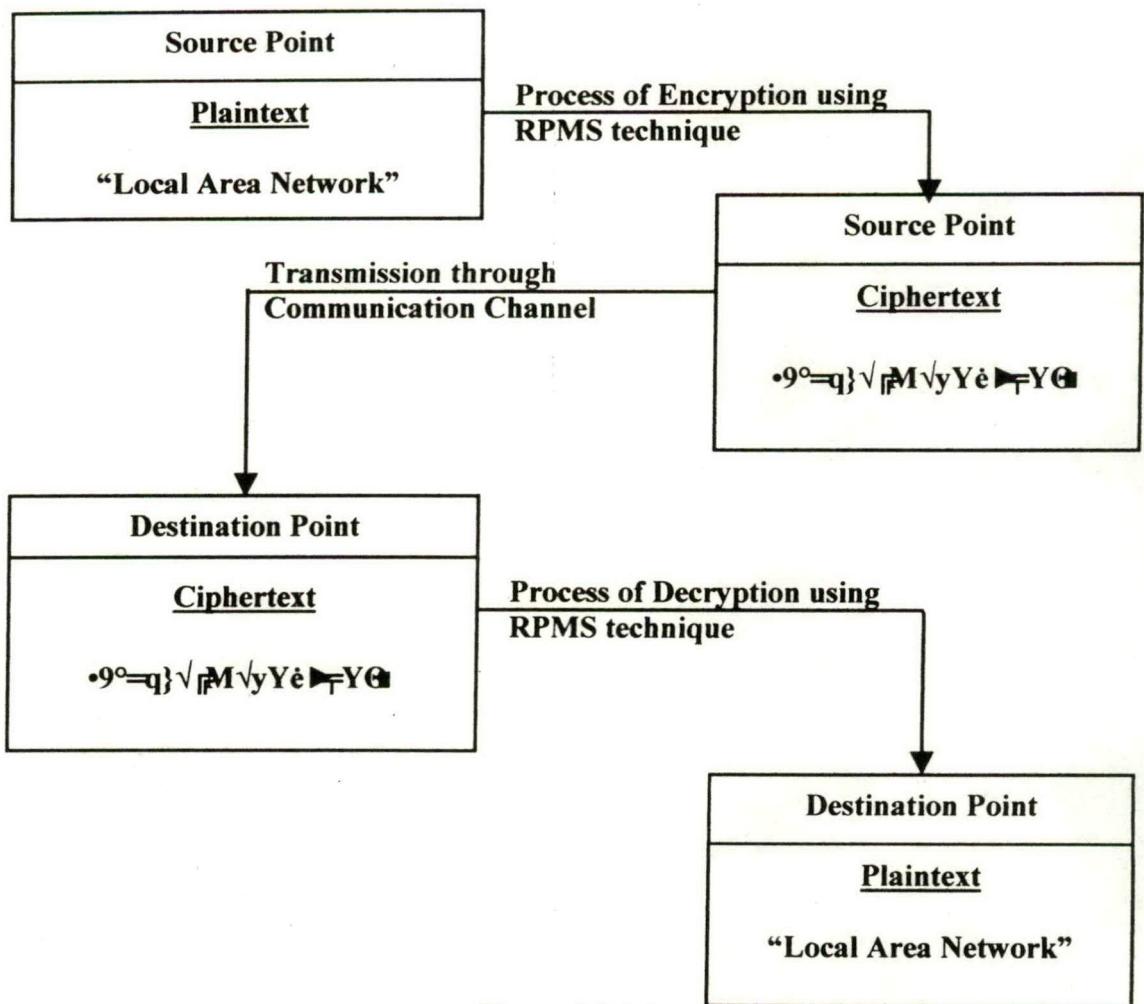
$$T_5=1110100110110001.$$

Now, applying the process of decryption shown in figures 5.2.1.4 and 5.2.1.5, the corresponding source blocks  $S_i$  are generated for all  $T_i, 1 \leq i \leq 5$ .

As for example, for the target block  $T_1$ , we may proceed in the following way:

“First odd number is 1, 1<sup>st</sup> even is 2, 2<sup>nd</sup> odd number is 3, 3<sup>rd</sup> odd number is 5. 5<sup>th</sup> even number is 10, 10<sup>th</sup> even number is 20, 20<sup>th</sup> odd number is 39, 39<sup>th</sup> odd number is 77, 77<sup>th</sup> odd number is 153, 153<sup>rd</sup> even number is 306, 306<sup>th</sup> even number is 612, 612<sup>th</sup> odd number is 1223, 1223<sup>rd</sup> even number is 2446, 2446<sup>th</sup> even number is 4892, 4892<sup>nd</sup> even number is 9784, 9784<sup>th</sup> even number is 19568, 19568<sup>th</sup> odd number is 39135, 39135<sup>th</sup> even number is 78720, 78720<sup>th</sup> even number is 156540, 156540<sup>th</sup> odd number is 313079, 313079<sup>th</sup> odd number is 626157, 626157<sup>th</sup> odd number is 1252313, 1252313<sup>th</sup> even number is 2504626, 2504626<sup>th</sup> even number is 5009252, 5009252<sup>nd</sup> odd number is 100018503, 100018503<sup>rd</sup> even number is 20037006, 20037006<sup>th</sup> even number is 40074012, 40074012<sup>th</sup> odd number is 80148023, 80148023<sup>rd</sup> odd number is 160296045, 160296045<sup>th</sup> odd number is 320592089, 320592089<sup>th</sup> odd number is 641184177, and finally 641184177<sup>th</sup> odd number is 1282368353, for which the corresponding 32-bit stream is  $S_1=010011000110111011000110110001$ .”

In this way all the source blocks of bits are regenerated and combining those blocks in the same sequence, the source stream of bits are obtained to get the source message or the plaintext. The schematic diagram of this entire technique is shown in the figure 5.3.2.1.



**Figure 5.3.1.1**  
**Schematic Diagram of Encryption/Decryption Techniques for**  
**Plaintext “Local Area Network”**

#### 5.4 Results

In this section results have been taken on the basis of the following factors:

- Computation of the encryption time and the decryption time, and hence establishing graphical relationships among the source size, the encryption time and/or the decryption time; also calculation of the Chi square value between the source and the encrypted files
- Performing the frequency distribution test
- Comparison with the RSA system

Experimentations on the basis of these four factors are respectively shown in section 5.4.1, section 5.4.2, and section 5.4.3.

In the next section, section 5.5, all these results have been analyzed from different perspectives.

#### **5.4.1 Computing Encryption/Decryption Time**

The same set of real-life files we have considered for the experimentation purpose. To ease the implementation, a unique block length has been considered. In this section all the results have been shown for block size of 16 bits [54, 55, 56].

Section 5.4.1.1 analyzes the results taken for the .EXE files, section 5.4.1.2 analyzes the results taken for the .COM files, section 5.4.1.3 analyzes the results taken for .DLL files, section 5.4.1.4 analyzes the results taken for the .SYS files, and section 5.4.1.5 analyzes the results taken for the .CPP files. In each of these sections the encryption time and the decryption time have been presented and graphically it has been shown how these execution times vary with the size of source file and that of the encrypted file. Each section also shows that in no case there is any storage overhead. Section 5.4.1.6 discusses about the results of the chi square tests.

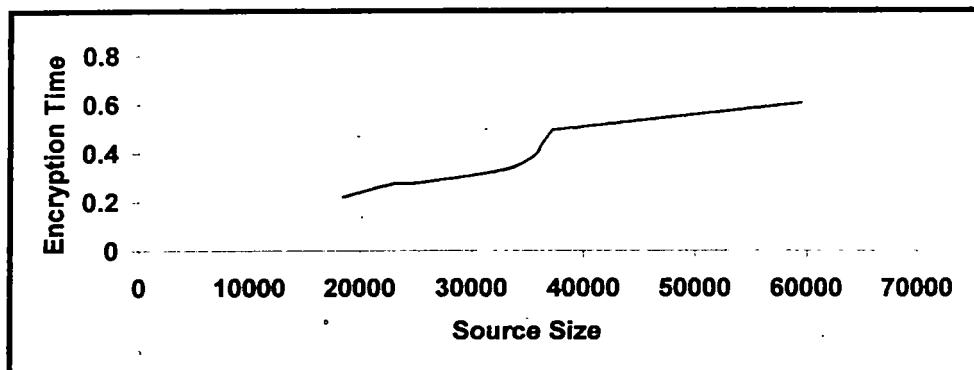
##### **5.4.1.1 Result for *EXE* Files**

Table 5.4.1.1.1 shows the result for the .EXE files. Eight files have been considered. The size of each file is in the range of 23044 bytes to 59398 bytes. The encryption time is in the range of 0.2198 seconds to 0.6044 seconds. The decryption time lies in the range of 0.1648 seconds to 0.3846 seconds. The Chi Square value is in the range of 38480 to 444374 with the degree of freedom ranging from 248 to 255.

**Table 5.4.1.1.1**  
**Results for EXE Files in Tabular Form for RPMS Technique**

Source File	Encrypted File	Source Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom
TLIB.EXE	A1.EXE	37220	0.3297	0.2198	364571	255
MAKER.EXE	A2.EXE	59398	0.6044	0.3846	444374	255
UNZIP.EXE	A3.EXE	23044	0.2747	0.1648	38480	255
RPPO.EXE	A4.EXE	35425	0.3846	0.2747	128642	255
PRIME.EXE	A5.EXE	37152	0.4945	0.3297	143696	255
TRIANGLE.EXE	A6.EXE	36242	0.4396	0.2198	136176	255
PING.EXE	A7.EXE	24576	0.2747	0.1648	127733	248
NETSTAT.EXE	A8.EXE	32768	0.3297	0.2198	387668	255
CLIPBRD.EXE	A9.EXE	18432	0.2198	0.1648	150396	255

The graphical relationship between the source file size and the encryption time on the basis of the results taken for the .EXE files is shown in figure 5.4.1.1.1. The figure establishes the fact that there exists a tendency that the encryption time increases linearly with the size of the source file.



**Figure 5.4.1.1.1**  
**Relationship between Source Size and Encryption Time for EXE Files in RPMS Technique**

#### 5.4.1.2 Result for COM Files

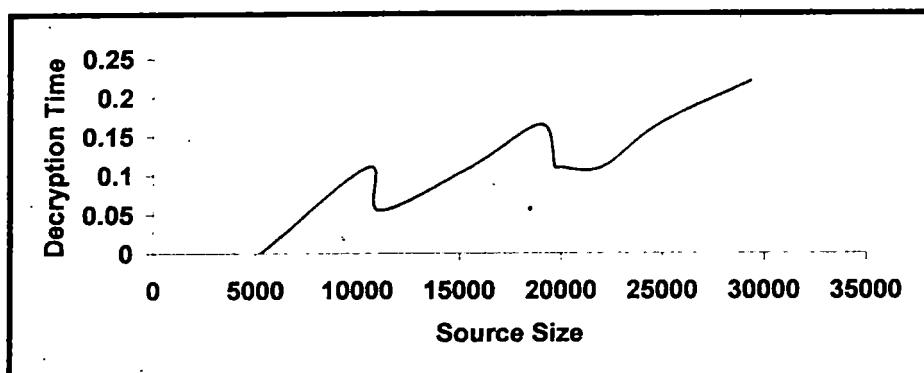
Table 5.4.1.2.1 presents the results for the .COM files. Ten files have been considered. The size of each file is in the range of 5239 bytes to 29271 bytes. The encryption time is in the range of 0.1099 seconds to 0.3297 seconds. The decryption time

lies in the range of 0.0000 seconds to 0.2198 seconds. The Chi Square value is in the range of 13822 to 121318 with the degree of freedom ranging from 230 to 255.

**Table 5.4.1.2.1**  
**Results for COM Files in Tabular Form for RPMS Technique**

Source File	Encrypted File	Source Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom
EMSTEST.COM	A1.COM	19664	0.2747	0.1099	68516	255
THELP.COM	A2.COM	11072	0.1648	0.0549	70590	250
WIN.COM	A3.COM	24791	0.2747	0.1648	79927	252
KEYB.COM	A4.COM	19927	0.2198	0.1099	121318	255
CHOICE.COM	A5.COM	5239	0.1099	0.0000	13822	232
DISKCOPY.COM	A6.COM	21975	0.2747	0.1099	91538	254
DOSKEY.COM	A7.COM	15495	0.1648	0.1099	51497	253
MODE.COM	A8.COM	29271	0.3297	0.2198	113529	255
MORE.COM	A9.COM	10471	0.1099	0.1099	15120	230
SYS.COM	A10.COM	18967	0.2198	0.1648	94004	254

Figure 5.4.1.2.1 graphically shows the relationship between the source file size and the decryption time for .COM files. In this case there exists no tendency of linear relationship between the decryption time and the size of the source file.



**Figure 5.4.1.2.1**  
**Relationship between Source Size and Encryption Time for COM Files in RPMS Technique**

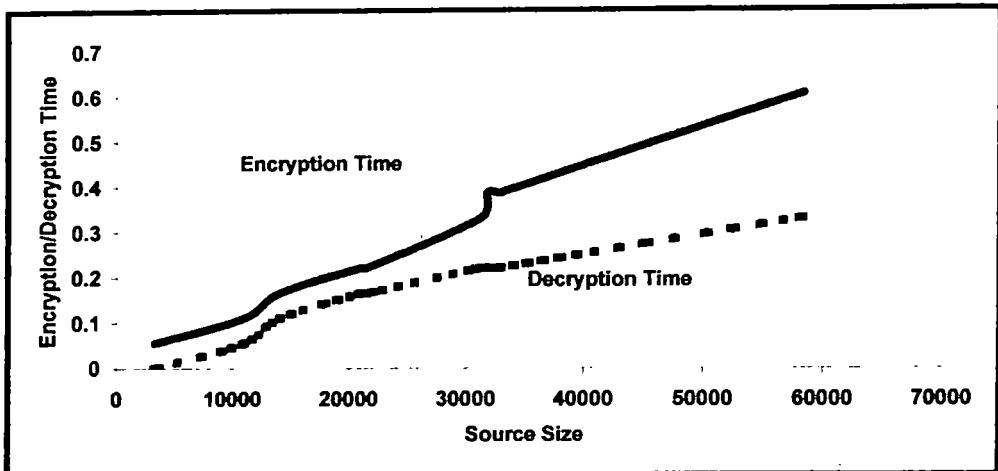
#### 5.4.1.3 Result for **DLL** Files

Table 5.4.1.3.1 represents the results taken for the .DLL files. Ten files have been considered. The size of each file is in the range of 3216 bytes to 58368 bytes. The encryption time is in the range of 0.0549 seconds to 0.6044 seconds. The decryption time lies in the range of 0.0000 seconds to 0.3297 seconds. The Chi Square value is in the range of 10696 to 414717 with the degree of freedom ranging from 217 to 255.

**Table 5.4.1.3.1**  
**Result for DLL Files for RPMS Technique**

Source File	Encrypted File	Source Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom
<i>SNMPAPI.DLL</i>	<i>A1.DLL</i>	32768	0.3846	0.2198	118235	253
<i>KPSHARP.DLL</i>	<i>A2.DLL</i>	31744	0.3846	0.2198	265630	254
<i>WINSOCK.DLL</i>	<i>A3.DLL</i>	21504	0.2198	0.1648	414717	252
<i>SPWHPT.DLL</i>	<i>A4.DLL</i>	32792	0.3846	0.2198	160065	255
<i>HIDCI.DLL</i>	<i>A5.DLL</i>	3216	0.0549	0.0000	10696	217
<i>PFPICK.DLL</i>	<i>A6.DLL</i>	58368	0.6044	0.3297	197903	255
<i>NDDEAPI.DLL</i>	<i>A7.DLL</i>	14032	0.1648	0.1099	128372	249
<i>NDDENB.DLL</i>	<i>A8.DLL</i>	10976	0.1099	0.0549	172239	251
<i>ICCCODES.DLL</i>	<i>A9.DLL</i>	20992	0.2198	0.1648	141924	252
<i>KPSCALE.DLL</i>	<i>A10.DLL</i>	31232	0.3297	0.2198	287292	255

Figure 5.4.1.3.1 represents the relationship of the source size with the encryption/decryption time for .DLL files. Here the continuous curve stands for the encryption time, whereas the dotted curve stands for the decryption time. It is clear from the figure that both the encryption time and the decryption time have the tendency of varying linearly with the source size.



**Figure 5.4.1.3.1**  
**Relationship between Source Size and Encryption/Decryption Time for DLL Files in RPMS Technique**

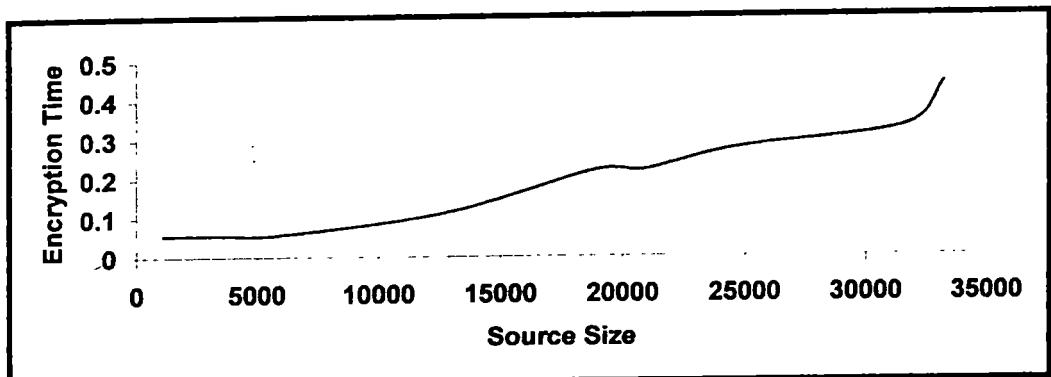
#### 5.4.1.4 Result for SYS Files

Table 5.4.1.4.1 shows the results taken for the .SYS files. Ten files have been considered. The size of each file is in the range of 1105 bytes to 33191 bytes. The encryption time is in the range of 0.0549 seconds to 0.4397 seconds. The decryption time lies in the range of 0.0000 seconds to 0.3297 seconds. The Chi Square value is in the range of 2278 to 241968 with the degree of freedom ranging from 165 to 255.

**Table 5.4.1.4.1**  
**Result for SYS Files for RPMS Technique**

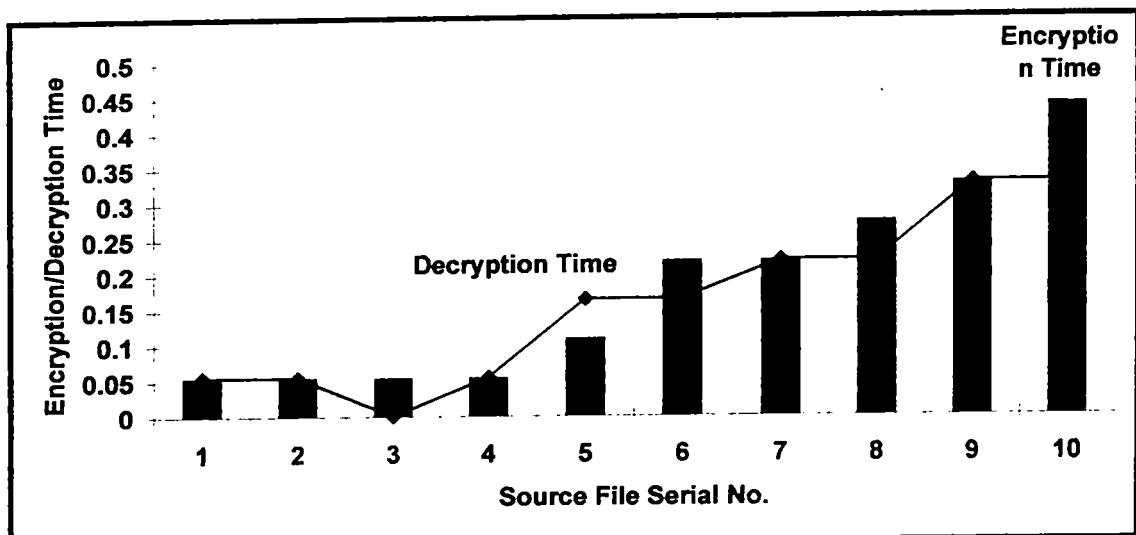
Source File	Encrypted File	Source Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom
HIMEM.SYS	A1.SYS	33191	0.4397	0.3297	106943	255
RAMDRIVE.SYS	A2.SYS	12663	0.1099	0.1648	22352	241
USBD.SYS	A3.SYS	18912	0.2198	0.1648	158928	255
CMD640X.SYS	A4.SYS	24626	0.2747	0.2198	129672	255
CMD640X2.SYS	A5.SYS	20901	0.2198	0.2198	114070	255
REDBOOK.SYS	A6.SYS	5664	0.0549	0.0549	23469	230
IFSHLP.SYS	A7.SYS	3708	0.0549	0.0549	13116	237
ASPI2HLP.SYS	A8.SYS	1105	0.0549	0.0000	2278	165
DBLBUFF.SYS	A9.SYS	2614	0.0549	0.0549	4606	215
CCPORT.SYS	A10.SYS	31680	0.3297	0.3297	241968	255

Figure 5.4.1.4.1 shows the relationship between the source file size and the encryption time for .SYS files. The figure shows that the encryption time varies linearly enough with the source size.



**Figure 5.4.1.4.1**  
**Relationship between Source Size and Encryption Time for SYS Files**

Figure 5.4.1.4.2 establishes the graphical relationship between the encryption time and the decryption time for .SYS files. In the figure black horizontal pillars stand for different encryption times. Along the left-to-right direction the pillars have been arranged As per the increasing order of their heights. On each pillar, the corresponding decryption time has been marked as a point and all these points have been joined together to obtain a curve. Except on one occasion (on 3<sup>rd</sup> pillar), the curve is moving upward. So, it can be interpreted that there exists a tendency that the decryption time varies linearly with the encryption time, although some exceptions may also exist.



**Figure 5.4.1.4.2**  
**Relationship between Encryption Time and Decryption Time for .SYS Files for RPMS Technique**

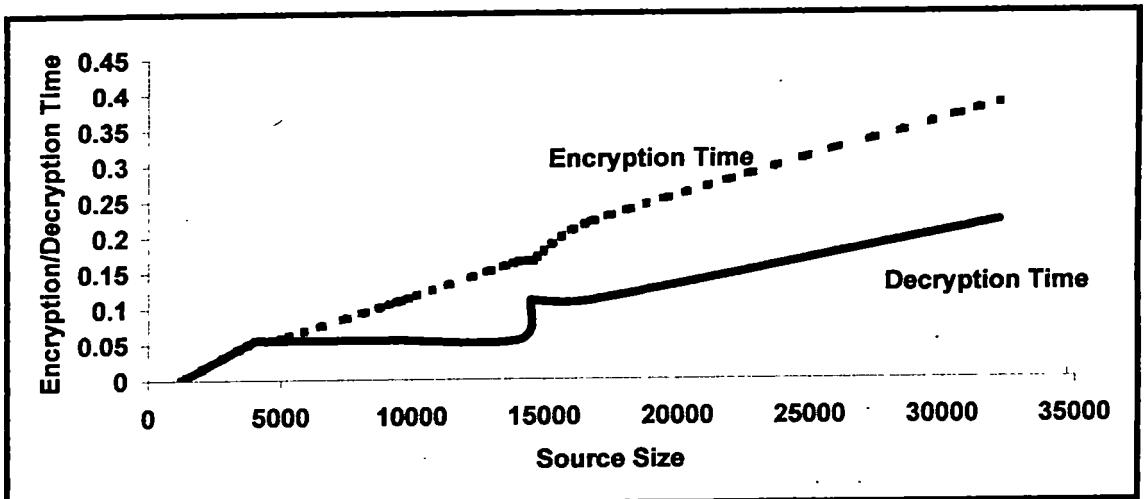
#### 5.4.1.5 Result for CPP Files

Table 5.4.1.5.1 presents the result for the .CPP files. Ten files have been considered. The size of each file is in the range of 1257 bytes to 32150 bytes. The encryption time is in the range of 0.0000 seconds to 0.3846 seconds. The decryption time lies in the range of 0.0000 seconds to 0.2198 seconds. The Chi Square value is in the range of 1794 to 438133 with the degree of freedom ranging from 69 to 90.

**Table 5.4.1.5.1**  
**Result for .CPP Files**

Source File	Encrypted File	Source Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom
BRICKS.CPP	A1.CPP	16723	0.2198	0.1099	113381	88
PROJECT.CPP	A2.CPP	32150	0.3846	0.2198	438133	90
ARITH.CPP	A3.CPP	9558	0.1099	0.0549	143723	77
START.CPP	A4.CPP	14557	0.1648	0.1099	297753	88
CHARTCOM.CPP	A5.CPP	14080	0.1648	0.0549	48929	84
BITIO.CPP	A6.CPP	4071	0.0549	0.0549	9101	70
MAINC.CPP	A7.CPP	4663	0.0549	0.0549	22485	83
TTEST.CPP	A8.CPP	1257	0.0000	0.0000	1794	69
DO.CPP	A9.CPP	14481	0.1648	0.1099	294607	88
CAL.CPP	A10.CPP	9540	0.1099	0.0549	143672	77

Graphically the relationship between the source size and the encryption/decryption time for .CPP files is shown in figure 5.4.1.5.1. Here the dotted curve stands for the encryption time and the other stands for the decryption time. It is observed that both the encryption time and the decryption time have the tendency of varying linearly with the size of the source file.

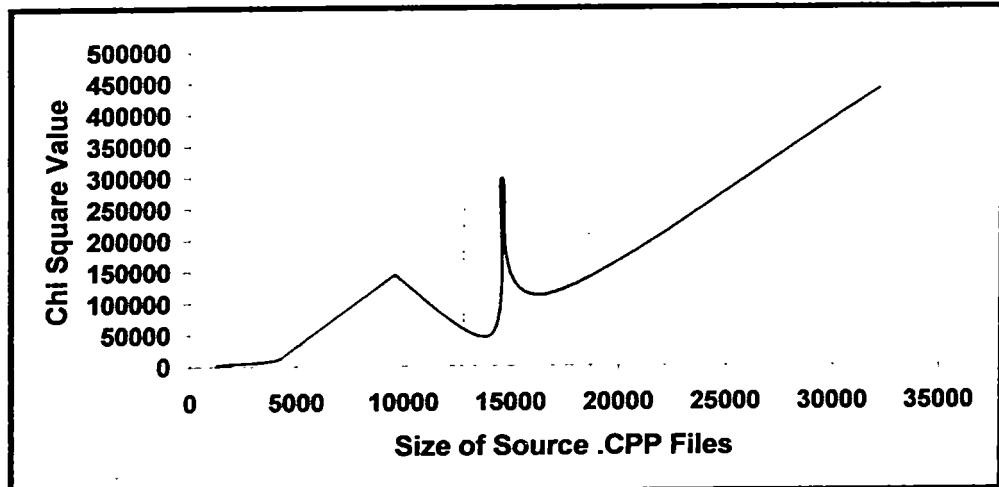


**Figure 5.4.1.5.1**  
**Graphical Relationship between Source Size and Encryption/Decryption Time for .CPP Files in RPMS Technique**

#### 5.4.1.6 Discussion on Chi Square Tests

As it is observed from the results of the chi square tests taken for .EXE, .COM, .DLL, and .SYS files, chi square values are much more less in comparison to values for .CPP files [44].

Figure 5.4.1.6.1 shows how chi square values change with source file size only for the category of .CPP files. From this figure any fixed conclusion hardly can be drawn. As the Chi Square value actually depends on the content of the source and the corresponding encrypted file; for files of almost same size Chi Square values may differ to a large extend. But it is observed that there exists a tendency that the Chi Square value increases with the file size, although the rate at which it increases is certainly not fixed and there exists a number of exceptions too.

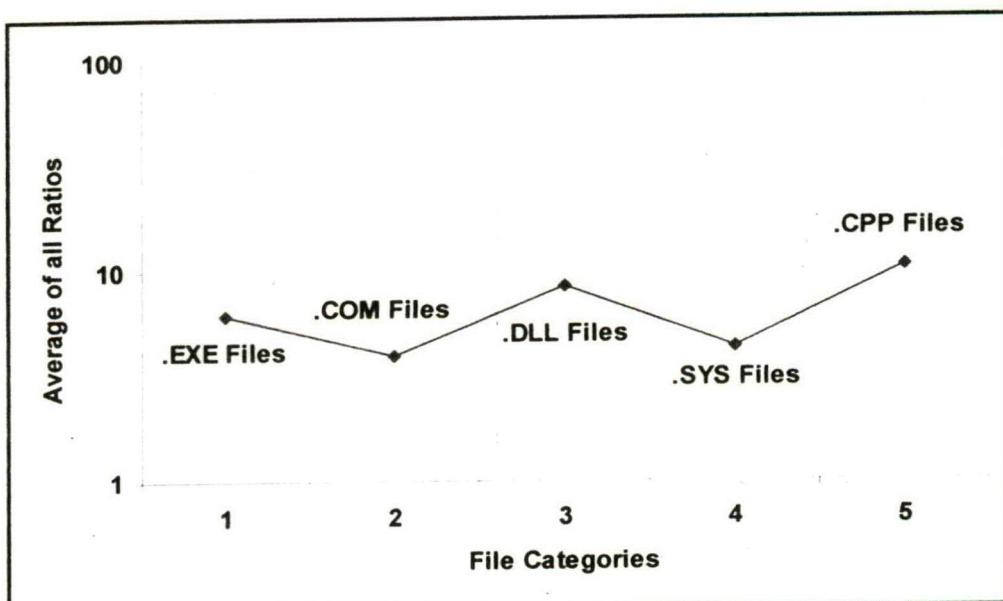


**Figure 5.4.1.6.1**  
**Variation of Chi Square Values with File Sizes (For .CPP Files)**

Figure 5.4.1.6.2 attempts to represent a graphical outlook of a comparative analysis of the averages of all ratios of the chi square value and the source file size for all the five categories considered here. These average values are enlisted in table 5.4.1.6.1. From the table and the figure, it is observed that the result is most satisfactory in case of the .CPP files, where the average value is found to be 10.3263, in comparison with 3.9578 for .COM files, 4.3256 for .SYS files, 6.1545 for .EXE files, and 8.3660 for .DLL files.

**Table 5.4.1.6.1**  
**Ratios of Average Chi Square Value and**  
**Average Source Size for Different Categories of Files**

Category of Files	Average of all Ratios of Chi Square Value and Source File Size
.EXE	6.1545
.COM	3.9578
.DLL	8.3660
.SYS	4.3256
.CPP	10.3263

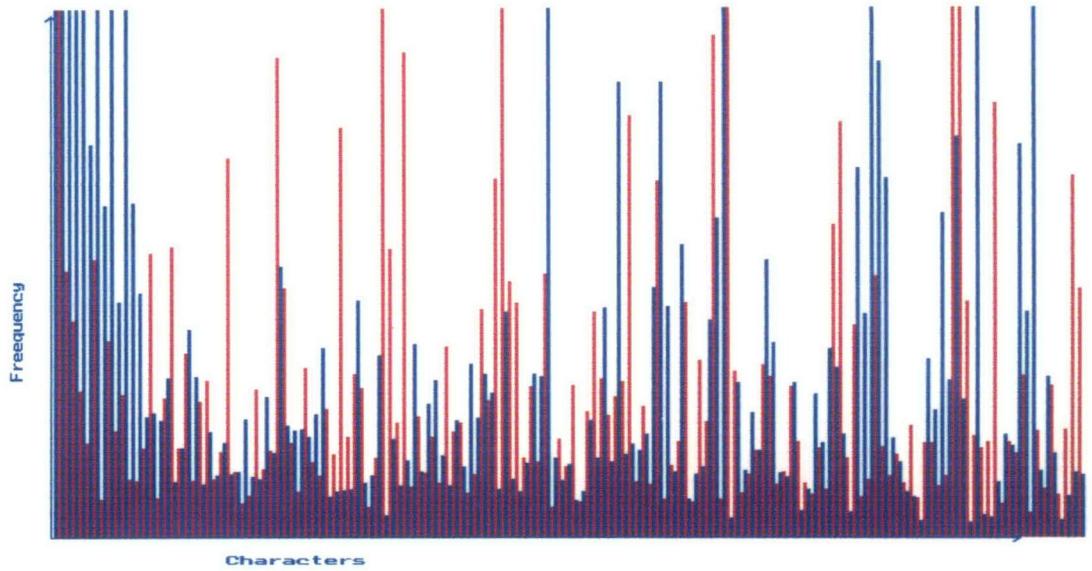


**Table 5.4.1.6.2**  
**Comparative Graphical Representation of**  
**Ratios of Average Chi Square Value and**  
**Average Source Size for Different Categories of Files**

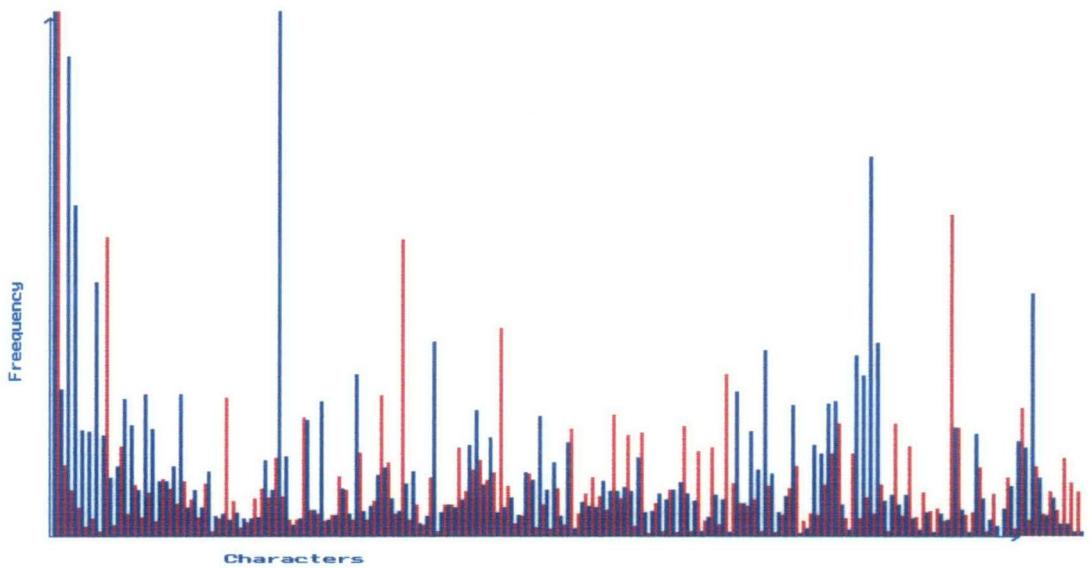
#### 5.4.2 Result on Frequency Distribution Tests

Representing the result of the frequency distribution test for all the files considered in section 5.4.1 being an impractical task, here in this section, for the representation purpose only 5 files, one each from .EXE, .COM, .DLL, .SYS, and .CPP have been considered. In each case, frequency distribution is pictorially represented for the source file and the encrypted file. It is seen for all cases that the characters in the encrypted files are well distributed, which indicates that the technique proposed here is quite compatible with existing techniques. The red bars represent frequencies of characters in the encrypted file and those in blue color represent frequencies of characters in the source file. The frequencies of characters in encrypted files are evenly distributed. Therefore the source and the corresponding encrypted file are heterogeneous in nature. Hence it can be interpreted that through the proposed technique, a good quality of encryption is obtained.

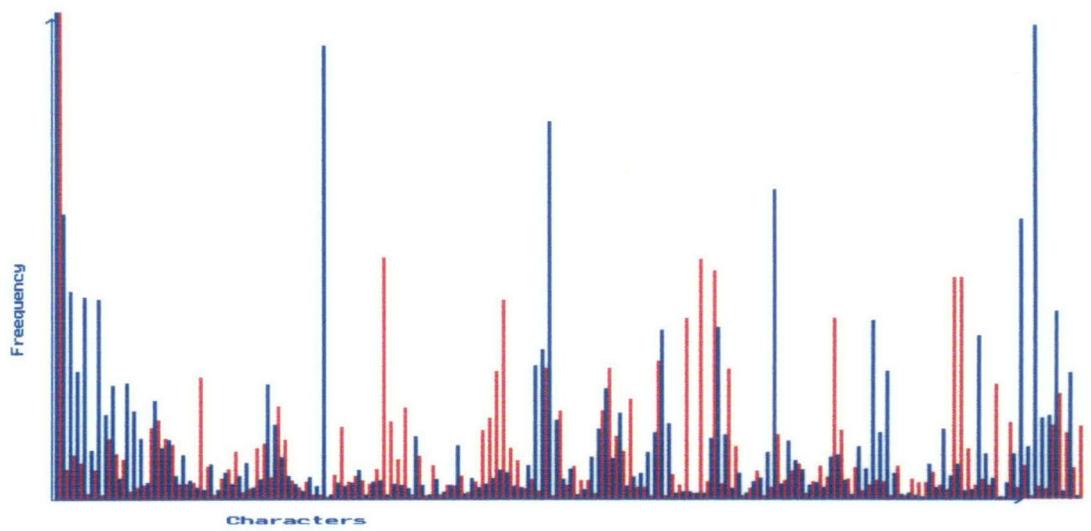
Figure 5.4.2.1 to figure 5.4.2.5 show these results respectively.



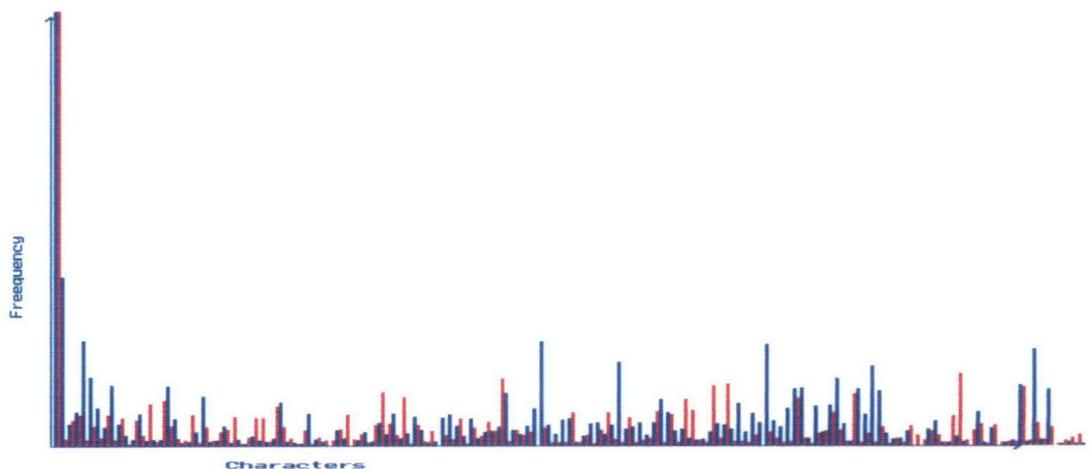
**Figure 5.4.2.1**  
**Frequency Distribution for PRIME.EXE and its Encrypted File for RPMS Technique**



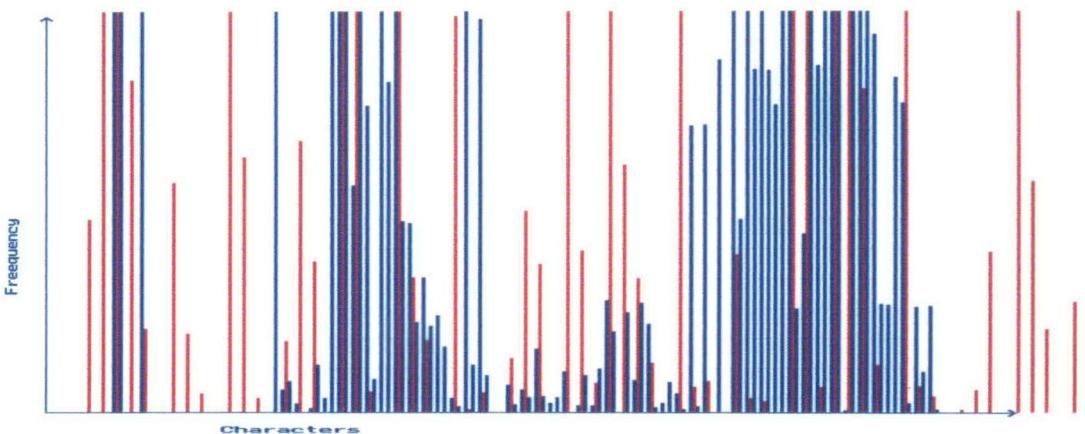
**Figure 5.4.2.2**  
**A Segment of Frequency Distribution for DOSKEY.COM and its Encrypted File for RPMS Technique**



**Figure 5.4.2.3**  
**A Segment of Frequency Distribution for NDDENB.DLL and its Encrypted File for RPMS Technique**



**Figure 5.4.2.4**  
**A Segment of Frequency Distribution for REDBOOK.SYS and its Encrypted File for RPMS Technique**



**Figure 5.4.2.4**  
**A Segment of Frequency Distribution for PROJECT.CPP and its Encrypted File for RPMS Technique**

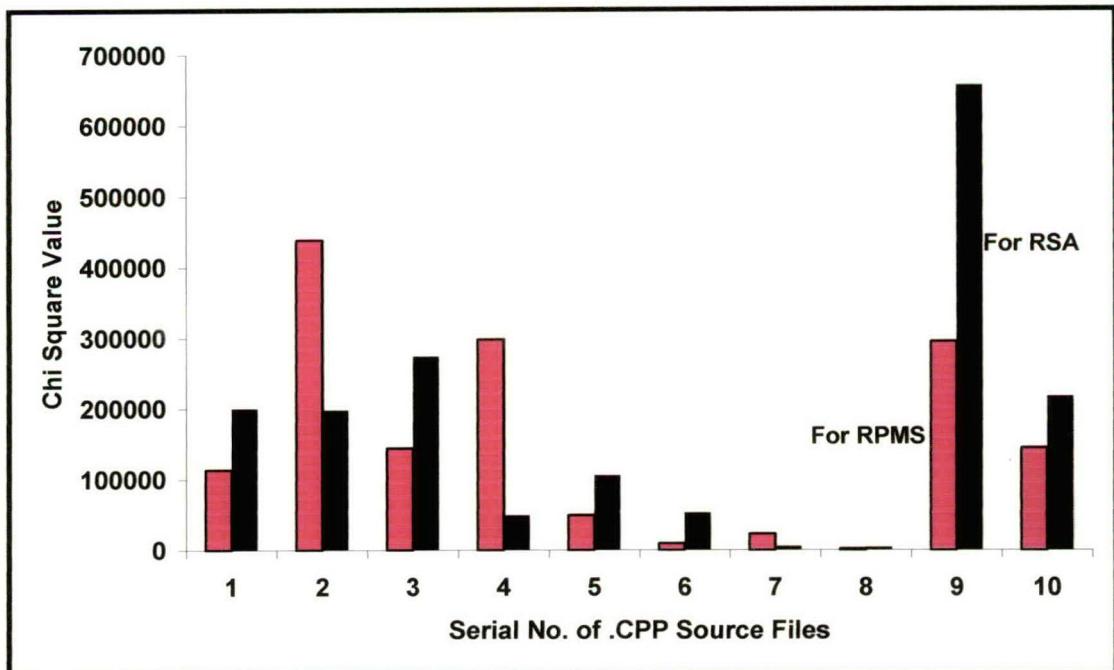
### 5.4.3 Comparison with RSA Technique

For the purpose of comparing the performance of the RPMS technique with the RSA system for a given set of files, the same set of 10 .CPP files have been considered. Table 5.4.3.1 represents this report. When the proposed RPMS technique is used for the encryption purpose, the Chi Square value is observed to be in the range of 1794 to 438133. If the existing RSA technique is used for the purpose of encryption, the Chi Square value is observed to be in the range of 3652 to 655734 [8, 40].

**Table 5.4.3.1**  
**Comparative Results between RPMS Technique and RSA System for .CPP Files**

Source File	Encrypted File Using RPMS Technique	Encrypted File Using RSA Technique	Chi Square Value For RPMS Technique	Chi Square Value For RSA Technique	Degree of Freedom
BRICKS.CPP	A1.CPP	CPP1.CPP	113381	200221	88
PROJECT.CPP	A2.CPP	CPP2.CPP	438133	197728	90
ARITH.CPP	A3.CPP	CPP3.CPP	143723	273982	77
START.CPP	A4.CPP	CPP4.CPP	297753	49242	88
CHARTCOM.CPP	A5.CPP	CPP5.CPP	48929	105384	84
BITIO.CPP	A6.CPP	CPP6.CPP	9101	52529	70
MAINC.CPP	A7.CPP	CPP7.CPP	22485	4964	83
TTEST.CPP	A8.CPP	CPP8.CPP	1794	3652	69
DO.CPP	A9.CPP	CPP9.CPP	294607	655734	88
CAL.CPP	A10.CPP	CPP10.CPP	143672	216498	77

The information of table 5.4.3.1 has graphically been shown in figure 5.4.3.1. Here black pillars stand for Chi Square results for the RSA technique and red pillars stand for the same for the RPMS technique. As it is seen from the figure that red pillars are well compatible with black pillars and there exist three cases where red pillars are taller than black ones.



**Figure 5.4.3.1**  
**Graphical Comparison of Chi Square Values between RPMS and RSA Techniques**

## 5.5 Analysis and Conclusion including Comparison with RPSP, TE, RPPO

Out of the four encryption techniques proposed so far in this thesis, it is observed that this RPMS technique produces the maximum Chi Square value on the average. Table 5.5.1 presents this information. The average Chi Square value observed for the RPMS technique is 140196.94, against 10701.70 for the RPSP technique, 64188.04 for the TE technique, and 85350.94 for the RPPO technique. In case of the RPMS technique, the average encryption time is observed to the lowest among the four techniques, which is 0.23659592 seconds, and the same is true for the average decryption time as well, which is 0.15137143 seconds [45, 48, 50, 52].

In chapter 10, graphical comparisons have been drawn.

**Table 5.5.1**  
**Average Encryption/Decryption Time and Chi Square Value obtained in  
 RPSP, TE, RPPO, RPMS**

<b>Proposed Technique</b>	<b>Average Encryption Time</b>	<b>Average Decryption Time</b>	<b>Average Chi Square Value</b>	<b>Average Degree of Freedom</b>
<b>RPSP</b>	<b>8.75713800</b>	<b>8.73955200</b>	<b>10701.70</b>	<b>214</b>
<b>TE</b>	<b>0.86703290</b>	<b>0.94175818</b>	<b>64188.04</b>	
<b>RPPO</b>	<b>0.73186806</b>	<b>7.03076904</b>	<b>85350.94</b>	
<b>RPMS</b>	<b>0.23659592</b>	<b>0.15137143</b>	<b>140196.94</b>	

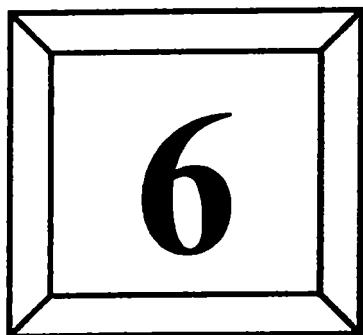
Since for the purpose of practical implementation blocks are constructed of unique size of only 16 bits, it is expected to achieve much better performance if bigger blocks are constructed, not necessarily of a unique size.

From the schematic point of view, the proposed RPMS technique is not different from at least some of the other proposed techniques like the RSBP technique and the RSBM technique.

Here the strength of the encryption policy becomes prominent if blocks are constructed of varying sizes. But, since here neither any cycle is formed, nor there exists any option for a source block to choose the corresponding encrypted block, the degree of variation in sizes of blocks should be made much high, so that a reasonably long key space becomes inevitable for a successful decryption.

Figure 8.2.4.1 in chapter 8 shows one proposed format of a 110-bit key constructed by strictly following a set of protocols for blocks formation. By allowing more flexibility in the approach of blocks formation, a much longer key space can be generated.

Evaluating from all perspectives, it can be concluded that the RPMS encryption policy is expected to offer a very satisfactory level of information security with providing construction of varying sizes of blocks. Also it may participate successfully in the cascaded approach of implementation discussed in chapter 9.



**Encryption Through  
Recursive Substitutions of Bits  
Through Prime-Nonprime (RSBP)  
Detection of Sub-stream**

<u>Contents</u>	<u>Pages</u>
-----------------	--------------

<b>6.1      Introduction</b>	<b>197</b>
<b>6.2      The Scheme</b>	<b>198</b>
<b>6.3      Implementation</b>	<b>206</b>
<b>6.4      Results</b>	<b>217</b>
<b>6.5      Analysis and Conclusion including Comparison with RPSP, TE, RPPO, RPMS</b>	<b>231</b>

## **6.1 Introduction**

The Recursive Substitutions of Bits through Prime-nonprime (RSBP) detection of sub-stream is a completely different technique in comparison to the other proposed techniques because this is the only proposed technique, in which in anticipation of getting the maximum security a little storage overhead has been accepted.

Like the RPSP technique, described in chapter 2, in this RSBP technique also the basic operation to be performed is prime-oriented. So, in this regard, this technique is quite different from the RPPO technique, described in chapter 4, and the TAE/TSE technique, described in chapter 3, because in those two techniques the basic operations performed were Boolean operations. Also, unlike the RPSP technique and the RPPO technique, here there is no formation of cycle. That means, directly by applying this technique once the target file can be generated from the source one.

Since, like all the other proposed techniques, this is also a bit-level technique, the source file is to be converted into a stream of bits. To avoid the increasing complexity of the implementation, it is suggested to decompose the stream into blocks of unit length. For a block of bits, the corresponding decimal number is to be calculated, like the RPMS technique, described in chapter 5. Depending on whether that decimal number is prime or not, a corresponding 1-bit code value is assigned for that block and, to identify the source block uniquely, a rank value is also assigned to the block.

To form the encrypted file, code values of all the source blocks are put together first, followed by all the rank values, preferably in the reverse sequence. In between these two, a few extra 0's may have to be inserted to form size of the encrypted file as a multiple of 8 bits.

During the process of decryption, the key of the technique helps in getting the unique block size and hence the total number of blocks, assuming that the size of the source file is known to the receiver. From these, it is to be calculated the possible length of the rank values corresponding to each of the two code values. Combining these, all the source blocks are generated one by one [3, 29, 30, 52, 53].

Section 6.2 discusses the scheme. The technique is implemented for a certain text in section 6.3. Section 6.4 enlists the results from different perspectives after the technique is implemented for the same groups of files that were considered for the other

proposed techniques. An analytical evaluation of the technique is presented in section 6.5.

## 6.2 The Scheme

Like all the other techniques proposed, this RSBP technique is also of bit-level implementation. Naturally, given a source file to be encrypted using this technique, the first task to be performed is to obtain the corresponding stream of bits. Section 6.2.1 discusses how this source bit of stream is encrypted and section 6.2.2 discusses the process of decryption to regenerate the original stream.

### 6.2.1 The Encryption Technique

One major difference of this technique with all the other proposed techniques is that here corresponding to one block, there is no so called “target block”. It is because of the fact that for each block there is one 1-bit code value and one rank value, and these two are not put together. For ensuring the error-free decryption, the code values of all the blocks are put together one by one as to start from the MSB position and the corresponding rank values are put together one by one from last as to start from the LSB position. Between these two, a few extra bits, preferably 0's, the number of which should not exceed 7, may have to be inserted, so that the size of the encrypted stream becomes a multiple of 8.

Following is the stepwise approach to be followed to encrypt the source stream of bits.

**Step 1:** Decompose the source stream, say, into a finite number of blocks, each preferably of the same size, say, L.

**Step 2:** Calculate the total number of primes and nonprimes in the range of 0 to  $(2^L - 1)$ . Accordingly, find minimum how many bits are required to represent each of these two numbers.

Step 3 to step 5 are to be applied for all the blocks.

**Step 3:** For the block under consideration, calculate the decimal number corresponding to that. Say, it is D.

**Step 4:** Find out if D is prime or nonprime. If D is prime, the code value for that block is 1 and if not so, it is 0.

**Step 5:** In the series of primes or nonprimes (whichever be applicable for D) in the range of 0 to  $(2^L-1)$ , find the position of D. Represent this position in terms of binary values. This is the rank of this block.

After repeating these steps (3, 4 and 5) for all the blocks, following steps are to be followed.

**Step 6:** Say, there are N number of blocks. In the target stream of bits, put all the N code values one by one starting from the MSB position. So, in the target stream, the first N bits are code values for N blocks.

**Step 7:** For putting all the rank values in the target stream, we are to start from the  $N^{th}$  bit from the MSB position and then to come back bit-by-bit. Immediately after the  $N^{th}$  bit, put the rank value of the  $N^{th}$  block, followed by the rank value of the  $(N-1)^{th}$  block, and so on. In this way, the rank value of the first block will be placed at the last.

**Step 8:** Combining all the code values as well as the rank values, if the total number of bits in the target stream is not a multiple of 8, then to make it so, at most 7 bits may have to be inserted. Insertion of these extra bits is to be started from the  $(N+1)^{th}$  position. So, a maximum of 7 right shifting operations may have to be performed in the  $(N+1)^{th}$  position, where that many 0's are inserted.

Consider a stream S=1010100101010010 of only 16 bits. We apply these steps to get the target stream T corresponding to S using the RSBP technique.

Following step 1, we decompose S into four 4-bit blocks taking bits four by four from the MSB, which are  $D_1=1010$ ,  $D_2=1001$ ,  $D_3=0101$  and  $D_4=0010$ . So, as per step 1,  $L=4$ .

Following step 2, we find that the total number of primes in the range of 0 to  $2^4-1=15$  is 6 (2, 3, 5, 7, 11, 13) and that of nonprimes is 10 (0, 1, 4, 6, 8, 9, 10, 12, 14, 15). So, to represent the position of a prime number the number of bits required is 3,

because since there are 6 primes, their positions range from 0 to 5. Similarly, to represent the position of a nonprime number the number of bits required is 4 because their positions range from 0 to 9 as there are 10 nonprime numbers.

Now, we apply the next three steps (3, 4, 5) for blocks  $D_1, D_2, D_3$  and  $D_4$ .

The decimal equivalent of  $D_1=1010$  is 10, which is the 6<sup>th</sup> nonprime. So, the code value of  $D_1$  is  $C_1=0$  and the rank is  $R_1=0110$ .

The decimal equivalent of  $D_2=1001$  is 9, which is the 5<sup>th</sup> nonprime. So, the code value of  $D_2$  is  $C_2=0$  and the rank is  $R_2=0101$ .

The decimal equivalent of  $D_3=0101$  is 5, which is the 2<sup>nd</sup> prime. So, the code value of  $D_3$  is  $C_3=1$  and the rank is  $R_3=010$ .

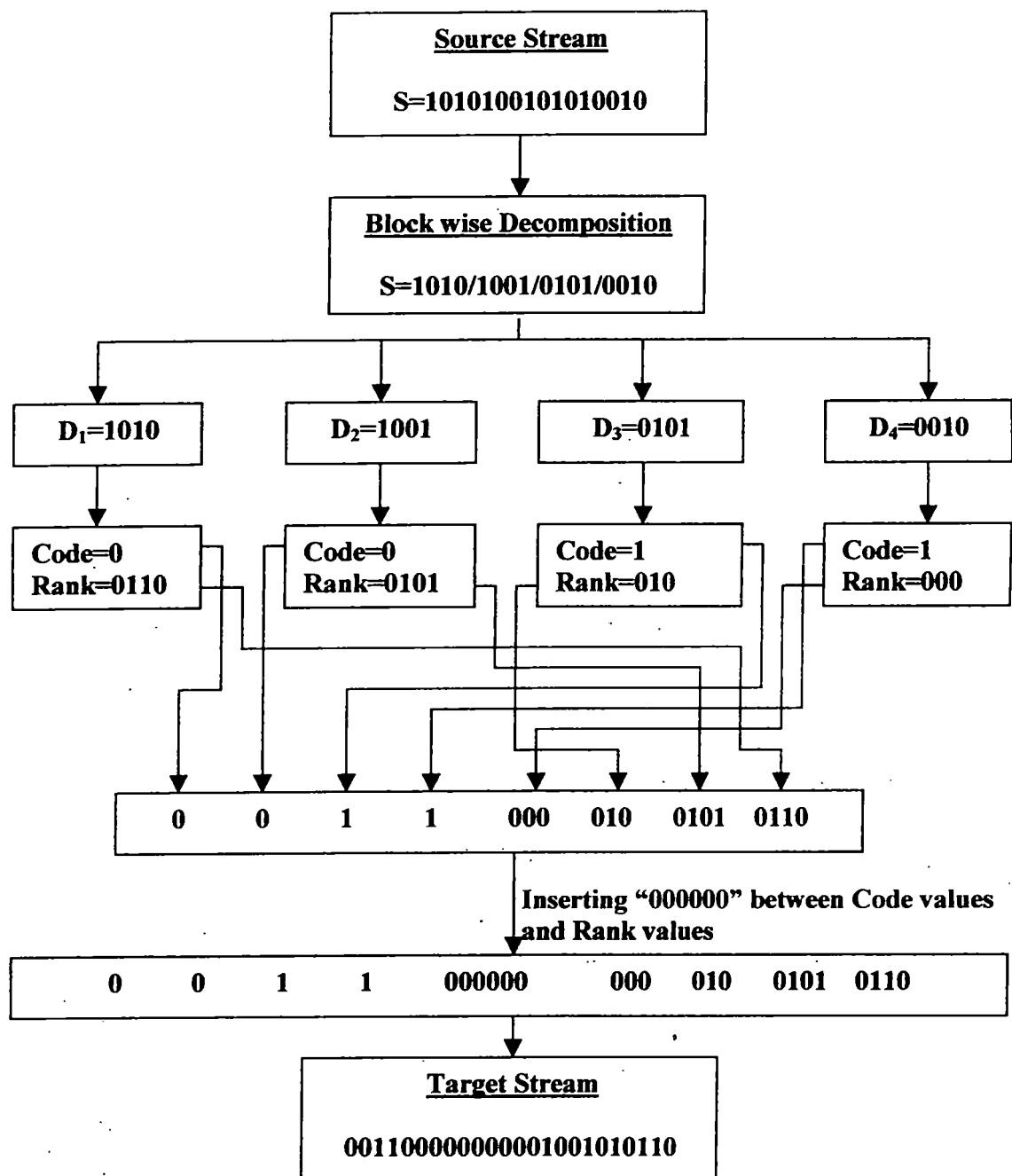
The decimal equivalent of  $D_4=0010$  is 2, which is the 0<sup>th</sup> prime. So, the code value of  $D_4$  is  $C_4=1$  and the rank is  $R_4=000$ .

Following step 6 and step 7, to form the target stream, first we put all the code values one by one starting from the MSB position to get 0/0/1/1 and they are followed by the rank values of all the blocks starting from the last, i.e., 000/010/0101/0110. Here “/” works just as the separator. So, combining these code values and rank values we obtain 001100001001010110, a stream of length 18.

Following step 8, to make the length a multiple of 8, a block “000000” is to be inserted between the code values and the rank values, so that the stream 0011/000000/00001001010110 is formed. Therefore corresponding to the 16-bit source stream  $S=1010100101010010$ , the 24-bit target stream is as follows:

$$T=001100000000001001010110.$$

Figure 6.2.1.1 gives a pictorial representation of this example.



**Figure 6.2.1.1**  
**Pictorial Representation of Encrypting  $S=1010100101010010$  using RSBP Technique**

### 6.2.2 The Decryption Technique

In the policy of encryption, the strategy of distributing the code values and the rank values of all the source blocks was adopted with the objective of having an unambiguous decryption. As the result of that, the policy of decryption discussed in this section offers an absolute correct outcome.

Following are the steps to be followed during the decryption:

**Step 1:** Get the unique block length from the key. Say, it is L.

**Step 2:** Calculate the total number of blocks generated from the source stream of bits. This calculation is done by the following:

Total Number of Blocks (B) = Source Stream Size / Unique Block Length,  
"/" denoting the integer division.

So, the first B number of bits, starting from position 0 (MSB position) to position (B-1) in the encrypted stream denotes the code values of B blocks.

**Step 3:** Calculate the total number of primes in the range of 0 to  $(2^L - 1)$ .

Say, it is P. Hence calculate how many maximum bits are required to express P in binary form. Say, it is X. Then  $X \geq \lfloor \log_2 P \rfloor + 1$ , where  $\lfloor \log_2 P \rfloor$  denotes the integral part of  $\log_2 P$ .

**Step 4:** Calculate the total number of nonprimes in the range of 0 to  $(2^L - 1)$ . Say, it is Q. Hence calculate how many maximum bits are required to express Q in binary form. Say, it is Y. Then  $Y \geq \lfloor \log_2 Q \rfloor + 1$ , where  $\lfloor \log_2 Q \rfloor$  denotes the integral part of  $\log_2 Q$ . It is mentionable here that  $Q = 2^L - P$ .

**Step 5:** Consider the MSB. It is the code value of the first source block.

If MSB=1,

1. Consider the last block of X bits, convert the binary number represented by this block of bits into the corresponding decimal, Say, it is M. Mark this block as being processed.

2. Find the  $M^{\text{th}}$  prime number in the series of natural numbers (with the assumption that the position of the first prime number is 0, not 1).
3. The L-bit binary number corresponding to the decimal prime number obtained in 2 is the first source block.
4. Mark the MSB as being processed.

If MSB=0,

1. Consider the last block of Y bits; convert the binary number represented by this block of bits into the corresponding decimal, Say, it is M. Mark this block as being processed.
2. Find the  $M^{\text{th}}$  prime number in the series of natural numbers (with the assumption that the position of the first prime number is 0, not 1).
3. The L-bit binary number corresponding to the decimal nonprime number obtained in 2 is the first source block.
4. Mark the MSB as being processed.

**Step 6:** Repeat step 7 and step 8 for  $(B-1)$  number of times for the values of I ranging from 1 to  $(B-1)$  as there are  $(B-1)$  more blocks left to be considered. Set  $I = 1$ .

**Step 7:** Consider the  $I^{\text{th}}$  bit from the MSB position. Let it be denoted by  $T_I$ .

If  $T_I = 1$ ,

1. Consider the first unprocessed block of P bits in the LSB-to-MSB direction, convert the binary number represented by this block of bits into the corresponding decimal, Say, it is M. Mark this block being processed.

2. Find the  $M^{\text{th}}$  prime number in the series of natural numbers (with the assumption that the position of the first prime number is 0, not 1).
3. The L-bit binary number corresponding to the decimal prime number obtained in 2 is the  $I^{\text{th}}$  source block.

If  $T_I = 0$ ,

1. Consider the first unprocessed block of Q bits in the LSB-to-MSB direction, convert the binary number represented by this block of bits into the corresponding decimal. Say, it is M. Mark this block being processed.
2. Find the  $M^{\text{th}}$  nonprime number in the series of natural numbers (with the assumption that the position of the first prime number is 0, not 1).
3. The L-bit binary number corresponding to the decimal nonprime number obtained in 2 is the  $I^{\text{th}}$  source block.

**Step 8:** Let  $I = I + 1$ .

**Step 9:** Concatenate all the blocks obtained so far in the sequence of their generation and this is the source stream.

The length of the source stream is  $(L * B)$  and accordingly  $L_T - (L * B)$  number of 0's in the positions between the code values and the target values in the target stream will remain being unmarked, as these 0's were inserted at the end of the encryption process;  $L_T$  being considered as the length of the target stream.

Continue with the same example for illustration, where the target stream obtained was  $T = 00110000000001001010110$ .

Now, from step 1, from the key we get the unique block length  $L = 4$ .

Following step 2, we obtain the total number of blocks as  $B = 16 / 4 = 4$ , as it is assumed to be known to the receiver that the source stream before being encrypted was of

length 16 bits. Therefore in the encrypted stream, the first four bits are the code values of four blocks.

Following step 3, we calculate the total number of primes in the range of 0 to 15 (i.e.,  $2^4 - 1$ ), which is  $P = 6$ , and to represent it by a binary number the maximum number of bits needed is  $X = 3$ .

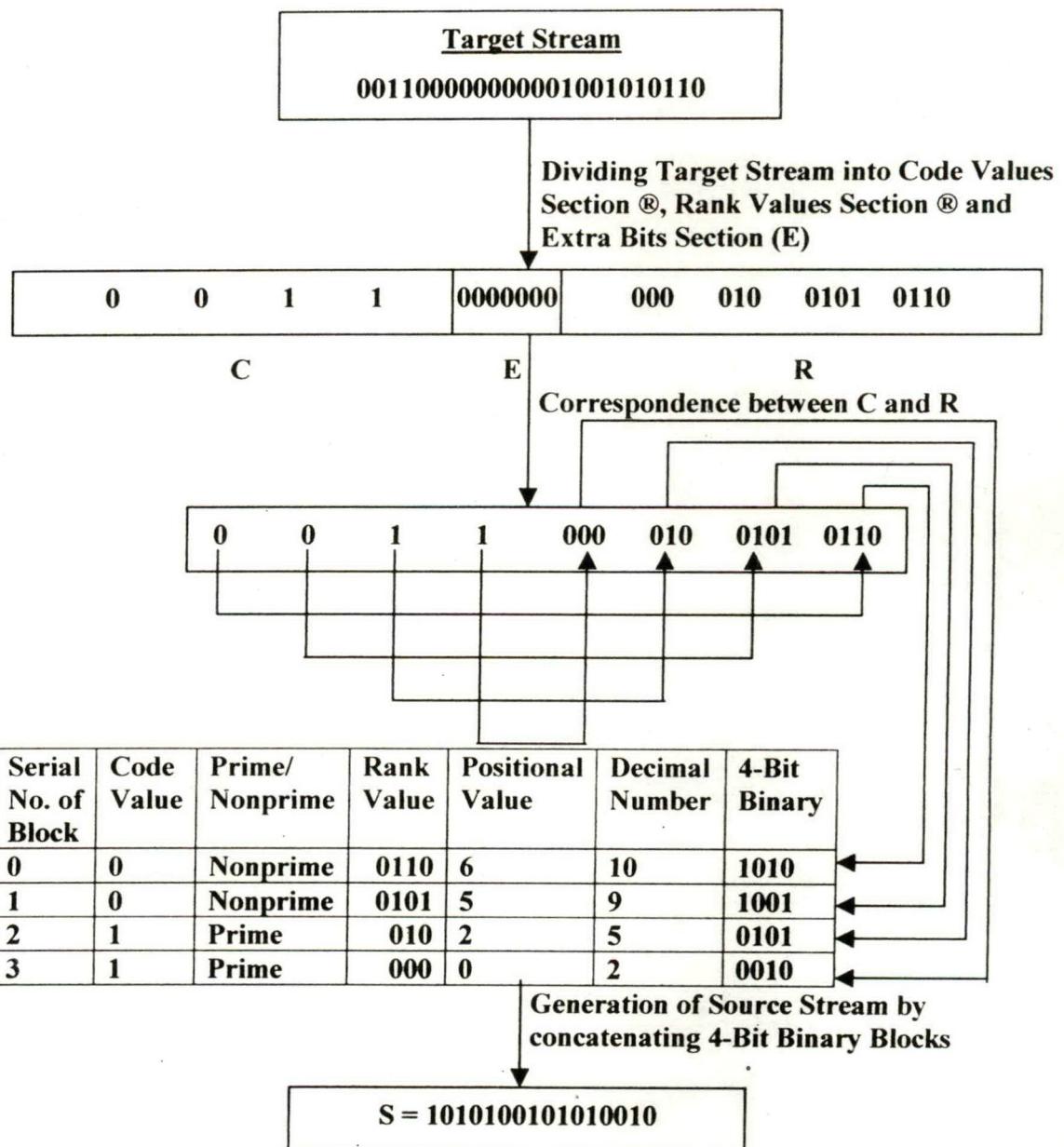
Similarly, following step 4, we calculate the total number of nonprimes in the range of 0 to 15 (i.e.,  $2^4 - 1$ ), which is  $P = 10$ , and to represent it by a binary number the maximum number of bits needed is  $Y = 4$ .

Now, following step 5, we find the MSB as 0, so that we are to consider the block of the last  $Y = 4$  number of bits, which is 0110, the decimal of which is  $M = 6$ . So, we are to find the 6<sup>th</sup> nonprime number in the series of natural numbers. It is 10 (assuming that 0 is the 0<sup>th</sup> nonprime, 1 is the 1<sup>st</sup> nonprime, and so on), the 4-bit binary of which is 1010. Hence the first source block is 1010.

Using step 6, we can say that step 7 and step 8 are to be repeated for 3 times as there are still 3 blocks left. Step 7 only does the job of moving from one block to another and, in fact, step 8 works in the same way as step 5. So, proceeding in the same way, we obtain the remaining blocks as 1001, 0101 and 0010.

Following step 9, we concatenate all the blocks in the same sequence of their generation to obtain the source stream  $S = 1010100101010010$ .

Figure 6.2.2.1 represents this example diagrammatically.



**Figure 6.2.2.1**  
**Pictorial Representation of Decrypting  $T = 001100000000001001010110$  using RSBP Technique**

### 6.3 Implementation

For the purpose of real implementation of the RSBP technique, we take the plaintext “Local Area Network” that was also considered in the RPMS technique in chapter 5 to have a comparative look at the two ciphertexts.

Section 6.3.1 shows the process of encryption and the process of decryption is described in section 6.3.2.

### **6.3.1 Implementation of Encryption Technique of RSBP**

Corresponding to the plaintext “Local Area Network” the 144-bit stream obtained in section 5.3 in chapter 5 is as follows:

S=01001100/01101111/01100011/01100001/01101100/00100000/01000001/01110010/01100101/01100001/00100000/01001110/01100101/01110100/01110111/01101111/01110010/01101011, “/” being used as the separator of two consecutive bytes.

Now, the unique block size is taken as 8 bits. In the range of 0 to 255 ( $= 2^8 - 1$ ), there are 54 primes and 202 nonprimes. So, the rank value corresponding to the code value of 1 (prime) should be of 6 bits and the same corresponding to the code value of 0 (nonprime) should be of 8 bits. Table 6.3.1.1 shows the status of all the blocks in terms of the RSBP technique.

**Table 6.3.1.1**  
**Status of Different Blocks for the Message Stream of Bits of “Local Area Network”**

Block Serial No.	Block Value (Binary)	Block Value (Decimal)	Prime/ Nonprime	Code Value	Positional Value	Rank Value
0	01001100	76	Nonprime	0	55	00110111
1	01101111	111	Nonprime	0	82	01010010
2	01100011	99	Nonprime	0	74	01001010
3	01100001	97	Prime	1	24	011000
4	01101100	108	Nonprime	0	80	01010000
5	00100000	32	Nonprime	0	21	00010101
6	01000001	65	Nonprime	0	47	00101111
7	01110010	114	Nonprime	0	84	01010100
8	01100101	101	Prime	1	25	011001
9	01100001	97	Prime	1	24	011000
10	00100000	32	Nonprime	0	21	00010101
11	01001110	78	Nonprime	0	57	00111001
12	01100101	101	Prime	1	25	011001
13	01110100	116	Nonprime	0	86	01010110
14	01110111	119	Nonprime	0	89	01011001
15	01101111	111	Nonprime	0	82	01010010
16	01110010	114	Nonprime	0	84	01010100
17	01101011	107	Prime	1	27	011011

From the column corresponding to “Code Value” in the table 6.3.1.1, we get code values of all the blocks and placing these values one after the other we get the stream C = 000100001100100001 of the length of 18 bits.

From the column corresponding to “Rank Value” in the same table, we take rank values of blocks in the opposite direction, which means that the rank value of the last block is taken first, followed by the rank value of the previous block, and so. In this way, all these rank values are placed together to form the stream R as the following: 0110110101000101001001011001010101100110010011100100010101011000011001

01010100001011100010101010000011000010010100101001000110111, the length of which is 134 bits.

Now, since the combined length of the two streams C and R is  $18 + 134 = 152$  bits, which is already a multiple of 8, so that there is no need to insert any extra block of bits between C and R. to form the target stream T. Hence  $T = C + R$ , where “+” denotes the concatenation operator.

Therefore the 152-bit target stream T corresponding to the 136-bit source stream S is as follows:

00010000110010000101101101010001010010010110010101100110010011100100  
01010101100001100101010000101111000101010100000110000100101001010010  
00110111.

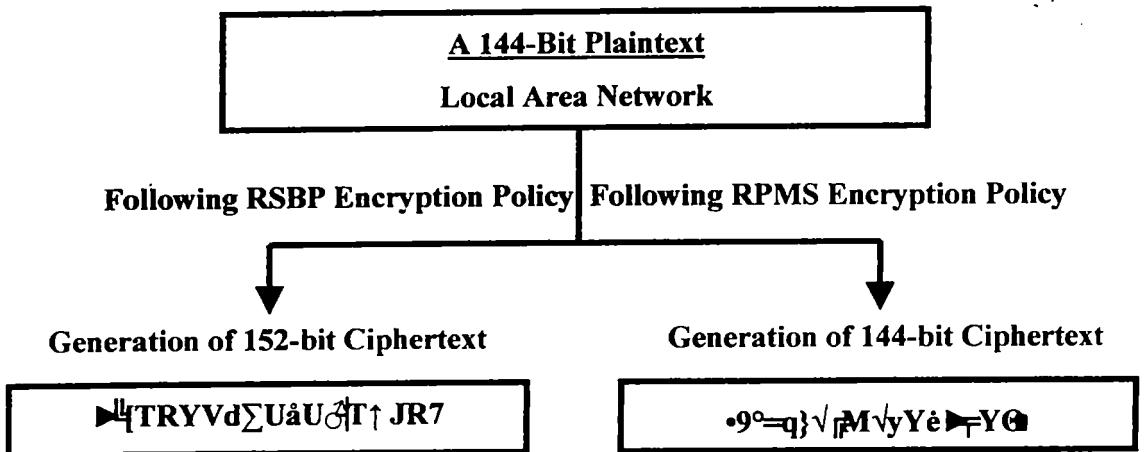
For the purpose of converting it into the ciphertext, we re-write it in the byte wise manner, so that it looks like the following:

00010000/11001000/01011011/01010100/01010010/01011001/01010110/01100100/111  
00100/01010101/10000110/01010101/00001011/11000101/01010100/00011000/010010  
10/01010010/00110111.

Finally, we obtain the ciphertext, which looks like the following:

►TRYVdΣUåU©T↑JR7

Figure 6.3.1.1 pictorially shows the ciphertexts obtained for “Local Area Network” for this RSBP technique as well as for the RPMS technique discussed in chapter 5.



**Figure 6.3.1.1**  
**Ciphertexts for RSBP and RPMS for Plaintext “Local Area Network”**

### 6.3.2 Implementation of Decryption Technique of RSBP

The process of decryption in this technique of RSBP is started with the 152-bit ciphertext, which, in the form of a binary stream is as follows:

0001000011001000010110110101000101001001011001010101100110010011100100  
0101010110000110010101010000101111000101010100000110000100101001010010  
00110111.

Now, combining the knowledge of the key and the size of the source file, it is known to the receiver of the ciphertext that the total number of blocks that had been created from the source stream prior to the encryption is 18. Hence out of the 152 bits in the received ciphertext, the first 18 bits stand for the code values of the 18 blocks.

Therefore the received stream of bits can be decomposed into the following two sections:

#### Code Value Section:

000100001100100001

#### Rank Value Section (Including Extra Bits, if any):

011011010101000101001001011001010101100110010011100100010101011000  
011001010101000010111100010101010000011000010010100101001000110111

Now, the receiver calculates that the total numbers of possible primes and nonprimes in the range of 0 to 255 ( $= 2^8 - 1$ ) are 54 and 202 respectively, so that blocks of 6 bits and 8 bits respectively are enough to represent their positions using bits.

Now, the scanning of code values from the beginning starts one after the other and accordingly the corresponding rank values are also consider using the following steps:

**Step 1:** We consider the first unmarked code bit in the target stream, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is "00110111". From this we conclude that it indicates the 55<sup>th</sup> nonprime number, which is 76, i.e., 01001100 in 8-bit format. We mark the code bit and the block of the rank value as follows:

0`0010000110010000101101101010100010100100101100101010  
11001100100111001000101010110000110010101010000101110  
001010101010000110000100101001010010(00110111) ✓

**Step 2:** We consider the first unmarked code bit in the stream of step 1, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is "1010010". From this we conclude that it indicates the 82<sup>nd</sup> nonprime number, which is 111, i.e., 01101111 in 8-bit format. We mark the code bit and the block of the rank value as follows:

0`0`0`010000110010000101101101010001010010010110010101  
011001100100111001000101010110000110010101010000101111  
00010101010100001100001001010(01010010) ✓ (00110111) ✓

**Step 3:** We consider the first unmarked code bit in the stream of step 2, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is "01001010". From this we conclude that it indicates the 74<sup>th</sup> nonprime number, which is 99, i.e., 01100011 in 8-bit format. We mark the code bit and the block of the rank value as follows:

0`0`0`0`1000011001000010110110101000101001001011001010  
101100110010011100100010101011000011001010101000010111  
10001010101010000011000 (01001010) ✓ (01010010) ✓  
(00110111)

**Step 4:** We consider the first unmarked code bit in the stream of step 3, which is 1, so that the first unmarked 6-bit block is considered from the LSB position, which is “011000”. From this we conclude that it indicates the 24<sup>th</sup> prime number, which is 97, i.e., 01100001 in 8-bit format. We mark the code bit and the block of the rank value as follows:

**0** **0** **1** **0** **000011001000010110110101000101001001011001010  
101100110010011100100010101011000011001010101000010111  
100010101010000**(011000)** **(01001010)** **(01010010)** **(00110111)****

**Step 5:** We consider the first unmarked code bit in the stream of step 4, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is “01010000”. From this we conclude that it indicates the 80<sup>th</sup> nonprime number, which is 108, i.e., 01101100 in 8-bit format. We mark the code bit and the block of the rank value as follows:

**0** **0** **1** **0** **0001100100001011011010100010100100101100101  
010110011001001110010001010101100001100101010100001011  
1100010101**(01010000)** **(011000)** **(01001010)** **(01010010)** **(00110111)****

**Step 6:** We consider the first unmarked code bit in the stream of step 5, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is “00010101”. From this we conclude that it indicates the 21<sup>st</sup> nonprime number, which is 32, i.e., 00100000 in 8-bit format. We mark the code bit and the block of the rank value as follows:

**0** **0** **1** **0** **0** **00110010000101101101010001010010010110010  
101011001100100111001000101010110000110010101010000101  
111**(00010101)** **(01010000)** **(011000)** **(01001010)** **(01010010)**  
**(00110111)****

**Step 7:** We consider the first unmarked code bit in the stream of step 6, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is “00101111”. From this we conclude that it indicates the 47<sup>th</sup> nonprime number, which is 65, i.e., 01000001 in 8-bit format. We mark the code bit and the block of the rank value as follows:

0`0`0`1`0`0`0`0`0`11001000010110110101000101001001011001  
 01010110011001001110010001010101100001100101010100(001  
 01111)` (00010101)` (01010000)` (011000)` (01001010)`  
 (01010010)` (00110111)`

**Step 8:** We consider the first unmarked code bit in the stream of step 7, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is “01010100”. From this we conclude that it indicates the 84<sup>th</sup> nonprime number, which is 114, i.e., 01110010 in 8-bit format. We mark the code bit and the block of the rank value as follows:

0`0`0`1`0`0`0`0`0`11001000010110110101000101001001011001  
 010101100110010011100100010101011000011001(01010100)(00  
 101111)` (00010101)` (01010000)` (011000)` (01001010)`  
 (01010010)` (00110111)`

**Step 9:** We consider the first unmarked code bit in the stream of step 8, which is 1, so that the first unmarked 6-bit block is considered from the LSB position, which is “011001”. From this we conclude that it indicates the 25<sup>th</sup> prime number, which is 101, i.e., 01100101 in 8-bit format. We mark the code bit and the block of the rank value as follows:

0`0`0`1`0`0`0`0`1`100100001011011010100010100100101100  
 1010101100110010011100100010101011000(011001)`  
 (01010100)` (00101111)` (00010101)` (01010000)` (011000)`  
 (01001010)` (01010010)` (00110111)`

**Step 10:** We consider the first unmarked code bit in the stream of step 9, which is 1, so that the first unmarked 6-bit block is considered from the LSB position, which is “011000”. From this we conclude that it indicates the 24<sup>th</sup> prime number, which is 97, i.e., 01100001 in 8-bit format. We mark the code bit and the block of the rank value as follows:

0<sup>✓</sup>0<sup>✓</sup>0<sup>✓</sup>1<sup>✓</sup>0<sup>✓</sup>0<sup>✓</sup>0<sup>✓</sup>1<sup>✓</sup>1<sup>✓</sup>0010000101101101010001010010010110  
 01010101100110010011100100010101(011000)<sup>✓</sup>(011001)<sup>✓</sup>  
 (01010100)<sup>✓</sup>(00101111)<sup>✓</sup>(00010101)<sup>✓</sup>(01010000)<sup>✓</sup>(011000)<sup>✓</sup>  
 (01001010)<sup>✓</sup>(01010010)<sup>✓</sup>(00110111)<sup>✓</sup>

**Step 11:** We consider the first unmarked code bit in the stream of step 10, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is “00010101”. From this we conclude that it indicates the 21<sup>st</sup> nonprime number, which is 32, i.e., 00100000 in 8-bit format. We mark the code bit and the block of the rank value as follows:

0<sup>✓</sup>0<sup>✓</sup>0<sup>✓</sup>1<sup>✓</sup>0<sup>✓</sup>0<sup>✓</sup>0<sup>✓</sup>1<sup>✓</sup>1<sup>✓</sup>0<sup>✓</sup>01000010110110101000101001001011.  
 0010101011001100100111001(00010101)<sup>✓</sup>(011000)<sup>✓</sup>(011001)<sup>✓</sup>  
 (01010100)<sup>✓</sup>(00101111)<sup>✓</sup>(00010101)<sup>✓</sup>(01010000)<sup>✓</sup>(011000)<sup>✓</sup>  
 (01001010)<sup>✓</sup>(01010010)<sup>✓</sup>(00110111)<sup>✓</sup>

**Step 12:** We consider the first unmarked code bit in the stream of step 11, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is “00111001”. From this we conclude that it indicates the 57<sup>th</sup> nonprime number, which is 78, i.e., 01001110 in 8-bit format. We mark the code bit and the block of the rank value as follows:

0<sup>✓</sup>0<sup>✓</sup>0<sup>✓</sup>1<sup>✓</sup>0<sup>✓</sup>0<sup>✓</sup>0<sup>✓</sup>1<sup>✓</sup>1<sup>✓</sup>0<sup>✓</sup>0<sup>✓</sup>1000010110110101000101001001011.  
 00101010110011001(00111001)<sup>✓</sup>(00010101)<sup>✓</sup>(011000)<sup>✓</sup>  
 (011001)<sup>✓</sup>(01010100)<sup>✓</sup>(00101111)<sup>✓</sup>(00010101)<sup>✓</sup>(01010000)<sup>✓</sup>  
 (011000)<sup>✓</sup>(01001010)<sup>✓</sup>(01010010)<sup>✓</sup>(00110111)<sup>✓</sup>

**Step 13:** We consider the first unmarked code bit in the stream of step 12, which is 1, so that the first unmarked 6-bit block is considered from the LSB position, which is “011001”. From this we conclude that it indicates the 25<sup>th</sup> prime number, which is 101, i.e., 01100101 in 8-bit format. We mark the code bit and the block of the rank value as follows:

**Step 14:** We consider the first unmarked code bit in the stream of step 13, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is “01010110”. From this we conclude that it indicates the 86<sup>th</sup> nonprime number, which is 116, i.e., 01110100 in 8-bit format. We mark the code bit and the block of the rank value as follows:

**Step 15:** We consider the first unmarked code bit in the stream of step 14, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is “01011001”. From this we conclude that it indicates the 89<sup>th</sup> nonprime number, which is 119, i.e., 01110111 in 8-bit format. We mark the code bit and the block of the rank value as follows:

**Step 16:** We consider the first unmarked code bit in the stream of step 15, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is “01010010”. From this we conclude that it indicates the 82<sup>nd</sup> nonprime number, which is 111, i.e., 01101111 in 8-bit format. We mark the code bit and the block of the rank value as follows:

**Step 17:** We consider the first unmarked code bit in the stream of 16, which is 0, so that the first unmarked 8-bit block is considered from the LSB position, which is “01010100”. From this we conclude that it indicates the 84<sup>th</sup> nonprime number, which is 114, i.e., 01110010 in 8-bit format. We mark the code bit and the block of the rank value as follows:

**Step 18:** We consider the first unmarked code bit in the stream of step 17, which is 1, so that the first unmarked 8-bit block is considered from the LSB position, which is “011011”. From this we conclude that it indicates the 27<sup>th</sup> prime number, which is 107, i.e., 01101011 in 8-bit format. We mark the code bit and the block of the rank value as follows:

$0^{\vee}0^{\vee}0^{\vee}1^{\vee}0^{\vee}0^{\vee}0^{\vee}0^{\vee}1^{\vee}1^{\vee}0^{\vee}0^{\vee}1^{\vee}0^{\vee}0^{\vee}0^{\vee}0^{\vee}1^{\vee}(011011)^{\vee}(01010100)$   
 $\vee(01010010)^{\vee}(01011001)^{\vee}(01010110)^{\vee}(011001)^{\vee}(00111001)^{\vee}$   
 $(00010101)^{\vee}(011000)^{\vee}(011001)^{\vee}(01010100)^{\vee}(00101111)^{\vee}$

(00010101) ✓ (01010000) ✓ (011000) ✓ (01001010) ✓ (01010010) ✓  
(00110111) ✓

Since in the source stream there are 18 blocks, the decryption process is over and incidentally there is no unmarked block of bits left because while encrypting no extra bits were needed to place between the code values and the rank values.

Now, combining all the 8-bit values generated from step 1 to step 18, we get the following stream:

S=01001100/01101111/01100011/01100001/01101100/00100000/01000001/01110010/01100101/01100001/00100000/01001110/01100101/01110100/01110111/01101111/01110010/0010/01101011.

Converting the bytes of S into characters, the plaintext is regenerated as “**Local Area Network**”.

#### 6.4 Results

In this section different sample files have been encrypted using the RSBP technique for blocks of unique length of only 8 bits. One characteristic of using small blocks is the fast implementation, but as it is being discussed throughout the thesis, that for ensuring the security of the highest level, blocks of lengths at least of 64 bits are to be considered. The overall performance in terms of ensuring security increases with the increment in block size. This improvement in performance is reflected in chi square values and frequency distribution tests [44, 54].

The execution time being so less because of considering small blocks, this section emphasizes on the alteration of file size rather than on the encryption/decryption time.

Section 6.4.1 shows results of the encryption/decryption time, and the chi square value, this section also analyzes the alteration in file size. Section 6.4.2 depicts pictorial result of the frequency distribution tests, section 6.4.3 presents results of the comparison with the RSA system.

#### 6.4.1 Result of Encryption/Decryption Time, Chi Square Value and File Size Alteration

Section 6.4.1.1 to section 6.4.1.5 presents results for files of different categories, namely, .EXE, .COM, .DLL, .SYS, and .CPP. Each section consists of tables with information on the source file name and size, the encrypted file size, the encryption time, the decryption time, average size alteration, and the chi square value, with the degree of freedom [55, 56].

##### 6.4.1.1 Result for EXE Files

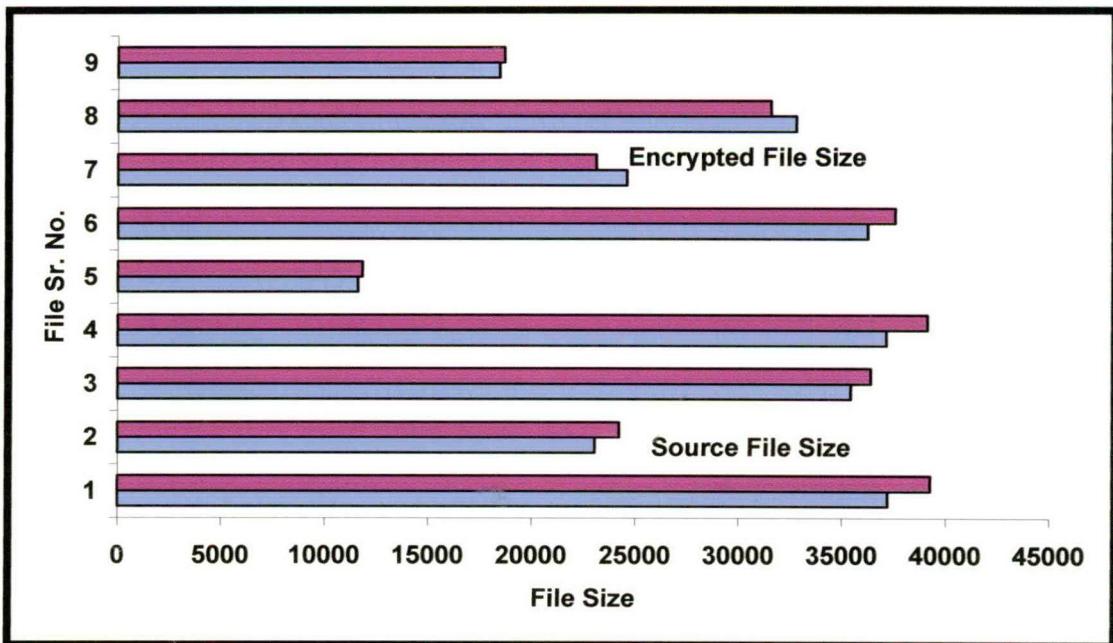
Table 6.4.1.1.1 presents the result for .EXE files. Here nine files have been considered. There sizes are in the range of 11611 bytes to 37220 bytes. Sizes of the encrypted files are in the range of 11816 to 39262. Out of nine, in seven cases sizes increase through encryption, whereas in remaining two cases reduction in size occurs. On the average, 2.03% of size is expanded. The encryption time ranges between 0.5500 seconds to 1.3700 seconds. whereas the decryption time ranges from 0.0500 seconds to 0.2800 seconds. The Chi Square value for each of the cases lies in the range of 32199 to 1501384 with the degree of freedom ranging from 248 to 255.

**Table 6.4.1.1.1  
Result for EXE Files for RSBP Techniques**

Source File	Source File Size	Encrypted File Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom	Average Size Alteration
<i>TLIB.EXE</i>	37220	39262	1.3200	0.1700	193871	255	+2.03 %
<i>UNZIP.EXE</i>	23044	24222	0.9900	0.1700	32199	255	
<i>RPPO.EXE</i>	35425	36390	1.2700	0.2700	108005	255	
<i>PRIME.EXE</i>	37152	39124	1.3700	0.2200	106441	255	
<i>TCDEF.EXE</i>	11611	11816	0.5500	0.0500	55944	254	
<i>TRIANGLE.EXE</i>	36242	37550	1.3100	0.1600	103763	255	
<i>PING.EXE</i>	24576	23107	0.9300	0.1100	1501384	248	
<i>NETSTAT.EXE</i>	32768	31544	1.1500	0.2200	422135	255	
<i>CLIPBRD.EXE</i>	18432	18673	0.8300	0.2800	171438	255	

Figure 6.4.1.1.1 graphically compares the size between the source file and the encrypted file for each pair for .EXE files. Each horizontal bar of color blue stands for the source size, whereas that of color brown stands for the encrypted file size. It is observed

from the figure that whatever be the case, size reduction or size expansion, never there exists a huge change in size.



**Figure 6.4.1.1.1**  
**A Comparison between Source Size and Encrypted File Size for**  
**EXE Files in RSBP Technique**

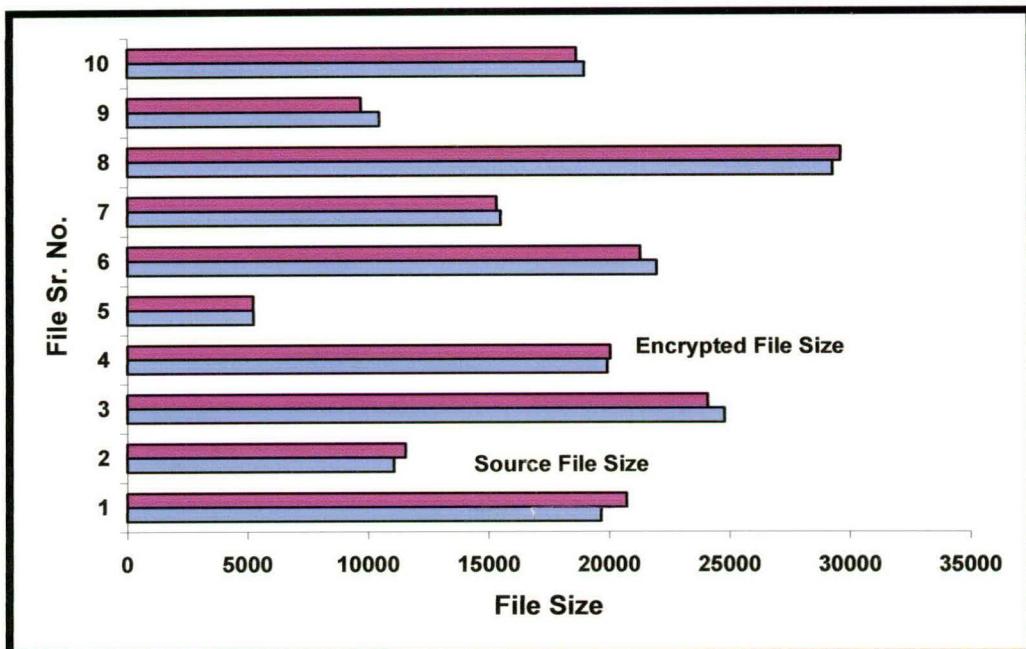
#### 6.4.1.2 Result for COM Files

Table 6.4.1.2.1 presents the result for .COM files. Here ten files have been considered. There sizes are in the range of 5239 bytes to 29271 bytes. Sizes of the encrypted files are in the range of 5225 to 29612. Out of ten, in only four cases sizes increase through encryption, whereas in remaining six cases reduction in size occurs. On the average, 0.36% of size is reduced. The encryption time ranges between 0.2800 seconds to 1.0900 seconds, whereas the decryption time ranges from 0.0500 seconds to 0.2200 seconds. The Chi Square value for each of the cases lies in the range of 53441 to 667403 with the degree of freedom ranging from 230 to 255.

**Table 6.4.1.2.1**  
**Result for COM Files for RSBP Technique**

Source File	Source File Size	Encrypted File Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom	Average Size Alteration
<i>EMSTEST.COM</i>	19664	20728	0.9300	0.0600	59855	255	-0.36%
<i>THELP.COM</i>	11072	11552	0.6100	0.0500	53441	250	
<i>WIN.COM</i>	24791	24101	0.9900	0.1700	434812	252	
<i>KEYB.COM</i>	19927	20045	0.8800	0.2200	132264	255	
<i>CHOICE.COM</i>	5239	5225	0.2800	0.1100	72041	232	
<i>DISKCOPY.COM</i>	21975	21299	0.9300	0.1100	325978	254	
<i>DOSKEY.COM</i>	15495	15325	0.6100	0.1100	179966	253	
<i>MODE.COM</i>	29271	29612	1.0900	0.2200	147782	255	
<i>MORE.COM</i>	10471	9708	0.4900	0.0500	667403	230	
<i>SYS.COM</i>	18967	18633	0.8700	0.1700	232277	254	

Figure 6.4.1.2.1 graphically compares the size between the source file and the encrypted file for each pair for .COM files. Each horizontal bar of color blue stands for the source size, whereas that of color brown stands for the encrypted file size. Here also it is observed from the figure that whatever be the case, size reduction or size expansion, never there exists a huge change in size.



**Figure 6.4.1.2.1**  
**A Comparison between Source Size and Encrypted File Size for**  
**COM Files in RSBP Technique**

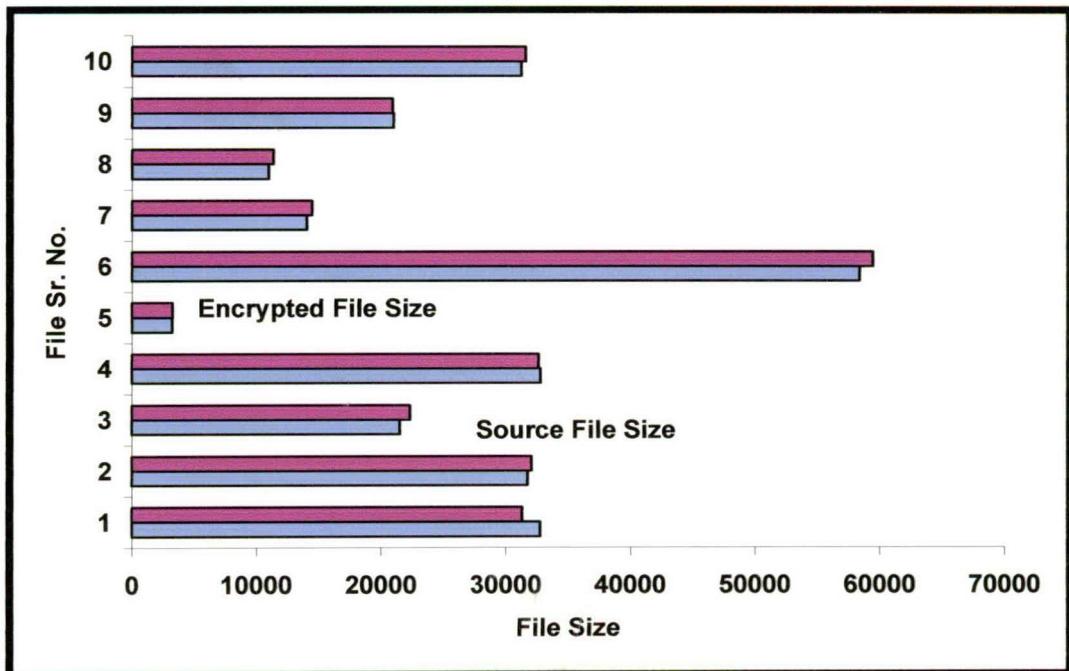
#### 6.4.1.3 Result for *DLL* Files

Table 6.4.1.3.1 presents the result for .DLL files. Here ten files have been considered. There sizes are in the range of 3216 bytes to 58368 bytes. Sizes of the encrypted files are in the range of 3251 to 59425. Out of ten, in six cases sizes increase through encryption, whereas in remaining four cases reduction in size occurs. On the average, 0.65% of size is expanded. The encryption time ranges between 0.1600 seconds to 1.6500 seconds, whereas the decryption time ranges from 0.0000 seconds to 0.2200 seconds. The Chi Square value for each of the cases lies in the range of 15369 to 725753 with the degree of freedom ranging from 217 to 255.

**Table 6.4.1.3.1**  
**Result for DLL Files for RSBP Technique**

Source File	Source File Size	Encrypted File Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom	Average Size Alteration
<i>SNMPAPI.DLL</i>	32768	31311	1.1000	0.1600	725753	253	+0.65%
<i>KPSHARP.DLL</i>	31744	32060	1.1000	0.1600	231924	254	
<i>WINSOCK.DLL</i>	21504	22309	0.8800	0.1100	97233	252	
<i>SPWHPT.DLL</i>	32792	32634	1.1600	0.1100	171222	255	
<i>HIDCI.DLL</i>	3216	3251	0.1600	0.0000	15369	217	
<i>PFPICK.DLL</i>	58368	59425	1.6500	0.2200	244965	255	
<i>NDDEAPI.DLL</i>	14032	14453	0.5500	0.0500	63622	249	
<i>NDDENB.DLL</i>	10976	11349	0.4400	0.0500	68331	251	
<i>ICCCODES.DLL</i>	20992	20927	0.7100	0.0600	158949	252	
<i>KPSCALE.DLL</i>	31232	31575	1.1500	0.1600	208467	255	

Figure 6.4.1.3.1 graphically compares the size between the source file and the encrypted file for each pair for .DLL files. Each horizontal bar of color blue stands for the source size, whereas that of color brown stands for the encrypted file size. Here also it is observed from the figure that whatever be the case, size reduction or size expansion, never there exists a huge change in size.



**Figure 6.4.1.3.1**  
**A Comparison between Source Size and Encrypted File Size for DLL Files in RSBP Technique**

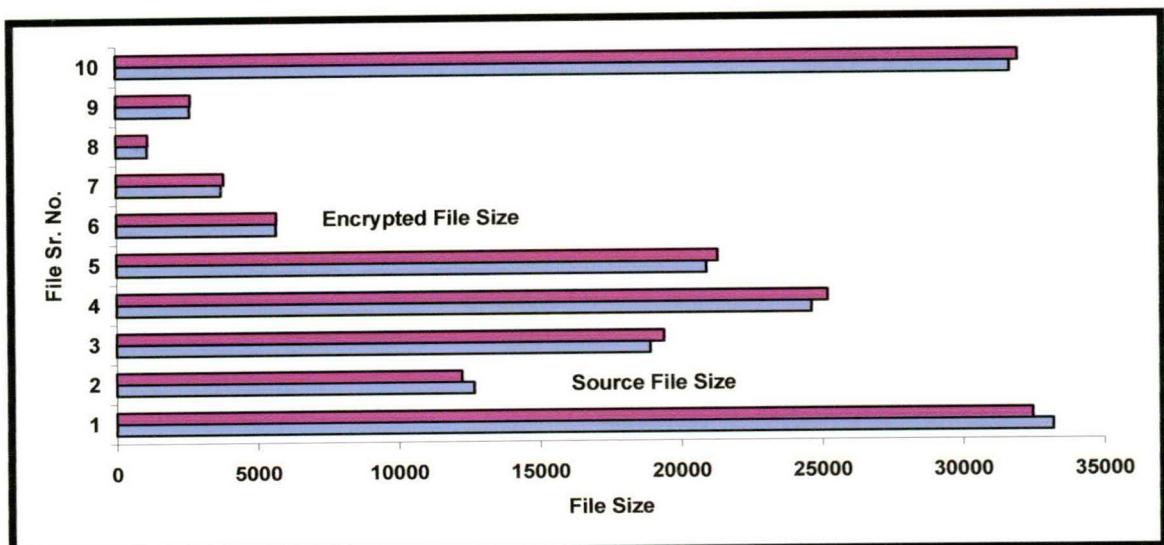
#### 6.4.1.4 Result for SYS Files

Table 7.4.1.4.1 presents the result for .SYS files. Here ten files have been considered. Their sizes are in the range of 5664 bytes to 33191 bytes. Sizes of the encrypted files are in the range of 5673 to 32453. Out of ten, in eight cases sizes increase through encryption, whereas in remaining two cases reduction in size occurs. On the average, 0.46% of size is expanded. The encryption time ranges between 0.1100 seconds to 1.1000 seconds, whereas the decryption time ranges from 0.0000 seconds to 0.1600 seconds. The Chi Square value for each of the cases lies in the range of 2755 to 830584 with the degree of freedom ranging from 165 to 255.

**Table 6.4.1.4.1**  
**Result for SYS Files for RSBP Technique**

Source File	Source File Size	Encrypted File Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom	Average Size Alteration
HIMEM.SYS	33191	32453	1.1000	0.1600	830584	255	+0.46%
RAMDRIVE.SYS	12663	12229	0.4400	0.0600	140587	241	
USBD.SYS	18912	19384	0.7100	0.1100	69223	255	
CMD640X.SYS	24626	25200	0.8800	0.1100	156293	255	
CMD640X2.SYS	20901	21300	0.9300	0.0500	163185	255	
REDBOOK.SYS	5664	5673	0.2800	0.0000	23680	230	
IFSHLP.SYS	3708	3807	0.2700	0.0500	22751	237	
ASPI2HLP.SYS	1105	1123	0.1100	0.0000	2755	165	
DBLBUFF.SYS	2614	2642	0.1100	0.0500	8396	215	
CCPORT.SYS	31680	31965	1.0900	0.1600	157014	255	

Figure 6.4.1.4.1 graphically compares the size between the source file and the encrypted file for each pair for .SYS files. Each horizontal bar of color blue stands for the source size, whereas that of color brown stands for the encrypted file size. Here also it is observed from the figure that whatever be the case, size reduction or size expansion, never there exists a huge change in size.



**Figure 6.4.1.4.1**  
**A Comparison between Source Size and Encrypted File Size for  
SYS Files in RSBP Technique**

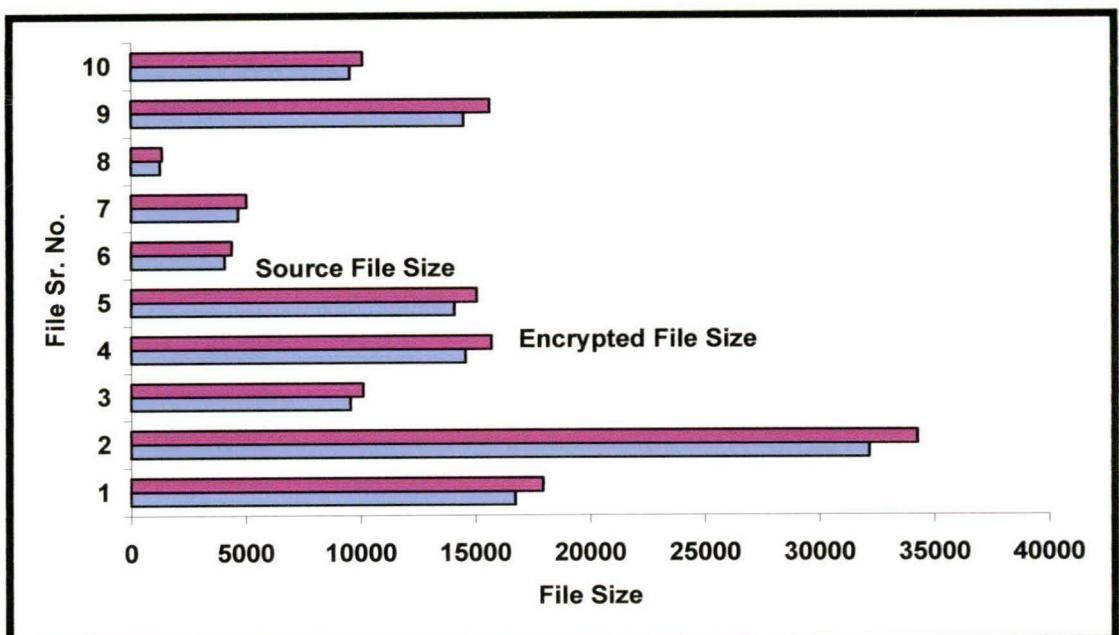
#### 6.4.1.5 Result for CPP Files

Table 6.4.1.4.1 presents the result for .CPP files. Here ten files have been considered. Their sizes are in the range of 1257 bytes to 32150 bytes. Sizes of the encrypted files are in the range of 1338 to 34250. Out of ten, in all cases, sizes increase through encryption. On the average, the highest among the five categories of files, expansion occurs, which is 6.92%. The encryption time ranges between 0.0600 seconds to 1.0400 seconds, whereas the decryption time ranges from 0.0000 seconds to 0.2200 seconds. The Chi Square value for each of the cases lies in the range of 2088 to 469578 with the degree of freedom ranging from 69 to 90.

**Table 6.4.1.5.1  
Result for CPP Files for RSBP Technique**

Source File	Source File Size	Encrypted File Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom	Average Size Alteration
<i>BRICKS.CPP</i>	16723	17934	0.5400	0.1100	162768	88	+6. 92%
<i>PROJECT.CPP</i>	32150	34250	1.0400	0.2200	469578	90	
<i>ARITH.CPP</i>	9558	10101	0.3300	0.0600	40089	77	
<i>START.CPP</i>	14557	15690	0.4400	0.1100	182263	88	
<i>CHARTCOM.CPP</i>	14080	15039	0.6000	0.0500	212197	84	
<i>BITIO.CPP</i>	4071	4381	0.1600	0.0000	13544	70	
<i>MAINC.CPP</i>	4663	5024	0.1600	0.0000	24048	83	
<i>TTEST.CPP</i>	1257	1338	0.0600	0.0000	2088	69	
<i>DO.CPP</i>	14481	15614	0.6000	0.0500	195541	88	
<i>CAL.CPP</i>	9540	10083	0.4400	0.0600	34129	77	

Figure 6.4.1.5.1 graphically compares the size between the source file and the encrypted file for each pair for .CPP files. Each horizontal bar of color blue stands for the source size, whereas that of color brown stands for the encrypted file size. Here also it is observed from the figure that whatever be the case, size reduction or size expansion, never there exists a huge change in size. But even through a glance at the figure one can understand that each brown bars are longer than the corresponding blue bar since in each of the cases file size has been increased.

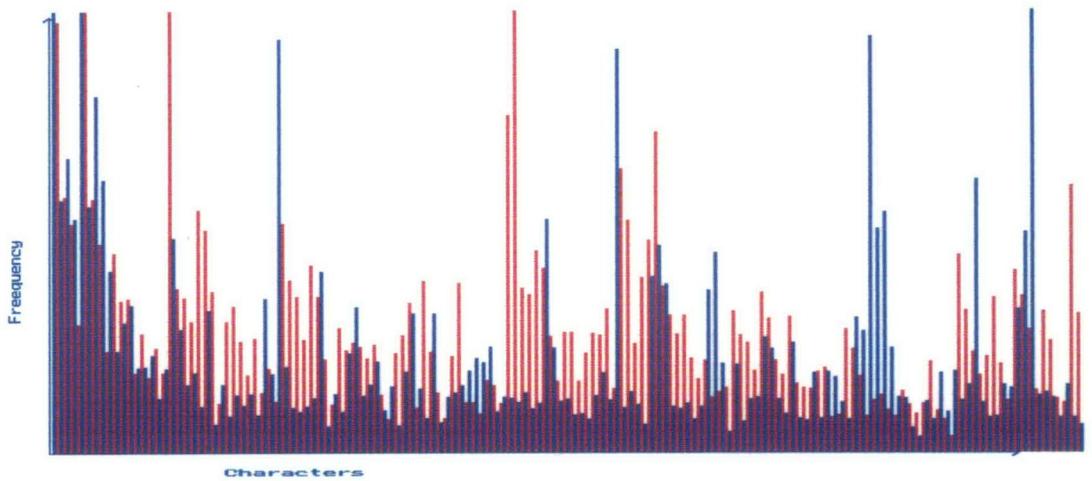


**Figure 6.4.1.5.1**  
**A Comparison between Source File Size and Encrypted File Size for CPP Files in RSBP Technique**

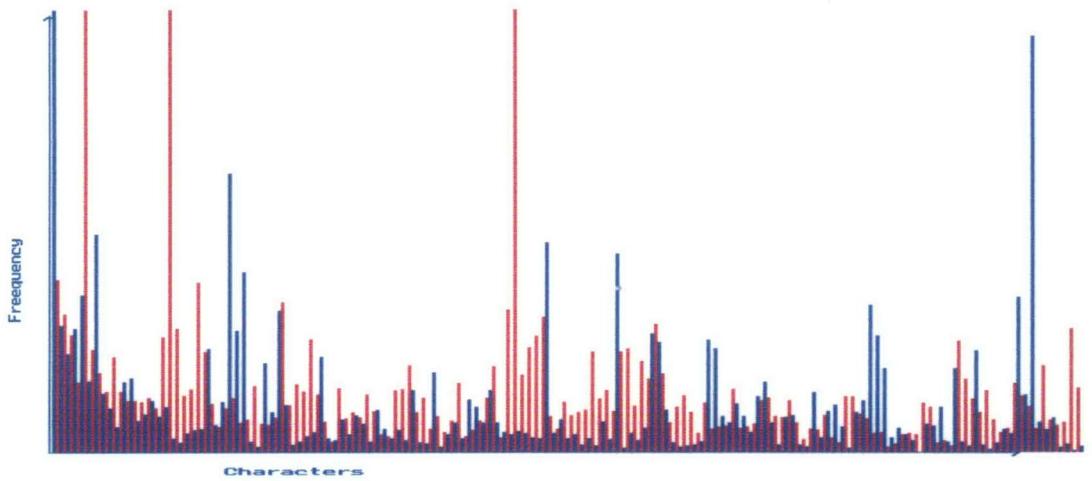
#### 6.4.2 Result for Frequency Distribution Tests

The frequency distribution tests have been performed for all types of sample files under consideration. It is seen for all cases that the characters in the encrypted files are well distributed, which indicates that the technique proposed here is quite compatible with existing techniques. The red bars represent frequencies of characters in the encrypted file and those in blue color represent frequencies of characters in the source file. The frequencies of characters in encrypted files are evenly distributed. Therefore the source and the corresponding encrypted file are heterogeneous in nature. Hence it can be interpreted that through the proposed technique, a good quality of encryption is obtained.

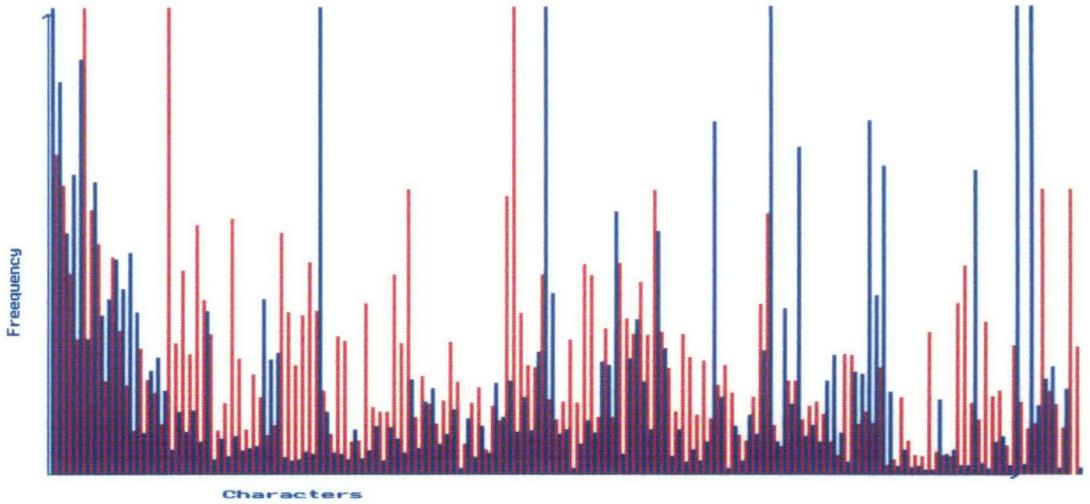
Figure 6.4.2.1 to figure 6.4.2.5 are pictorial representations of a few of these tests.



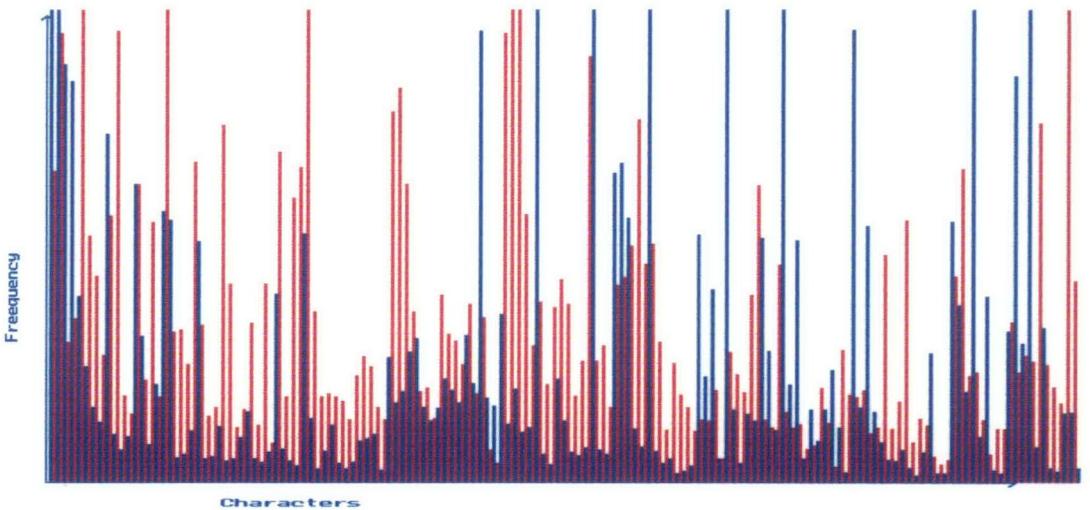
**Figure 6.4.2.1**  
**Frequency Distribution in UNZIP.EXE and Encrypted File for  
RSBP Technique**



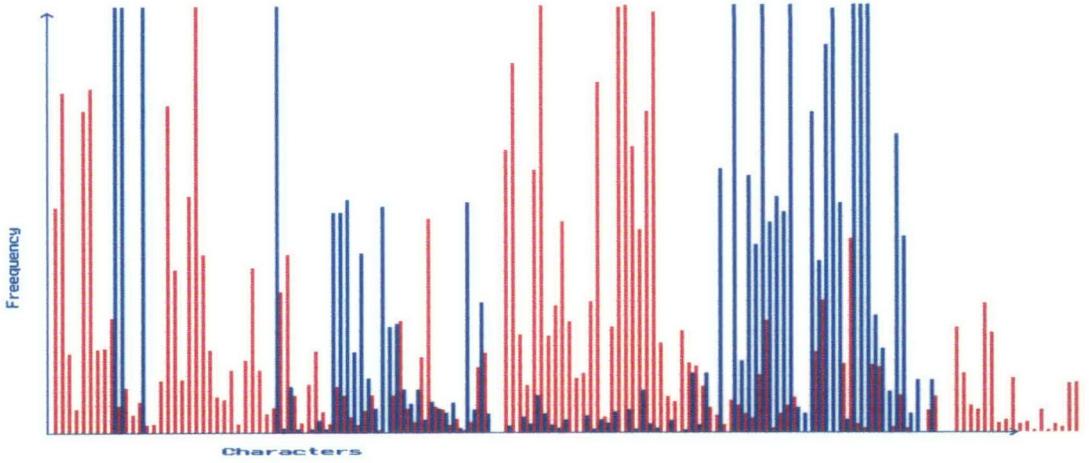
**Figure 6.4.2.2**  
**Frequency Distribution in THELP.COM and Encrypted File for  
RSBP Technique**



**Figure 6.4.2.3**  
**Frequency Distribution in WINSOCK.DLL and Encrypted File for RSBP Technique**



**Figure 6.4.2.4**  
**Frequency Distribution in CCPORT.SYS and Encrypted File for RSBP Technique**



**Figure 6.4.2.5**  
**Frequency Distribution in CHARTCOM.CPP and Encrypted File for RSBP Technique**

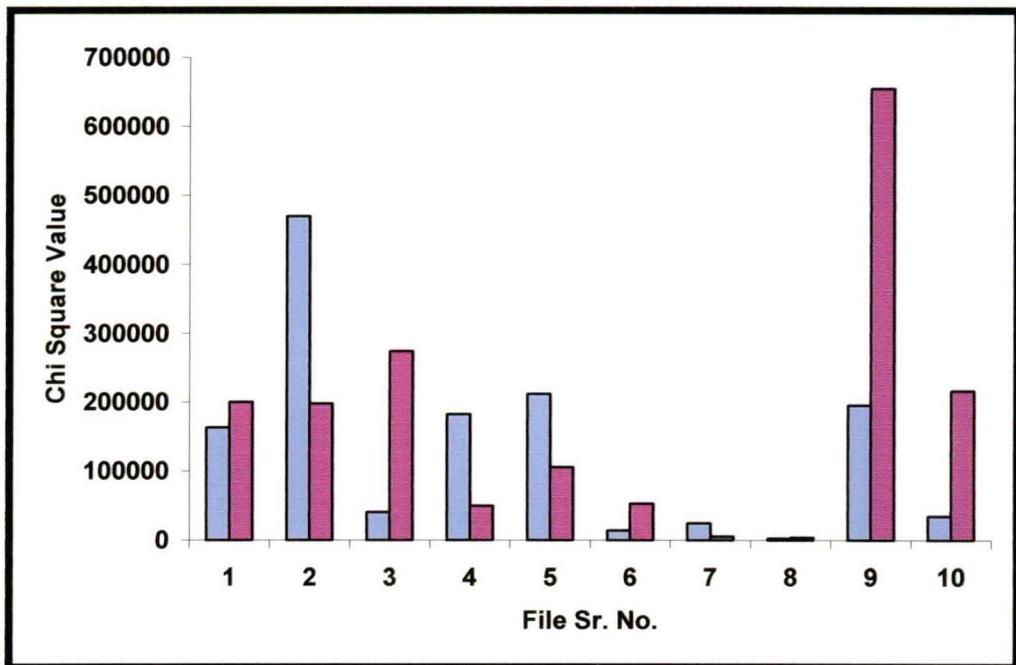
#### 6.4.3 Comparison with RSA Technique

The result of implementing the proposed RSBP technique on the sample .CPP files has been compared with the result of implementing the existing RSA technique on the same set of files. The comparison is made in terms of chi square values. Table 6.4.3.1 enlists this comparative performance. Here the same ten files of type .CPP have been considered, sizes of which are ranging from 1257 bytes to 32150 bytes. The Chi Square value between a source file and the corresponding encrypted file, encrypted using the proposed RPSP technique, is in the range of 2088 to 469578, whereas the same between a source file and the corresponding encrypted file, encrypted using the existing RSA technique, is in the range of 3652 to 655734. The degrees of freedom are in the range of 69 to 90 [41].

**Table 6.4.3.1**  
**Comparative Result between Proposed RSBP and Existing RSA on the basis of**  
**Chi Square Values**

File Name	File Size	Chi Square Value In RPSP	Chi Square Value In RSA	Degree of Freedom
<i>BRICKS.CPP</i>	16723	162768	200221	88
<i>PROJECT.CPP</i>	32150	469578	197728	90
<i>ARITH.CPP</i>	9558	40089	273982	77
<i>START.CPP</i>	14557	182263	49242	88
<i>CHARTCOM.CPP</i>	14080	212197	105384	84
<i>BITIO.CPP</i>	4071	13544	52529	70
<i>MAINC.CPP</i>	4663	24048	4964	83
<i>TTEST.CPP</i>	1257	2088	3652	69
<i>DO.CPP</i>	14481	195541	655734	88
<i>CAL.CPP</i>	9540	34129	216498	77

Graphically, using horizontal bars, it has been exhibited in figure 6.4.3.1. Each blue bar stands for the Chi Square value obtained implementing the RSBP technique, and each brown bar stands for the Chi Square value obtained implementing the RSA technique. There exist three blue bars, the height of each of which is more than the corresponding brown bar.



**Figure 6.4.3.1**  
**Graphical Comparison between Chi Square Values for**  
**RPSP (in Blue) and RSA (in Red)**

## 6.5 Analysis and Conclusion including Comparison with RPSP, TE, RPPO, RPMS

Out of the five techniques proposed so far in the thesis, this RSBP technique, on the basis of the practical implementation done, offers the maximum Chi Square value on the average. Table 6.5.1 shows these results. It is observed from the table that the average Chi Square value of 201990.76 is obtained by implementing the RSBP technique, which is much more than 10701.70, obtained for the RPSP technique, 64188.04, obtained for the TE technique, 85350.94, obtained for the RPPO technique, and 140196.94, obtained for RPMS technique. The average encryption time during implementing the RSBP technique is observed to be 0.74673470 seconds, which is more than times taken implementing the RPMS (0.23659592 seconds) and the RPPO (0.73186806 seconds) techniques, whereas it is lesser than times taken implementing the TE (0.86703290 seconds) and the RPSP (8.75713800 seconds) techniques. The decryption time on the average is observed to be the lowest among these five proposed techniques, which is 0.11040816 seconds [48, 49, 50, 51, 54].

Graphical comparison in this aspect is drawn in chapter 10.

**Table 6.5.1**

**Average Encryption/Decryption Time and Chi Square Value obtained in RPSP, TE, RPPO, RPMS, RSBP**

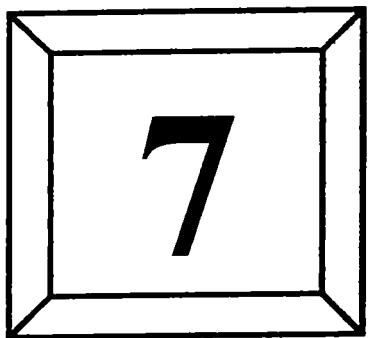
<b>Proposed Technique</b>	<b>Average Encryption Time</b>	<b>Average Decryption Time</b>	<b>Average Chi Square Value</b>	<b>Average Degree of Freedom</b>
<b>RPSP</b>	<b>8.75713800</b>	<b>8.73955200</b>	<b>10701.70</b>	<b>214</b>
<b>TE</b>	<b>0.86703290</b>	<b>0.94175818</b>	<b>64188.04</b>	
<b>RPPO</b>	<b>0.73186806</b>	<b>7.03076904</b>	<b>85350.94</b>	
<b>RPMS</b>	<b>0.23659592</b>	<b>0.15137143</b>	<b>140196.94</b>	
<b>RSBP</b>	<b>0.74673470</b>	<b>0.11040816</b>	<b>201990.76</b>	

The proposed RSBP encryption policy is expected to be highly fruitful in ensuring information security to its best. The marginal alteration in size enhances the security. The requirement of the source file size during the process of decryption may be considered as an overhead, but, in turn, it helps in forming a larger key space as it is proposed to accommodate the source size in the secret key. Figure 8.2.3.1 in chapter 8 shows one proposed 123-bit key format for this RSBP policy strictly following a set of protocols used to form different blocks from the source stream of bits.

Forming blocks of varying lengths in this case makes the process of implementation a bit more difficult, but, if can be implemented faultlessly, ensures much better performance than what is observed in section 6.4.

The process of decryption involves much calculation, especially in the case of varying sizes of different blocks, as for each block size, it is required to calculate the lengths of different rank value sections. This computation overhead, in turn, promises better security.

Besides being highly effective while being implemented independently, if the RSBP encryption technique participates in the cascaded approach, it plays an important role in forming larger key space to enhance the security.



**Encryption Through  
Recursive Substitutions of Bits  
Through Modulo-2 (RSBM)  
Detection of Sub-stream**

<u>Contents</u>	<u>Pages</u>
7.1 Introduction	235
7.2 The Scheme of RSBM	236
7.3 Implementation	244
7.4 Results	252
7.5 Analysis and Conclusion including Comparison with RPSP, TE, RPPO, RPMS, RSBP	264

## 7.1 Introduction

In an attempt to attain the security of a satisfactory level, the RSBP technique, discussed in chapter 6, allowed the storage expansion. The technique proposed in this chapter, the Recursive Substitutions of Bits Through Modulo-2 (RSBM) Detection of Sub-stream, removes the possibility of this overhead by doing a simple alteration from the RSBP technique.

Like the RPMS technique, discussed in chapter 5, here in the RSBM technique also the basic operation to be performed is the modulo-2 operation. In this respect, it is different from the TE technique discussed in chapter 3 and the RPPO technique discussed in chapter 4, because here there is no application of Boolean algebra.

Like all the other proposed techniques, this RSBM is also a bit-level application. So, after converting the source message to be transmitted into a stream of bits, the stream is to be decomposed into a finite number of blocks. Now, in the RSBP technique, to reduce the complexity it was suggested to take unique size for all the blocks. But here in the RSBM technique, since the basic technique is simpler than the RSBP technique and, moreover, since there is no possibility of having the storage expansion, it is highly recommended to choose blocks of varying sizes. As it was shown in chapter 3, while implementing the TE technique, the key length becomes large enough to almost nullify the chance of breaking the cipher by cryptanalysis, if different block lengths are chosen.

Now, for a certain block of bits, the code value and the rank value are to be calculated. For this purpose, the decimal value corresponding to the block is to be computed. Depending on the modulo-2 operation on this decimal value, a 1-bit code value will be set. Preferably, the code value is set to 0 if the modulo-2 operation gives 0 as the result and the same is set to 1 if the result is 1. The rank value is constituted on the basis of the position of the decimal value in the series of natural odd or even numbers, whichever be applicable. It also requires a modulo-2 operation. For a block of size N, the rank value requires exactly (N-1) bits, so that together with the 1-bit code value, exactly N bits are required to represent the block [51, 52, 53]

Like in the RSBP technique, here also it is preferable not to place the code value and the rank value of a block together. Rather, it is suggested to place all the code values followed by all the rank values preferably in the reverse order. Unlike the RSBP

technique, here there is no necessity of placing extra bits between code values and rank values to make the size a multiple of 8, because of no expansion of block size by this approach.

The technique of decryption involves collecting the code value and the corresponding rank value one by one and getting source blocks in this manner.

The technique when is implemented with varying block sizes offers good security, no expansion in size being the added advantage.

Section 7.2 discusses the scheme with pinpointing the exact difference between this and the RSBP technique. The technique is implemented in section 7.3 for the same confidential message that was considered in section 3.3 to be implemented using the TE technique. This section also includes a proposed format of the secret key for the technique. Section 7.4 enlists the results from different perspectives after the technique is implemented for the same groups of files that were considered for the other proposed techniques. An analytical evaluation of the technique is presented in section 7.5.

## 7.2 The Scheme

After generating the stream of bits from the source message to be transmitted, it is to be decomposed into a finite number of blocks. Till this point, the technique is the same as all the other proposed techniques. Section 7.2.1 discusses the policy of encryption and the technique to be followed for decryption is discussed in section 7.2.2.

### 7.2.1 The Encryption Technique

Since it is advised to decompose the source stream into blocks not necessarily of the same length, in this section, first the approach to be followed for calculating the code value and the rank value for a block is discussed, then the technique to integrate all these code values and rank values to form the target stream will be discussed.

Following is the set of steps to be followed for calculating the code value and the rank value of a certain block.

**Step 1:** For the block under consideration, say, of the length of L, calculate the corresponding decimal number. Say, it is D.

**Step 2:** Find out if D is odd or even. If D is odd, the code value for that block is 1 and if not so, it is 0.

**Step 3:** In the series of natural odd or even numbers (whichever be applicable for D) in the range of 0 to  $(2^L - 1)$ , find the position of D. Represent this position in terms of binary values, which will require  $(L-1)$  bits. This is the rank of this block.

These three steps are to be repeated for all blocks, the number of which is, say, N. After finding all the N code values and the N rank values, those are to be integrated using the following set of steps.

**Step 4:** In the target stream of bits, put all the N code values one by one starting from the MSB position. So, in the target stream, the first N bits are code values for N blocks.

**Step 5:** For putting all the rank values in the target stream, we are to start from the  $N^{th}$  bit from the MSB position and then to come back bit-by-bit. Immediately after the  $N^{th}$  bit, put the rank value of the  $N^{th}$  block, followed by the rank value of the  $(N-1)^{th}$  block, and so on. In this way, the rank value of the first block will be placed at the last.

Consider the same 16-bit stream  $S=1010100101010010$  that was considered in section 6.2.1 while discussing the RSBP technique. Here, blocks of different sizes being allowed, we decompose it into three blocks  $S_1=1010$ ,  $S_2=1001$  and  $S_3=01010010$ .

Applying steps 1 to 3 for the block  $S_1=1010$ , we obtain the code value as 0 and the rank value as 101, since corresponding to  $S_1$ , the decimal value is 10, which is  $5^{th}$  (0 being the  $0^{th}$  even number) even number.

Applying steps 1 to 3 for the block  $S_2=1001$ , we obtain the code value as 1 and the rank value as 100, since corresponding to  $S_2$ , the decimal value is 9, which is  $4^{th}$  (1 being the  $0^{th}$  odd number) odd number.

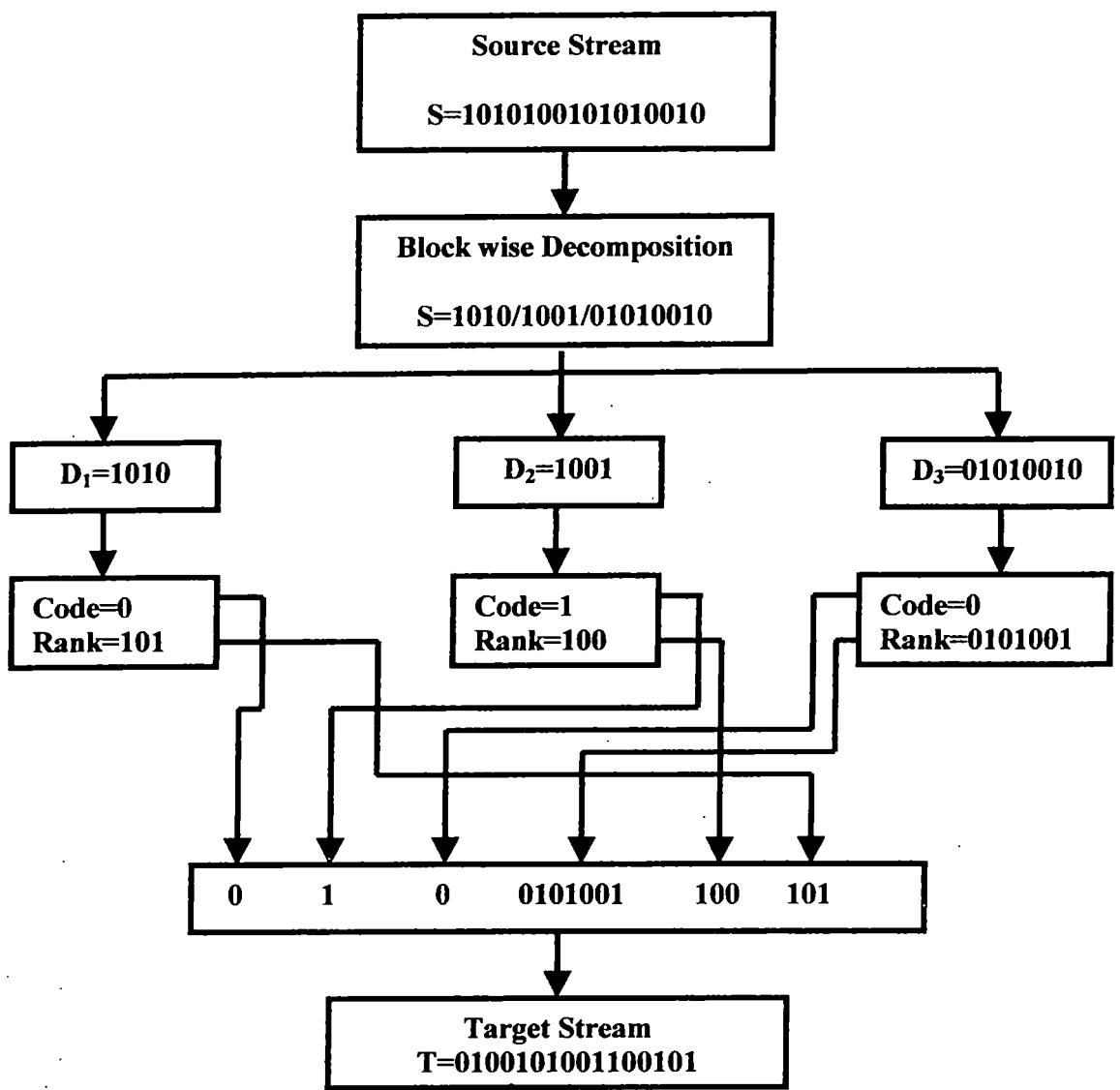
Applying steps 1 to 3 for the block  $S_3=01010010$ , we obtain the code value as 0 and the rank value as 0101001, since corresponding to  $S_3$ , the decimal value is 82, which is  $41^{st}$  (0 being the  $0^{th}$  even number) even number.

Applying step 4, code values of three blocks are placed together to get “010” as the first three bits of the target stream.

We apply step 5 to place all the rank values. After the last code value, we place the rank value “0101001” of the last block ( $S_3$ ), followed by the rank value “100” of the second-to-last block ( $S_2$ ), and finally followed by the rank value “101” of the first block ( $S_1$ ).

In this way, the target stream is obtained as  $T=0100101001100101$ .

Figure 7.2.1.1 is a pictorial representation of this approach.



**Figure 7.2.1.1**  
**Pictorial Representation of Encrypting  $S=1010100101010010$  using RSBM Technique**

It is to mention that for the same source stream  $S=1010100101010010$ , in figure 6.2.1.1 it was shown that the target stream using the RSBP technique was “0011000000000010010110”, which was 24 bits in length.

### 7.2.2 The Decryption Technique

Like in the RSBP technique, here also in the policy of encryption, the strategy of distributing the code values and the rank values of all the source blocks was adopted with the objective of having an unambiguous decryption. As the result of that, the policy of decryption discussed in this section offers an absolute correct outcome.

It is the secret key of the system with the help of which the receiver of the ciphertext will come to know the information regarding different blocks. This information includes two points:

1. The total number of blocks created during the process of encryption
2. The length of each of these blocks

Now, from the point 1, the receiver will come to know the fact that how many bits starting from the MSB are code values.

From point 2, it will be clear in the receiving end that exactly how many bits are to be extracted as the rank value corresponding to a particular code value; because for a block of size  $N$ , exactly  $(N-1)$  bits were used to represent its rank value while encrypting the plaintext.

Following is the set of steps to be followed for the purpose of decryption:

**Step1:** Find the total number of blocks from the secret key, say it is  $B$ .

Therefore, along with the MSB-to-LSB direction, the first  $B$  bits (bit 0 to bit  $(B-1)$ ) represent code values for  $B$  blocks.

**Step 2:** Consider the MSB of the encrypted stream. From the key, find the length of the first block. Say, it is  $N_0$ .

If the MSB is 0,

1. Extract the block consisting of the last  $(N_0 - 1)$  bits of the encrypted stream.
2. Calculate the decimal value corresponding to the block obtained in 1. Say, it is  $D_1$ . Mark this block as being processed.
3. Calculate the  $D_1^{\text{th}}$  even number in the series of natural numbers (with the assumption that

the position of the first even number is 0, not 1).

4. Calculate the  $N_0$ -bit binary value corresponding to the decimal value obtained in 3. It is the first block.
5. Mark the MSB as being processed.

If the MSB is 1,

1. Extract the block consisting of the last  $(N_0 - 1)$  bits of the encrypted stream.
2. Calculate the decimal value corresponding to the block obtained in 1. Say, it is  $D_1$ . Mark this block as being processed.
3. Calculate the  $D_1^{\text{th}}$  odd number in the series of natural numbers (with the assumption that the position of the first odd number is 0, not 1).
4. Calculate the  $N_0$ -bit binary value corresponding to the decimal value obtained in 3. It is the first block.
5. Mark the MSB as being processed.

**Step 3:** Repeat step 4 and step 5 for  $(B-1)$  number of times for the values of  $I$  ranging from 1 to  $(B-1)$  as there are  $(B-1)$  more blocks left to be considered. Set  $I = 1$ .

**Step 4:** Consider the  $I^{\text{th}}$  bit from the MSB position. Let it be denoted by  $T_I$ .

If  $T_I = 1$ ,

1. From the secret key, find out the length of this block. Say, it is  $N_I$ .
2. Consider the first unprocessed block of  $(N_I - 1)$  bits in the LSB-to-MSB direction, convert the binary number represented by this block of bits into the

corresponding decimal, Say, it is M. Mark this block being processed.

3. Find the  $M^{\text{th}}$  odd number in the series of natural numbers (with the assumption that the position of the first prime number is 0, not 1).
4. The  $N_l$ -bit binary number corresponding to the decimal odd number obtained in 3 is the  $I^{\text{th}}$  source block.

If  $T_1 = 0$ ,

1. From the secret key, find out the length of this block. Say, it is  $N_l$ .
2. Consider the first unprocessed block of  $(N_l-1)$  bits in the LSB-to-MSB direction, convert the binary number represented by this block of bits into the corresponding decimal, Say, it is M. Mark this block being processed.
3. Find the  $M^{\text{th}}$  even number in the series of natural numbers (with the assumption that the position of the first prime number is 0, not 1).
4. The  $N_l$ -bit binary number corresponding to the decimal even number obtained in 3 is the  $I^{\text{th}}$  source block.

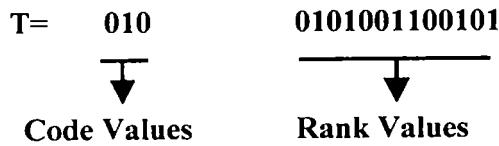
**Step 5:** Let  $I = I + 1$ .

**Step 6:** Concatenate all the blocks obtained so far in the sequence of their generation and this is the source stream.

Consider the same example that was considered in section 7.2.1.

The 16-bit target stream as generated  $T=0100101001100101$ .

Following step 1, using the secret key we find that the total number of blocks is 3, so that the target stream now can be subdivided into two portions as follows:



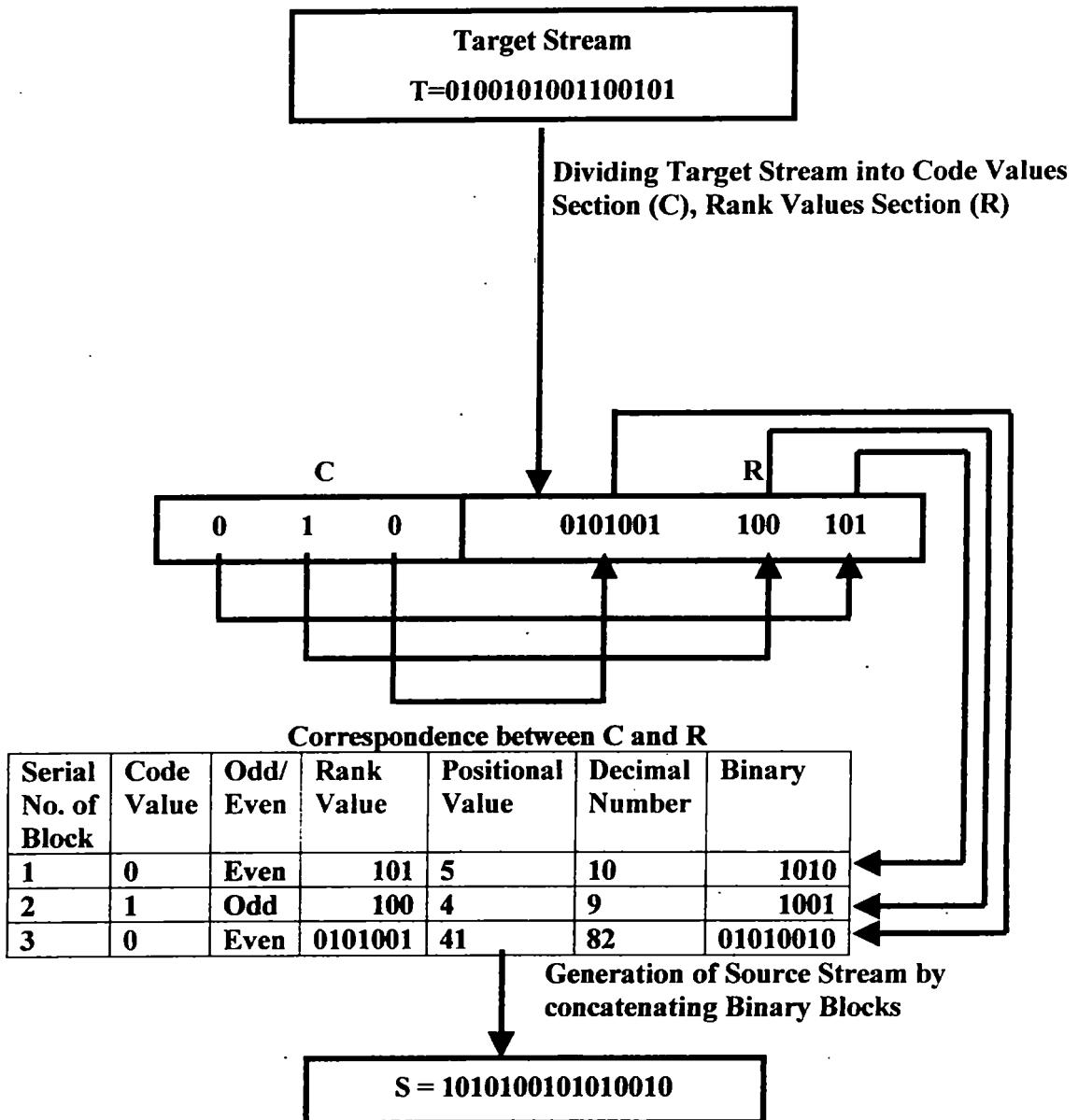
Following step 2, we first consider the MSB, which is 0. Using the secret key we find that the length of the first block is  $N_0=4$ , so that it is understood that  $N_0-1=3$  bits are required to represent the rank of this block. Accordingly, we extract the last three bits "101" from the target stream. Combining the code value and the rank value, we find the fact that the block that is to be generated is corresponding to the 5<sup>th</sup> even number in the series of natural numbers, which is 10, so that the corresponding 4-bit binary value 1010 is the resultant block.

Using step 4 repeatedly for remaining two blocks, we obtain the following information:

1. The second block corresponds to the odd number (the code value being 1) of position 4 (the rank value being 100), which is 9, so that the 4-bit second block is 1001.
2. The third block corresponds to the even number (the code value being 0) of position 41 (the rank value being 0101001), which is 82, so that the 8-bit third block is 01010010.

Concatenating all the blocks generated in the same sequence, the source stream is obtained as  $S=1010100101010010$ .

Figure 7.2.2.1 represents this example diagrammatically.



**Figure 7.2.2.1**  
**Pictorial Representation of Decrypting  $T=0100101001100101$  using RSBM Technique**

### 7.3 Implementation

Whenever in any technique it is suggested to form blocks of varying lengths, the format of the secret key becomes very vital; because in such a case, the frequent accessing of the secret key becomes inevitable during the process of decryption. This section chooses the same plaintext that was chosen in section 3.3 while implementing the

TE technique. Following is the plaintext to be encrypted using the RSBM technique before being transmitted:

**"TERRORIST ATTACK SUSPECTED HOTEL SNOWVIEW DEC 06 ALERT"**

Here also we consider some prefixed rules or agreement, which were also considered while encrypting using the TE technique. Following are the rules

- The maximum number of characters allowed in the message is 60.
- The maximum length of a block is 32 bits.
- The maximum number of blocks is 20.

Now, in contrast to the proposed format of secret key shown in figure 8.2.2.1 for the TE technique, here the structure will be different and it does not require 180 bits to ensure the correct decryption. Since to represent the length of a block, 6 bits are required, and at most there can be 20 blocks. a total of 120 bits are enough to complete the key structure. Figure 7.3.1 shows this structure.

Position of Bits

0

29

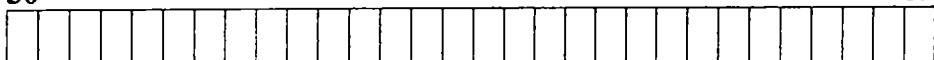


Size of Block 1	Size of Block 2	Size of Block 3	Size of Block 4	Size of Block 5
-----------------	-----------------	-----------------	-----------------	-----------------

Position of Bits

30

59



Size of Block 6	Size of Block 7	Size of Block 8	Size of Block 9	Size of Block 10
-----------------	-----------------	-----------------	-----------------	------------------

Position of Bits

60

89

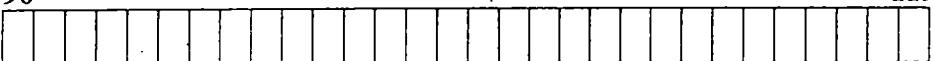


Size of Block 11	Size of Block 12	Size of Block 13	Size of Block 14	Size of Block 15
------------------	------------------	------------------	------------------	------------------

Position of Bits

90

119



Size of Block 16	Size of Block 17	Size of Block 18	Size of Block 19	Size of Block 20
------------------	------------------	------------------	------------------	------------------

**Figure 7.3.1**

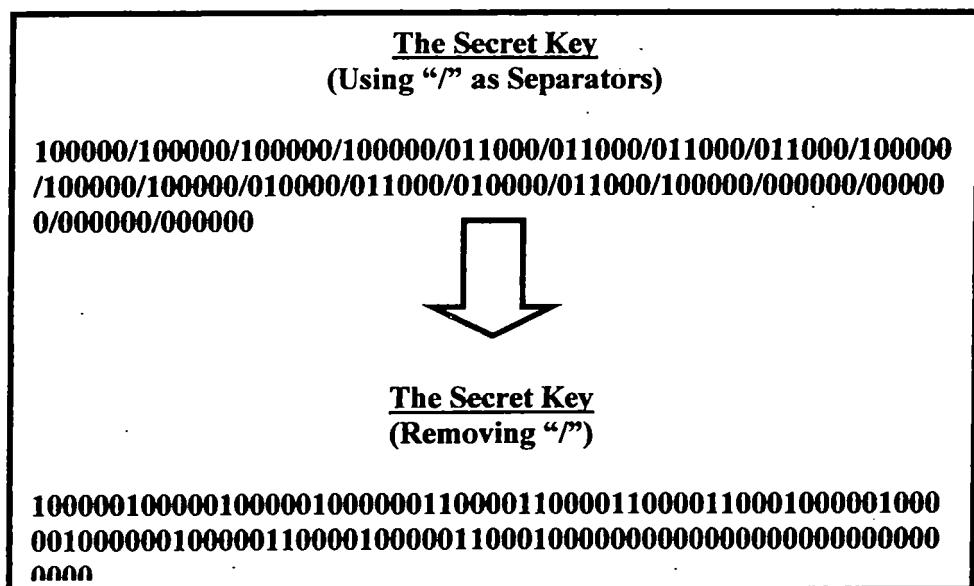
**Format of 120-Bit Secret Key**

Table 7.3.1 enlists the considerations to be followed during the process of encryption.

**Table 7.3.1**  
**Different Considerations for Encryption using RSBM Technique**

Total Number of Blocks: 16							
Size of Block 1	32	Size of Block 5	24	Size of Block 9	32	Size of Block 13	24
Size of Block 2	32	Size of Block 6	24	Size of Block 10	32	Size of Block 14	16
Size of Block 3	32	Size of Block 7	24	Size of Block 11	32	Size of Block 15	24
Size of Block 4	32	Size of Block 8	24	Size of Block 12	16	Size of Block 16	32

Following the format of the 120-bit secret key shown in figure 7.3.1, the key will be as shown in figure 7.3.2.



**Figure 7.3.2**  
**120-bit Secret Key in RSBM Technique**

By converting characters into bytes using table 3.3.2 in chapter 3, we obtain the following source stream of bits of the length of 432 bits:

010101000100010101010010010010011110101001001001001010100110101001  
 001000000100000101010100010101000100000101000011010010110010000001010011  
 01010101010011010100000100010101000011010100010001010100010000100000  
 010010000010111101010100010001010100110000100000010100110100111001001111  
 0101011101011001001000101000101011100100000010001000100010101000011  
 001000000011000000110110001000000100000101001100010001000101010010010100

Using the secret key, we construct table 7.3.2, where all the blocks are shown.

**Table 7.3.2**  
**Different Blocks Considered for Encryption in RSBM Technique**

<b>Block 1</b>	01010100010001010101001001010010	<b>Block 9</b>	00101111010101000100010101001100
<b>Block 2</b>	010011110101001001001001010011	<b>Block 10</b>	00100000010100110100111001001111
<b>Block 3</b>	01010011001000000100000101010100	<b>Block 11</b>	010101110101011001001001001000101
<b>Block 4</b>	01010100010000010100001101001011	<b>Block 12</b>	0101011100100000
<b>Block 5</b>	001000000101001101010101	<b>Block 13</b>	010001000100010101000011
<b>Block 6</b>	010100110101000001000101	<b>Block 14</b>	0010000000110000
<b>Block 7</b>	010000110101010001000101	<b>Block 15</b>	001101100010000001000001
<b>Block 8</b>	010001000010000001001000	<b>Block 16</b>	01001100010001010101001001010100

Different steps of the process of encryption are presented in tabular form in table 7.3.3 and table 7.3.4. Table 7.3.3 finds through modulo-2 operations whether different blocks represent even or odd numbers, and whatever is the case, it also finds positions in the series of natural numbers.

**Table 7.3.3**  
**Different Steps during Encryption (Part 1) in RSBM Technique**

Block Sr. No.	Block	Decimal Value	Even/ Odd	Position
1	01010100010001010101001001010010	1413829202	Even	706914601
2	0100111010100100100100101010011	1330792787	Odd	665396393
3	01010011001000000100000101010100	1394622804	Even	697311402
4	01010100010000010100001101001011	1413563211	Odd	706781605
5	001000000101001101010101	2118485	Odd	1059242
6	010100110101000001000101	5460037	Odd	2730018
7	010000110101010001000101	4412485	Odd	2206242
8	010001000010000001001000	4464712	Even	2232356
9	0010111010101000100010101001100	794051916	Even	397025958
10	00100000010100110100111001001111	542330447	Odd	271165223
11	01010111010101100100100101000101	1465272645	Odd	732636322
12	0101011100100000	22304	Even	11152
13	010001000100010101000011	4474179	Odd	2237089
14	0010000000110000	8240	Even	4120
15	001101100010000001000001	3547201	Odd	1773600
16	01001100010001010101001001010100	1279611476	Even	639805738

From information in the last two columns in table 7.3.3, code values and rank values of all the blocks are tabulated in table 7.3.4.

**Table 7.3.4**  
**Different Steps during Encryption (Part 2) in RSBM Technique**

Block Sr. No.	Block Length (L)	1-Bit Code Value	(L-1)-Bit Rank Value
1	32	0	0101010001000101010100100101001
2	32	1	0100111101010010010010010101001
3	32	0	010100110010000010000010101010
4	32	1	0101010001000001010000110100101
5	24	1	00100000010100110101010
6	24	1	01010011010100000100010
7	24	1	01000011010101000100010
8	24	0	01000100001000000100100
9	32	0	0010111101010100010001010100110
10	32	1	0010000001010011010011100100111
11	32	1	0101011101010110010010010100010
12	16	0	010101110010000
13	24	1	01000100010001010100001
14	16	0	001000000011000
15	24	1	00110110001000000100000
16	32	0	0100110001000101010100100100101010

By combining all code values from table 7.3.4 in the sequence of blocks, we construct the code value section (C) as follows:

**0101111001101010**

To construct the rank value section (R), we combine all rank values from table 3.3.5 in the reverse sequence, which means that the last rank value is to be placed first, followed by the previous rank value, and so on. Following this approach, we construct R as follows:

0100110001000101010100100101000110110001000000100000001000  
 000011000010001000100010101000010101011100100000101011101010  
 110010010010100010001000000101001101001110010011100101111010

**1010001000101001100100010000100000010010001000011010101000  
10001001010011010100001000100000010100110101010010100  
0100000101000011010010101001100100000010000010101010010011  
1101010010010010010101001010100100010001010101001001001001**

Finally, we construct the encrypted stream T by the following operation:

$$T = C + R$$

“+” denoting the concatenation operation. Following is the encrypted stream T:

**01011110011010010011000100010101001001010100011011000100  
00001000000010000000110000100010001000101010000101010110010  
0000101011101010110010010010001000100000010100110100111001  
0011100101111010101000100010101001100100010000001001000  
1000011010101000100010010011010100000100010001000000101001  
101010100101010001000001010000110100101010100110010000001000  
0010101010010011101010010010010010101001010100010001000101010  
100100101001**

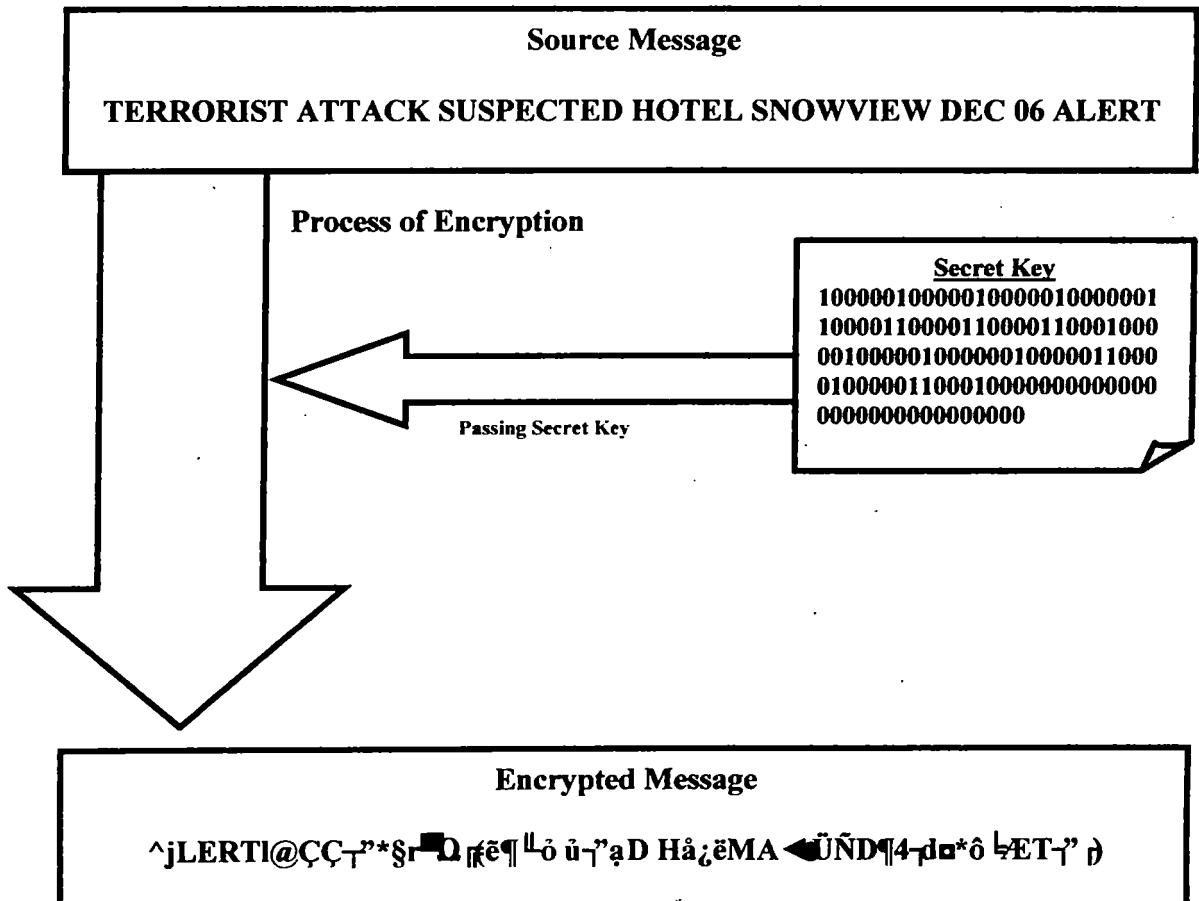
We re-write T in byte wise format as following to convert into stream of characters:

**01011110/01101010/01001100/01000101/01010010/01010100/01101100/  
01000000/10000000/10000000/11000010/00100010/00101010/00010101/  
01110010/00001010/11101010/11001001/00101000/10001000/00010100/  
11010011/10010011/10010111/10101010/00100010/10100110/01000100/  
00100000/01001000/10000110/10101000/10001001/01001101/01000001/  
00010001/00000010/10011010/10100101/01000100/00010100/00110100/  
10101010/01100100/00001000/00101010/10010011/11010100/10010010/  
01010100/10101010/00100010/10101001/00101001**

From this re-written T, we obtain the encrypted text or the ciphertext looking like the following:

**^jLERTI@ÇÇ¬\*§r¬Q¬k¬¶L¬ô¬¬äD Hå¬ëMA ¬ÜND¶4¬d¬\*ô¬ÅET¬”¬**

Since the same plaintext as in chapter 3 is taken here for the purpose of implementation, pictorially, using the same format of figure 3.3.19 in chapter 3, we present this implementation in figure 7.3.3.



**Figure 7.3.3**  
**Generating Encrypted Message from Source using 120-Bit Secret Key in RSBM Technique**

#### 7.4 Results

In this section different sample files have been encrypted using the RSBM technique for blocks of unique length of only 8 bits. One advantage of using small blocks is the fast implementation, but as it is being discussed throughout the thesis, that for ensuring the security of the highest level blocks of lengths at least of 64 bits are to be considered [44,51].

Section 7.4.1 shows results of the encryption/decryption time and the chi square value, section 7.4.2 depicts pictorial result of the frequency distribution tests, section 7.4.3 presents results of the comparison with the RSA system.

#### **7.4.1 Result of Encryption/Decryption Time and Chi Square Value**

For the experimental purpose, the RSBM technique has been implemented for the unique block size of 8 bits. Section 7.4.1.1 to section 7.4.1.5 presents results for files of different categories, namely, .EXE, .COM, .DLL, .SYS, and .CPP. Each section consists of tables with information on the source file name and size, the encrypted file name, the encryption time, the decryption time, and the chi square value, with the degree of freedom. All the results obtained suggest the fact that there exists a tendency of increasing the encryption/decryption time linearly with the source size [55, 56].

##### **7.4.1.1 Result for *EXE* Files**

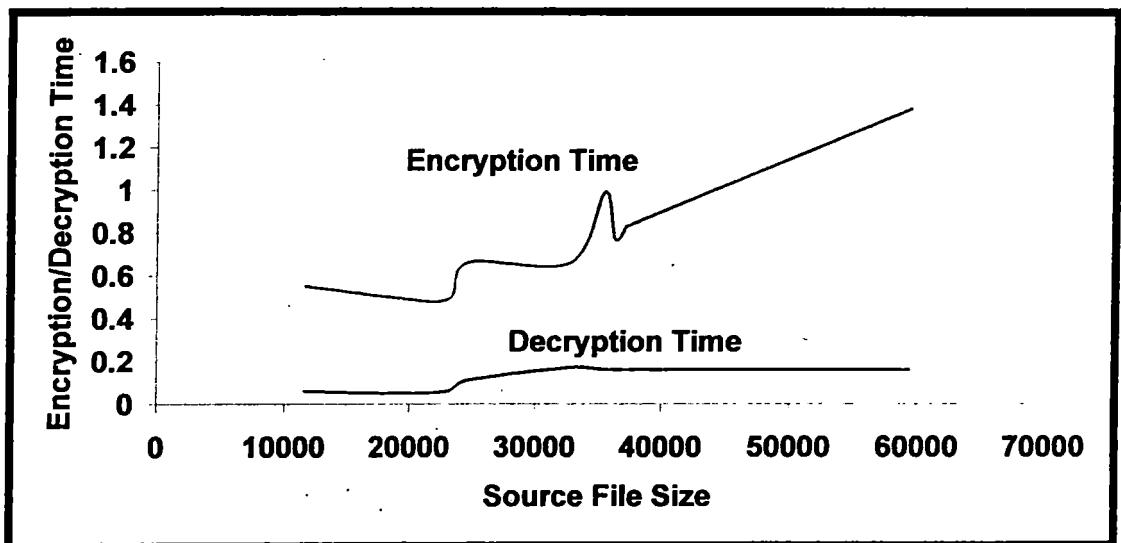
Table 7.4.1.1.1 presents the result for .EXE files. Ten files have been considered. The size of a file is in the range of 11611 bytes to 59398 bytes. The encryption time is in the range of 0.4900 seconds to 1.3800 seconds, whereas the decryption time lies in the range of 0.0500 seconds to 0.1700 seconds. The Chi Square value is between 15058 and 117233 with the degree of freedom ranging between 248 and 255.

**Table 7.4.1.1.1  
Result for EXE Files for RSBM Technique**

Source File	Encrypted File	Source File Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom
<i>TLIB.EXE</i>	<i>A1.EXE</i>	37220	0.8299	0.1600	113159	255
<i>MAKER.EXE</i>	<i>A2.EXE</i>	59398	1.3800	0.1600	117233	255
<i>UNZIP.EXE</i>	<i>A3.EXE</i>	23044	0.4900	0.0600	15058	255
<i>RPPO.EXE</i>	<i>A4.EXE</i>	35425	0.9900	0.1600	40176	255
<i>PRIME.EXE</i>	<i>A5.EXE</i>	37152	0.8299	0.1600	44662	255
<i>TCDEF.EXE</i>	<i>A6.EXE</i>	11611	0.5500	0.0600	21847	254
<i>TRIANGLE.EXE</i>	<i>A7.EXE</i>	36242	0.7699	0.1600	41091	255
<i>PING.EXE</i>	<i>A8.EXE</i>	24576	0.6600	0.1100	35076	248
<i>NETSTAT.EXE</i>	<i>A9.EXE</i>	32768	0.6600	0.1700	63082	255
<i>CLIPBRD.EXE</i>	<i>A10.EXE</i>	18432	0.5000	0.0500	43157	255

The graphical relationship between the source file size and the encryption/decryption time for .EXE files is shown in figure 7.4.1.1.1. Here the required

encryption/decryption times are too marginal to draw any conclusion regarding their nature.



**Figure 7.4.1.1.1**  
**Graphical Relationship between Source Size and Encryption/Decryption Time for EXE Files in RSBM Technique**

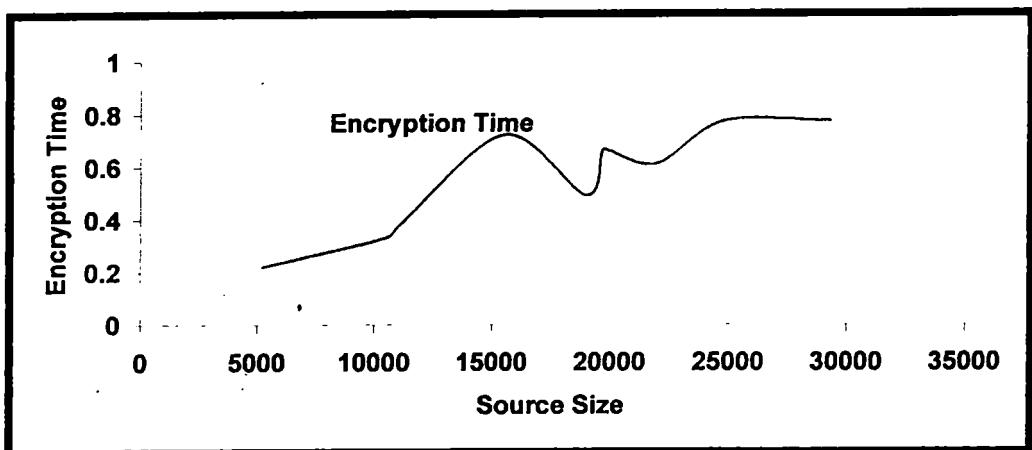
#### 7.4.1.2 Result for COM Files

Table 7.4.1.2.1 presents the result for .COM files. Ten files have been considered. The size of a file is in the range of 5239 bytes to 29271 bytes. The encryption time is in the range of 0.2200 seconds to 0.7700 seconds, whereas the decryption time lies in the range of 0.0000 seconds to 0.1100 seconds. The Chi Square value is between 5080 and 39050 with the degree of freedom ranging between 230 and 255.

**Table 7.4.1.2.1**  
**Result for COM Files for RSBM Technique**

Source File	Encrypted File	Source File Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom
<i>EMSTEST.COM</i>	<i>A1.COM</i>	19664	0.6600	0.0600	27754	255
<i>THELP.COM</i>	<i>A2.COM</i>	11072	0.3800	0.0600	19795	250
<i>WIN.COM</i>	<i>A3.COM</i>	24791	0.7700	0.0600	36397	252
<i>KEYB.COM</i>	<i>A4.COM</i>	19927	0.6600	0.0600	39050	255
<i>CHOICE.COM</i>	<i>A5.COM</i>	5239	0.2200	0.0000	5989	232
<i>DISKCOPY.COM</i>	<i>A6.COM</i>	21975	0.6100	0.1100	26877	254
<i>DOSKEY.COM</i>	<i>A7.COM</i>	15495	0.7200	0.1100	25433	253
<i>MODE.COM</i>	<i>A8.COM</i>	29271	0.7700	0.1100	34737	255
<i>MORE.COM</i>	<i>A9.COM</i>	10471	0.3300	0.0500	5080	230
<i>SYS.COM</i>	<i>A10.COM</i>	18967	0.4900	0.1100	26524	254

The graphical relationship between the source size and the encryption time for .COM files is shown in figure 7.4.1.2.1. Although there exists a tendency of having almost linear increment of the encryption time with the source size with a few exceptions, but the required times are too marginal to draw any conclusion.



**Figure 7.4.1.2.1**  
**Graphical Relationship between Source Size and Encryption/Decryption Time for COM Files in RSBM Technique**

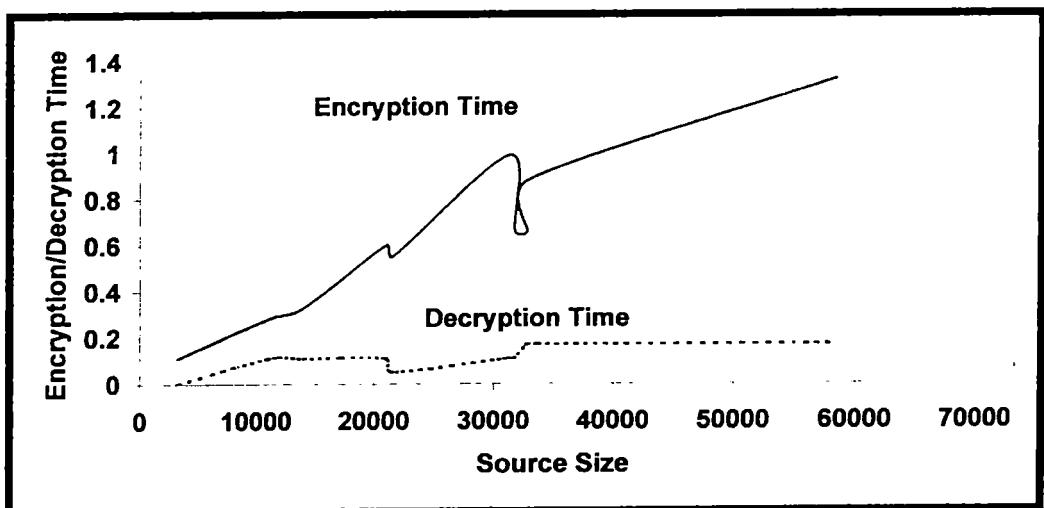
#### 7.4.1.3 Result for *DLL* Files

Table 7.4.1.3.1 presents the result for .DLL files. Ten files have been considered. The size of a file is in the range of 3216 bytes to 58368 bytes. The encryption time is in the range of 0.1100 seconds to 0.9900 seconds, whereas the decryption time lies in the range of 0.0000 seconds to 0.1700 seconds. The Chi Square value is between 5604 and 108189 with the degree of freedom ranging between 217 and 255.

**Table 7.4.1.3.1  
Result for DLL Files for RSBM Technique**

<b>Source File</b>	<b>Encrypted File</b>	<b>Source File Size</b>	<b>Encryption Time</b>	<b>Decryption Time</b>	<b>Chi Square Value</b>	<b>Degree of Freedom</b>
<i>SNMPAPI.DLL</i>	<i>A1.DLL</i>	32768	0.6600	0.1700	52620	253
<i>KPSHARP.DLL</i>	<i>A2.DLL</i>	31744	0.6600	0.1100	90282	254
<i>WINSOCK.DLL</i>	<i>A3.DLL</i>	21504	0.5500	0.0500	73168	252
<i>SPWHPT.DLL</i>	<i>A4.DLL</i>	32792	0.8800	0.1700	80508	255
<i>HIDCI.DLL</i>	<i>A5.DLL</i>	3216	0.1100	0.0000	5604	217
<i>PFPICK.DLL</i>	<i>A6.DLL</i>	58368	1.3200	0.1700	85428	255
<i>NDDEAPI.DLL</i>	<i>A7.DLL</i>	14032	0.3300	0.1100	41512	249
<i>NDDENB.DLL</i>	<i>A8.DLL</i>	10976	0.2800	0.1100	40654	251
<i>ICCCODES.DLL</i>	<i>A9.DLL</i>	20992	0.6000	0.1100	68138	252
<i>KPSCALE.DLL</i>	<i>A10.DLL</i>	31232	0.9900	0.1100	108189	255

Figure 7.4.1.3.1 establishes the graphical relationship between the source size and the encryption/decryption time for .DLL files. Here also encryption/decryption times are too marginal to draw any fixed conclusion about their nature.



**Figure 7.4.1.3.1**  
**Graphical Relationship between Source Size and Encryption/Decryption Time for DLL Files in RSBM Technique**

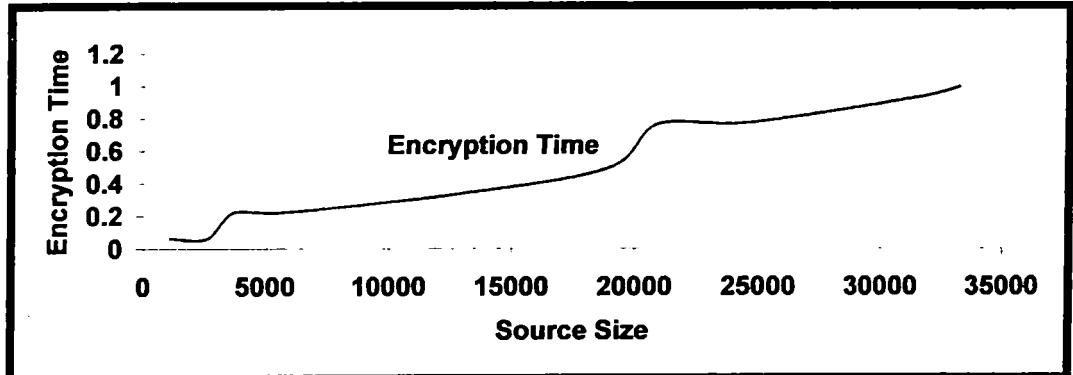
#### 7.4.1.4 Result for SYS Files

Table 7.4.1.4.1 presents the result for .SYS files. Ten files have been considered. The size of a file is in the range of 1105 bytes to 33191 bytes. The encryption time is in the range of 0.0600 seconds to 0.9900 seconds, whereas the decryption time lies in the range of 0.0000 seconds to 0.1700 seconds. The Chi Square value is between 2977 and 67190 with the degree of freedom ranging between 165 and 255.

**Table 7.4.1.4.1**  
**Result for SYS Files for RSBM Technique**

Source File	Encrypted File	Source File Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom
HIMEM.SYS	A1.SYS	33191	0.9900	0.1100	29029	255
RAMDRIVE.SYS	A2.SYS	12663	0.3300	0.0500	11376	241
USBD.SYS	A3.SYS	18912	0.4900	0.1100	45673	255
CMD610X.SYS	A4.SYS	24626	0.7700	0.1100	45907	255
CMD640X2.SYS	A5.SYS	20901	0.7600	0.1100	62374	255
REDBOOK.SYS	A6.SYS	5664	0.2200	0.1100	11661	230
IFSHLP.SYS	A7.SYS	3708	0.2200	0.0000	5710	237
ASPI2HLP.SYS	A8.SYS	1105	0.0600	0.0500	1088	165
DBLBUFF.SYS	A9.SYS	2614	0.0600	0.0500	2977	215
CCPORT.SYS	A10.SYS	31680	0.9300	0.1700	67190	255

Figure 7.4.1.4.1 exhibits the relationship between the source size and the encryption time for .SYS files. Here the relationship is observed to be almost linear in nature.



**Figure 7.4.1.4.1**  
**Graphical Relationship between Source Size and Encryption/Decryption Time for SYS Files in RSBM Technique**

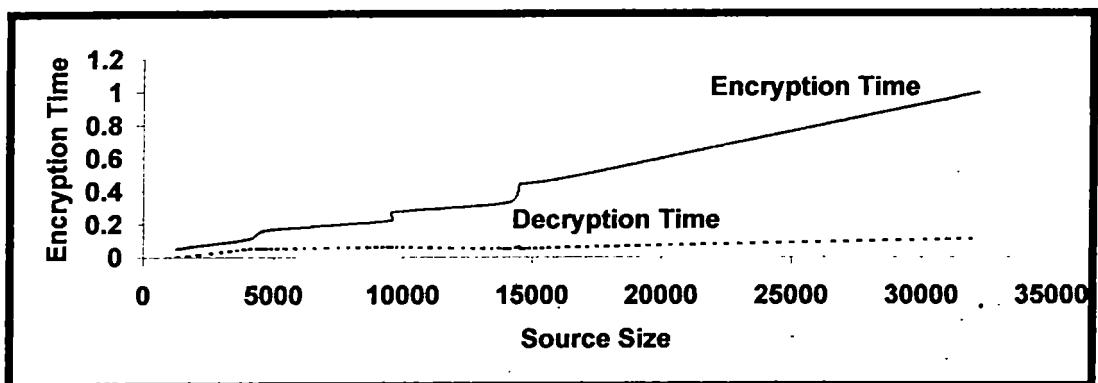
#### 7.4.1.5 Result for CPP Files

Table 7.4.1.4.1 presents the result for .CPP files. Ten files have been considered. The size of a file is in the range of 1257 bytes to 32150 bytes. The encryption time is in the range of 0.0500 seconds to 0.9900 seconds, whereas the decryption time lies in the range of 0.0000 seconds to 0.1100 seconds. The Chi Square value is between 965 and 131367 with the degree of freedom ranging between 69 and 90.

**Table 7.4.1.5.1**  
**Result for CPP Files for RSBM Technique**

Source File	Encrypted File	Source File Size	Encryption Time	Decryption Time	Chi Square Value	Degree of Freedom
<i>BRICKS.CPP</i>	<i>A1.CPP</i>	16723	0.4900	0.0600	38464	88
<i>PROJECT.CPP</i>	<i>A2.CPP</i>	32150	0.9900	0.1100	131367	90
<i>ARITH.CPP</i>	<i>A3.CPP</i>	9558	0.2700	0.0600	15345	77
<i>START.CPP</i>	<i>A4.CPP</i>	14557	0.4400	0.0500	38457	88
<i>CHARTCOM.CPP</i>	<i>A5.CPP</i>	14080	0.3300	0.0500	28785	84
<i>BITIO.CPP</i>	<i>A6.CPP</i>	4071	0.1100	0.0500	6013	70
<i>MAINC.CPP</i>	<i>A7.CPP</i>	4663	0.1600	0.0500	6245	83
<i>TTEST.CPP</i>	<i>A8.CPP</i>	1257	0.0500	0.0000	965	69
<i>DO.CPP</i>	<i>A9.CPP</i>	14481	0.4400	0.0600	36742	88
<i>CAL.CPP</i>	<i>A10.CPP</i>	9540	0.2200	0.0600	15436	77

Figure 7.4.1.5.1 is a graphical representation of the almost linear relationship of the encryption time with the source size for .CPP files. The figure also exhibits the nature of the decryption time, which is observed not to vary too much with the source size.

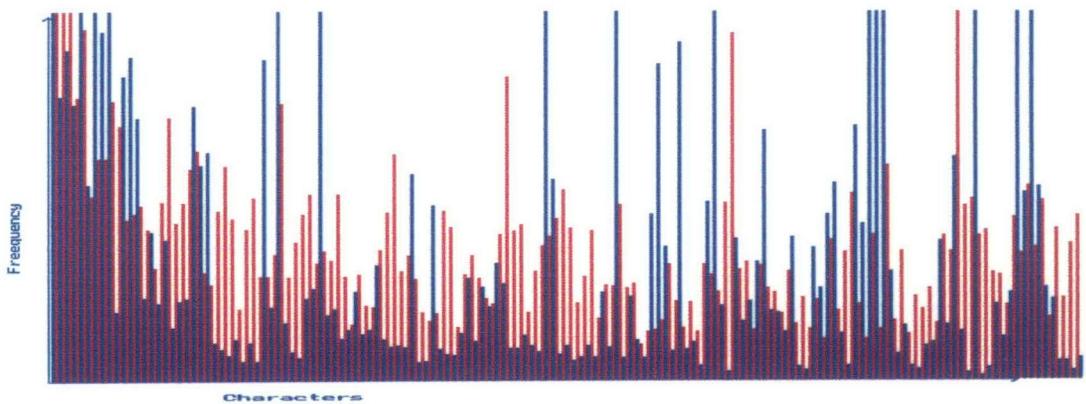


**Figure 7.4.1.5.1**  
**Graphical Relationship between Source Size and Encryption/Decryption Time for CPP Files**

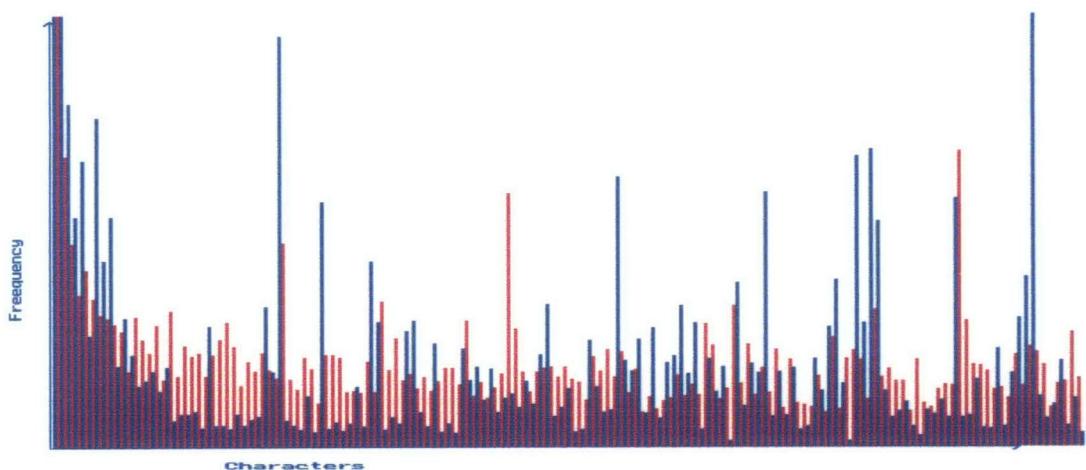
#### 7.4.2 Result of Frequency Distribution Tests

The frequency distribution test between each of the 50 sample pairs of the source file and the encrypted file has been performed. It is seen for all cases that the characters in the encrypted files are well distributed, which indicates that the technique proposed

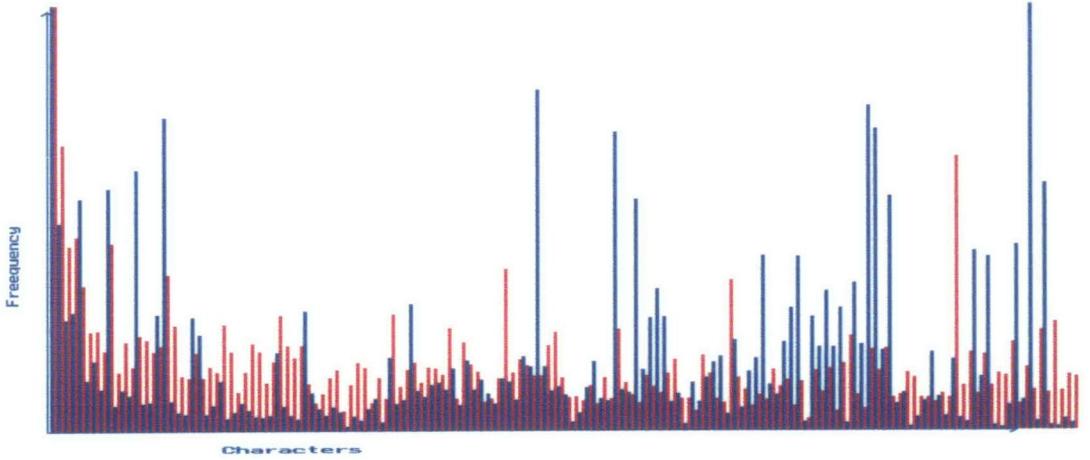
here is quite compatible with existing techniques. The red bars represent frequencies of characters in the encrypted file and those in blue color represent frequencies of characters in the source file. The frequencies of characters in encrypted files are evenly distributed. Therefore the source and the corresponding encrypted file are heterogeneous in nature. Hence it can be interpreted that through the proposed technique, a good quality of encryption is obtained. Here in figure 7.4.2.1 to figure 7.4.2.5, some results have been shown.



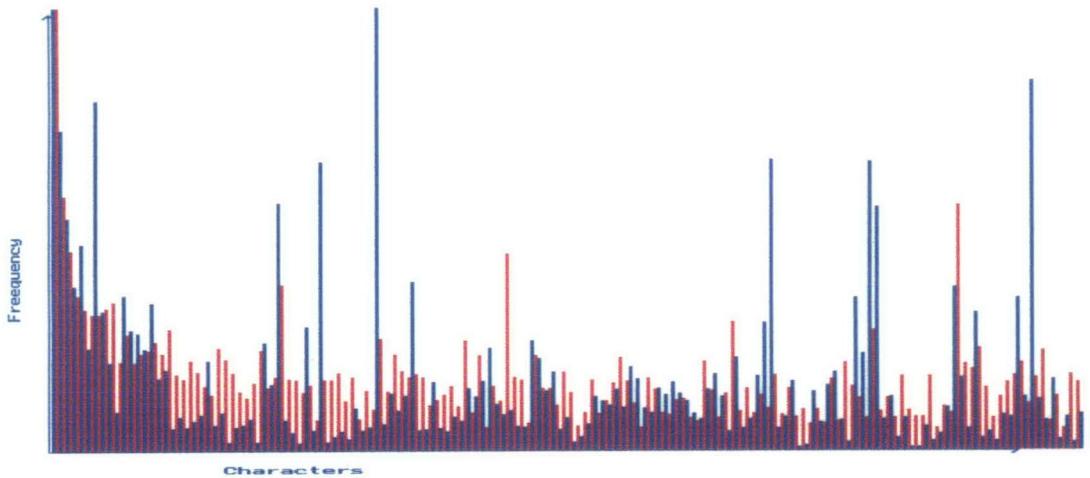
**Figure 7.4.2.1**  
**Frequency Distribution between**  
**TLIB.EXE and Encrypted FOX1.EXE in RSBM Technique**



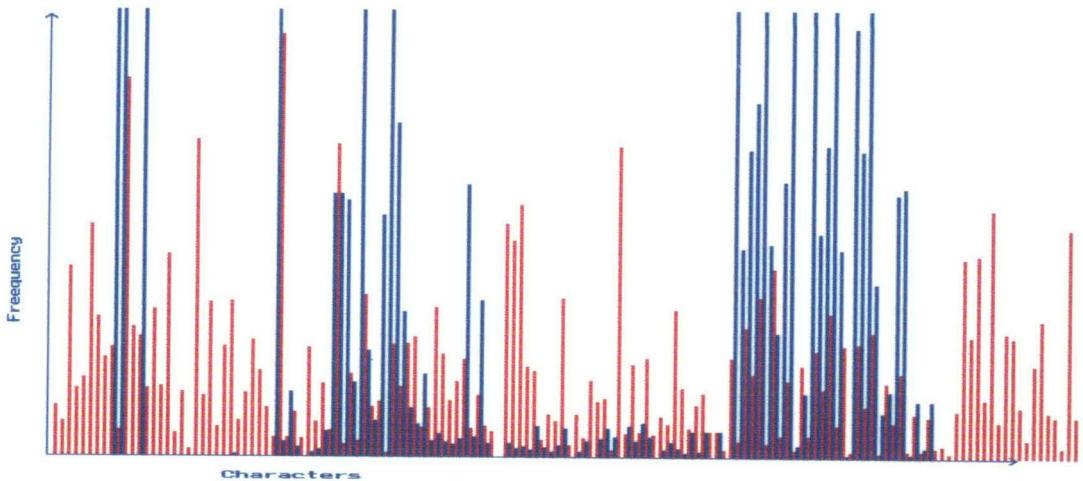
**Figure 7.4.2.2**  
**Frequency Distribution between**  
**EMSTEST.COM and Encrypted FOX1.COM in RSBM Technique**



**Figure 7.4.2.3**  
**Frequency Distribution between**  
**SNMPAPI.DLL and Encrypted FOX1.DLL in RSBM Technique**



**Figure 7.4.2.4**  
**Frequency Distribution between**  
**HIMEM.SYS and Encrypted FOX1.SYS in RSBM Technique**



**Figure 7.4.2.5**  
**Frequency Distribution between**  
**BRICKS.CPP and Encrypted FOX1.CPP in RSBM Technique**

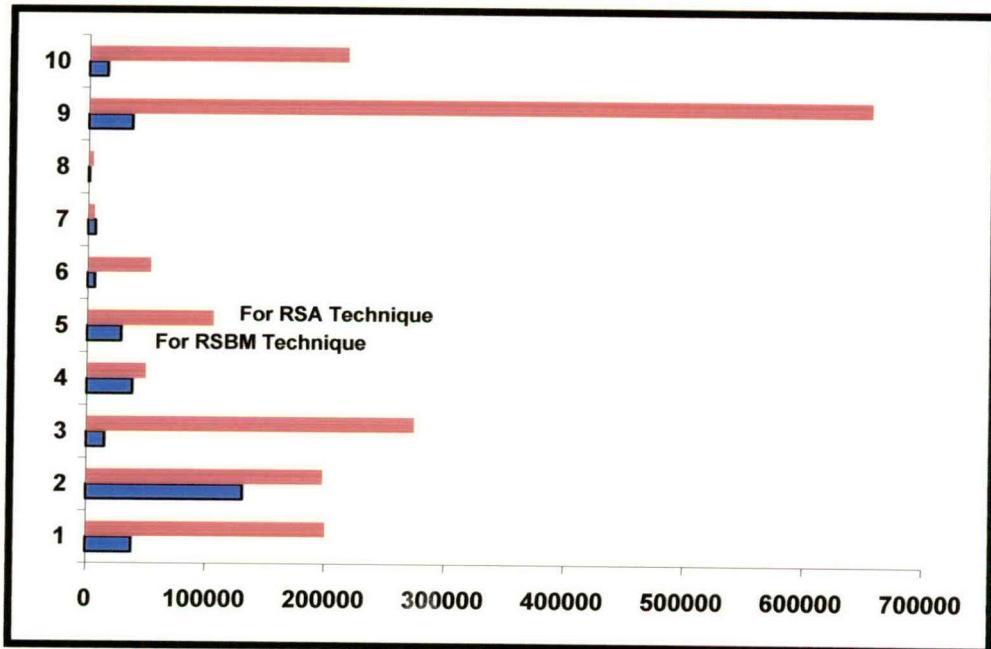
### 7.4.3 Comparison with RSA Technique

An analytical conclusion is attempted to establish on the efficiency of the proposed RSBM technique in terms of the result obtained for .CPP files. Table 7.4.3.1 enlists chi square values between .CPP sample files and corresponding encrypted files using both the proposed RSBM and the existing RSA techniques. Here the source file ranges from 1257 bytes to 32150 bytes in size. The chi Square value between a source file and the corresponding encrypted file, encrypted using the proposed RSBM technique, lies in the range of 965 to 131367, whereas the same for the RSA technique is in the range from 3652 to 655734. The degree of freedom ranges from 69 to 90.

**Table 7.4.3.1**  
**Comparative Result between Proposed RSBM and Existing RSA on the basis of**  
**Chi Square Values**

Source File (1)	Source Size	Encrypted File using RSBM Technique (2)	Encrypted File using RSA Technique (3)	Chi Square Value between (1) and (2)	Chi Square Value between (1) and (3)	Degree of Freedom
<i>BRICKS.CPP</i>	16723	<i>FOX1.CPP</i>	<i>CPP1.CPP</i>	38464	200221	88
<i>PROJECT.CPP</i>	32150	<i>FOX2.CPP</i>	<i>CPP2.CPP</i>	131367	197728	90
<i>ARITH.CPP</i>	9558	<i>FOX3.CPP</i>	<i>CPP3.CPP</i>	15345	273982	77
<i>START.CPP</i>	14557	<i>FOX4.CPP</i>	<i>CPP4.CPP</i>	38457	49242	88
<i>CHARTCOM.CPP</i>	14080	<i>FOX5.CPP</i>	<i>CPP5.CPP</i>	28785	105384	84
<i>BITIO.CPP</i>	4071	<i>FOX6.CPP</i>	<i>CPP6.CPP</i>	6013	52529	70
<i>MAINC.CPP</i>	4663	<i>FOX7.CPP</i>	<i>CPP7.CPP</i>	6245	4964	83
<i>TTEST.CPP</i>	1257	<i>FOX8.CPP</i>	<i>CPP8.CPP</i>	965	3652	69
<i>DO.CPP</i>	14481	<i>FOX9.CPP</i>	<i>CPP9.CPP</i>	36742	655734	88
<i>CAL.CPP</i>	9540	<i>FOX10.CPP</i>	<i>CPP10.CPP</i>	15436	216498	77

Figure 7.4.3.1 graphically shows this comparison. Here brown horizontal bars stand for results applying the RSA technique, whereas the blue horizontal bars stand for results applying the proposed RSBM technique. As is shown in the figure, in most of the cases brown bars are longer than the corresponding blue bars. Therefore in terms of Chi Square value results obtained from applying the RSBM technique is not satisfactory enough.



**Figure 7.4.3.1**  
**Comparison between Proposed RSBM Technique and Existing RSA Technique in terms of Chi Square Values**

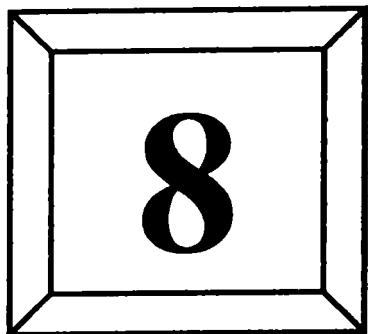
## 7.5 Analysis and Conclusion including Comparison with RPSP, TE, RPPO, RPMS, RSBP

Out of all the six proposed techniques, this RSBM encryption technique provides the least satisfactory result in terms of the Chi Square value. Table 7.5.1 enlists the comparative result. The average Chi Square value obtained is 40581.68. Since the technique is practically implemented for the unique block length of only 8 bits, the average encryption time (0.559594 seconds) and the average decryption time (0.0902 seconds) are observed to be lesser than those for almost all the other proposed techniques. Graphically these comparisons have been drawn in chapter 10 [48, 49, 50, 51, 52].

**Table 7.5.1**  
**Average Encryption/Decryption Time and Chi Square Value obtained in  
 All Proposed Techniques**

<b>Proposed Technique</b>	<b>Average Encryption Time</b>	<b>Average Decryption Time</b>	<b>Average Chi Square Value</b>	<b>Average Degree of Freedom</b>
<b>RPSP</b>	<b>8.75713800</b>	<b>8.73955200</b>	<b>10701.70</b>	214
<b>TE</b>	<b>0.86703290</b>	<b>0.94175818</b>	<b>64188.04</b>	
<b>RPPO</b>	<b>0.73186806</b>	<b>7.03076904</b>	<b>85350.94</b>	
<b>RPMS</b>	<b>0.23659592</b>	<b>0.15137143</b>	<b>140196.94</b>	
<b>RSBP</b>	<b>0.74673470</b>	<b>0.11040816</b>	<b>201990.76</b>	
<b>RSBM</b>	<b>0.55959400</b>	<b>0.09020000</b>	<b>40581.68</b>	

The RSBM policy of encryption matches most with the RPMS policy, since in both these policies, there exist no cycle, no alteration in file size, and no option available for a source block to choose an encrypted block. Naturally, like in the RPMS policy, it is the higher degree of variation in block sizes that can cause a steadily improved performance in enhancing security. Results presented in section 7.4 only establish an idea of practical implementation. But for attaining a real success, the decomposition of the source stream of bits is to be done in a tactful way, as is shown in section 8.2.4, so that a reasonably long key space may be generated.



## **Formation of Secret Key**

## Contents

<b>8.1</b>	<b>Introduction</b>	<b>268</b>
<b>8.2</b>	<b>Proposed Key Structures</b>	<b>268</b>
<b>8.3</b>	<b>Conclusion</b>	<b>276</b>

## **8.1 Introduction**

Formation of the key is the most important activity in a secret key system, because even if the encryption policy is publicized, it is the secrecy of the key that helps in enhancing the security. Moreover, there should exist a tactful strategy in forming the key format, so that even applying the brute force attack the key cannot be estimated. The proper key management does not necessarily emphasize on constructing a key space as much lengthy as possible, but it deals with some factors like unambiguousness, proper invalidation, easiness in access, etc., the detailed discussion on which is made in section 9.5.4.1 [12, 23, 43].

In section 8.2, proposals are presented on key structures of different proposed techniques. Section 8.3 draws a conclusion on this issue.

## **8.2 Proposed Key Structures**

This section provides the proposal of the key structure for each of the techniques proposed.

Schematically there exist some similarities between the RPSP and the RPPO schemes, since for both these schemes cycles are generated. Accordingly, in section 1.5.7 in chapter 1, these two techniques have been categorized as “Block Cipher with Repeated Block-to-Block Conversion”. Section 8.2.1 presents a combined proposal for the format of the secret key for both the RPSP and the RPPO techniques, where instead of specifying the fixed length of the key, a superficial structure has been proposed, and depending on the exact encryption policy the length may differ.

The TE technique is one, which schematically different from all the remaining proposed techniques. In section 1.5.7 in chapter 1, this technique has been categorized as “Block Cipher with Option-based Block-to-Block Conversion”. Section 8.2.2 provides the structure of the 180-bit key for the TE technique, which is to be constructed on the basis of some fixed assumptions on different issues.

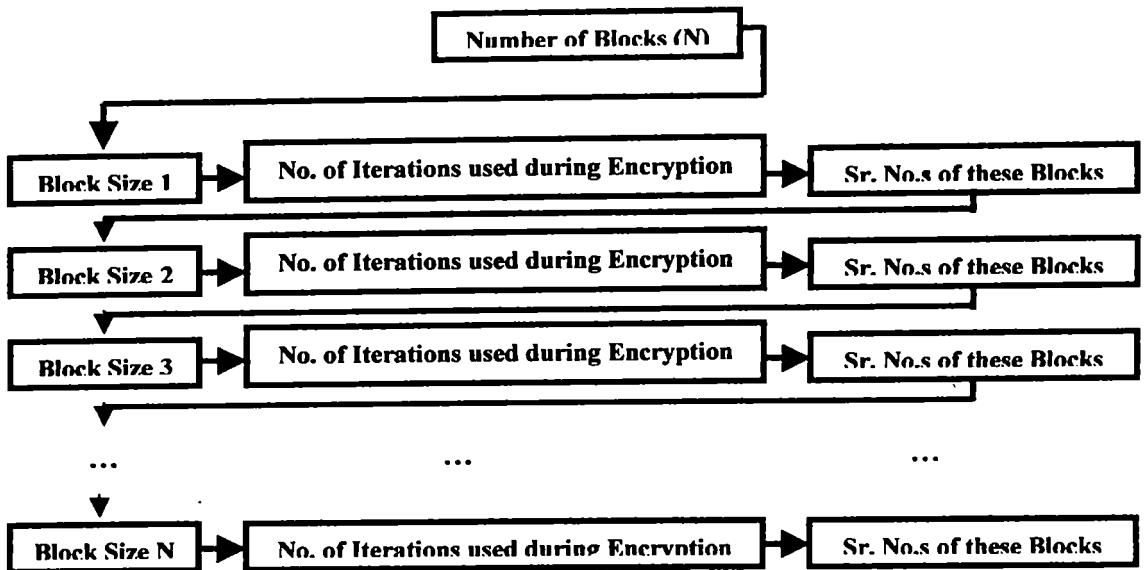
The RSBP technique, categorized as “Block Cipher with Non-Contiguous Bit-Allocation” in section 1.5.7 in chapter 1, is the only one among all the proposed techniques, in which there exists the possibility of the alteration in size. Therefore in the

proposed structure of the key there should exist the size of the original file to ensure the correct decryption. This has been presented in section 8.2.3.

For the remaining two proposed techniques, the RPMS and the RSBM, section 8.2.4 provides the proposed key structure, which is common to both. Although in section 1.5.7 in chapter 1, these two techniques have been categorized respectively as “Block Cipher with Direct Block-to-Block Conversion” and “Block Cipher with Non-Contiguous Bit-Allocation”.

### **8.2.1 Proposed Key Structure for RPSP and RPPO Techniques**

Decomposing the source file into blocks of fixed size is less secured as it requires less number of iterations to form the cycle for the entire stream of bits. Another equally important reason of not choosing blocks with fixed size is the simplicity of the key. Because in such a case, the key, which may be sent to the receiver through some secret channel, will consist of only two numbers: one, the fixed block size, and the other, number of iterations used during the encryption. If variable block size is chosen, the key will be too complicated to be guessed and for each these varying sizes if the number of iterations to form the cycle is known, then only the number of such iterations to be performed during the process of encryption is to be written in the key. Figure 8.2.1.1 presents only the proposed structure of a key applicable to both the RPSP and the RPPO techniques [49, 52].



**Figure 8.2.1.1**  
**One Suggested Key Format for RPSP and RPPO Techniques**

It is suggested that the key can be sent as a linear linked list, where each of the nodes stores the values stored inside a box, shown in figure 8.2.1.1. The number of nodes in the linked list depends on the number of blocks. If there exists  $N$  number of blocks, the number of nodes is  $(3N + 1)$ .

### 8.2.2 Proposed Key Structure for TE Technique

On the basis of a fixed encryption policy, which is applicable only to very tiny files, the structure of the secret key is formed here. The policy is as follows:

- The maximum number of characters allowed in the message is 60.
- The maximum length of a block is 32 bits.
- The maximum number of blocks is 20.

Since the block length cannot exceed 32 bits, the maximum number of bits required to represent the exact length of a block is 6. Since a total of four options are available to choose the encrypted block, maximum 3 bits are required to identify a choice. Therefore a total of 9 bits are required for one block, so that, 20 being the maximum number of blocks, altogether 180 bits are required for the entire message to be

encrypted. Figure 8.2.2.1 shows the proposed format of the 180-bit secret key. With the change in the policy, the format can be changed [48, 49].

**Position of Bits**



Size of Block 1	Option Chosen	Size of Block 2	Option Chosen	Size of Block 3	Option Chosen	Size of Block 4	Option Chosen

**Position of Bits**



Size of Block 5	Option Chosen	Size of Block 6	Option Chosen	Size of Block 7	Option Chosen	Size of Block 8	Option Chosen

**Position of Bits**



Size of Block 9	Option Chosen	Size of Block 10	Option Chosen	Size of Block 11	Option Chosen	Size of Block 12	Option Chosen

**Position of Bits**



Size of Block 13	Option Chosen	Size of Block 14	Option Chosen	Size of Block 15	Option Chosen	Size of Block 16	Option Chosen

**Position of Bits**



Size of Block 17	Option Chosen	Size of Block 18	Option Chosen	Size of Block 19	Option Chosen	Size of Block 20	Option Chosen

**Figure 8.2.2.1**  
Format of 180-Bit Secret Key for TE Encryption Technique

### 8.2.3 Proposed Key Structure for RSBP Technique

Here the structure of the key has been proposed with an assumption on the encryption policy.

The entire stream of bits of the source file is decomposed into a total of 12 segments. Out of these, on the first 11 segments, the encryption policy is applied using the RSBP technique. The final segment is remained as it is. Each segment is assigned a unique rank value. So, rank values start with 1 for the first segment, starting from the beginning of the file, and the final segment on which the encryption policy is to be applied is with the rank value of 11. In each segment, blocks are constructed of the unique size, but the maximum number of blocks in one segment cannot exceed a limiting value [50, 54].

The relationship among the rank value of a segment (R), the unique block size in the segment (S), and the maximum number of blocks in the segment (N) is established by the following topology:

For the segment of the rank R, there can exist a maximum of  $N = 2^{14-R}$  blocks, each of the unique size of  $S = 2^{14-R}$  bits, R starting from 1 and moving till 11.

For different values of R, following segments are generated:

Segment with  $R = 1$  formed with the first maximum 8192 blocks, each of size 8192 bits;

Segment with  $R = 2$  formed with the next maximum 4096 blocks, each of size 4096 bits;

Segment with  $R = 3$  formed with the next maximum 2048 blocks, each of size 2048 bits;

Segment with  $R = 4$  formed with the next maximum 1024 blocks, each of size 1024 bits;

Segment with  $R = 5$  formed with the next maximum 512 blocks, each of size 512 bits;

Segment with  $R = 6$  formed with the next maximum 256 blocks, each of size 256 bits;

Segment with  $R = 7$  formed with the next maximum 128 blocks, each of size 128 bits;

Segment with  $R = 8$  formed with the next maximum 64 blocks, each of size 64 bits;

Segment with  $R = 9$  formed with the next maximum 32 blocks, each of size 32 bits;

Segment with  $R = 10$  formed with the next maximum 16 blocks, each of size 16 bits;

Segment with  $R = 11$  formed with the next maximum 8 blocks, each of size 8 bits.

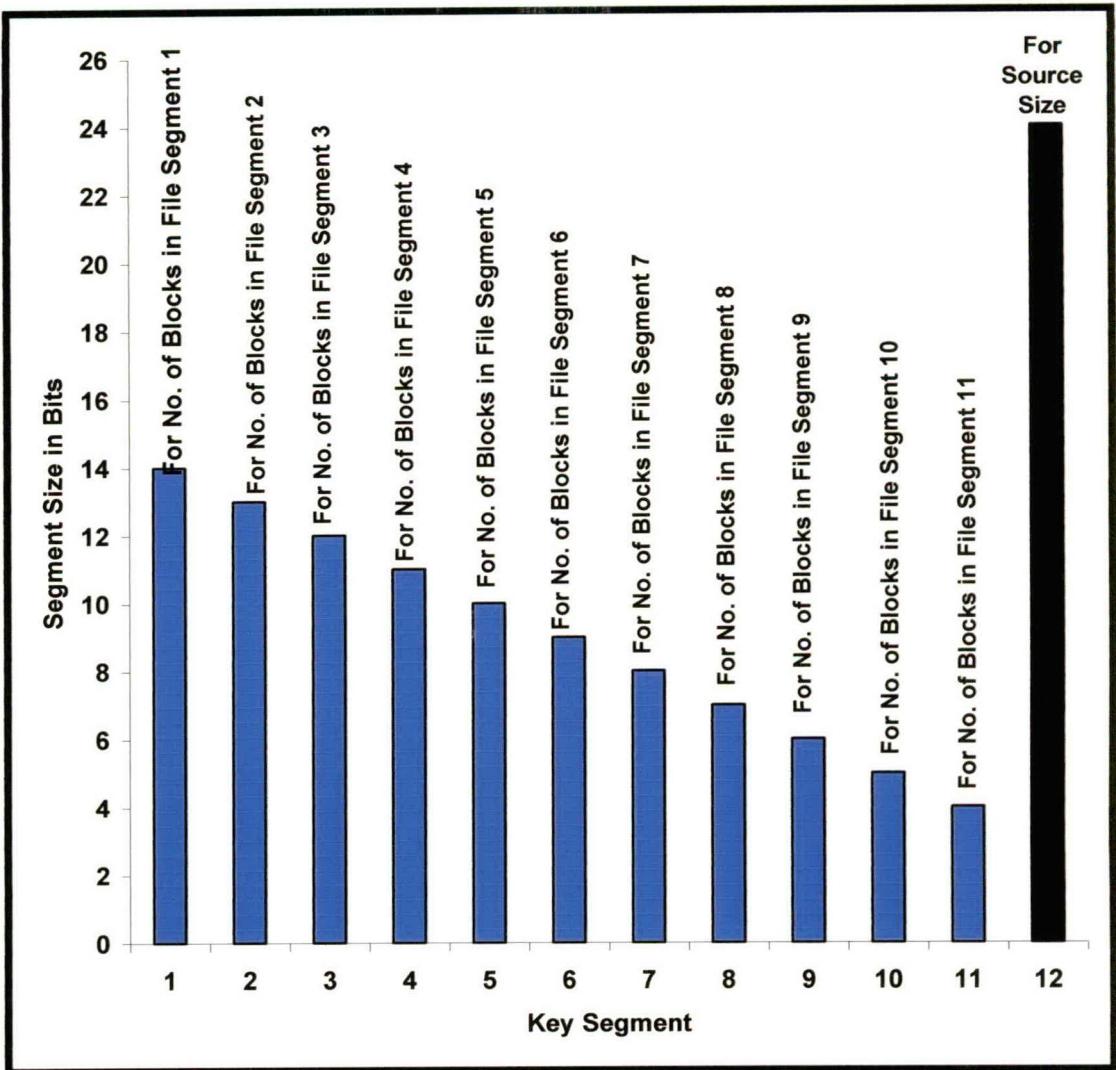
Since the total number of segments, the maximum number of blocks in a certain segment, and the size of blocks for a certain segment are fixed, a static structure of the key can be formed.

In the structure of the secret key, a total of 12 segments should exist, the first 11 of which are corresponding to the exact numbers of blocks in the respective segments of bits, and the final segment in the key stores the original file size. With this proposed format, the first 11 segments in the key require respectively 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, and 4 bits. The final segment requires 24 bits to accommodate the source size, since, As per the calculation, with this key structure, a file of size 11.18 MB can be encrypted, and to store this size in the key, 24 bits are required. Therefore the total size of the proposed key is 123 bits.

Figure 8.2.3.1 exhibits the structure of this proposed format of key. Here each segment is shown through a pillar, and the number of bits in a segment equals to the height of the pillar. The left-most pillar stands for the first segment from the MSB position, and so on. The pillars are made of two colors. There are eleven blue pillars and one black pillar.

Each blue pillar stands for storing the number of blocks in a segment. For example, the left-most blue pillar is made of height 14, which indicates that this segment in the key is of length 14 bits, and hence it can provide information on exactly there are how many 8192-bit blocks in the segment with R=1, since it is fixed that maximum 8192 blocks can be present in this segment, and to present 8192 in modulo-2 notation, 14 bits are required.

The only black pillar is used in the final segment of the key to store the source file size. The height for this pillar is taken as 24, so that 24 bits are allocated for the source file size.



**Figure 8.2.3.1**  
**123-bit Key Format with 12 Segments for RSBP Technique**

#### 8.2.4 Proposed Key Structure for RPMS and RSBM Techniques

Here in the RPMS and the RSBM techniques, unlike the RPSP and the RPPO, there exist no formation of cycle; unlike the TE technique, there exist no option regarding the encryption; and unlike the RSBP technique, there exist no alteration in file size. In that respect, the structure of the key for the techniques of RPMS and RSBM should be consisting of the minimal information, only with sizes of different blocks to be constructed. Here this structure has been proposed only by removing the last segment from the key, shown in figure 8.2.3.1, corresponding to the RSBP encryption technique.

But in an attempt to produce a reasonably long key space, a little alteration is made in the structures of different file segments [50, 51].

For the segment of the rank R, there can exist a maximum of  $N = 2^{15-R}$  blocks, each of the unique size of  $S = 2^{15-R}$  bits, R starting from 1 and moving till 11.

For different values of R, following segments are generated:

Segment with R=1 formed with the first maximum 16384 blocks, each of size 16384 bits;

Segment with R=2 formed with the first maximum 8192 blocks, each of size 8192 bits;

Segment with R=3 formed with the next maximum 4096 blocks, each of size 4096 bits;

Segment with R=4 formed with the next maximum 2048 blocks, each of size 2048 bits;

Segment with R=5 formed with the next maximum 1024 blocks, each of size 1024 bits;

Segment with R=6 formed with the next maximum 512 blocks, each of size 512 bits;

Segment with R=7 formed with the next maximum 256 blocks, each of size 256 bits;

Segment with R=8 formed with the next maximum 128 blocks, each of size 128 bits;

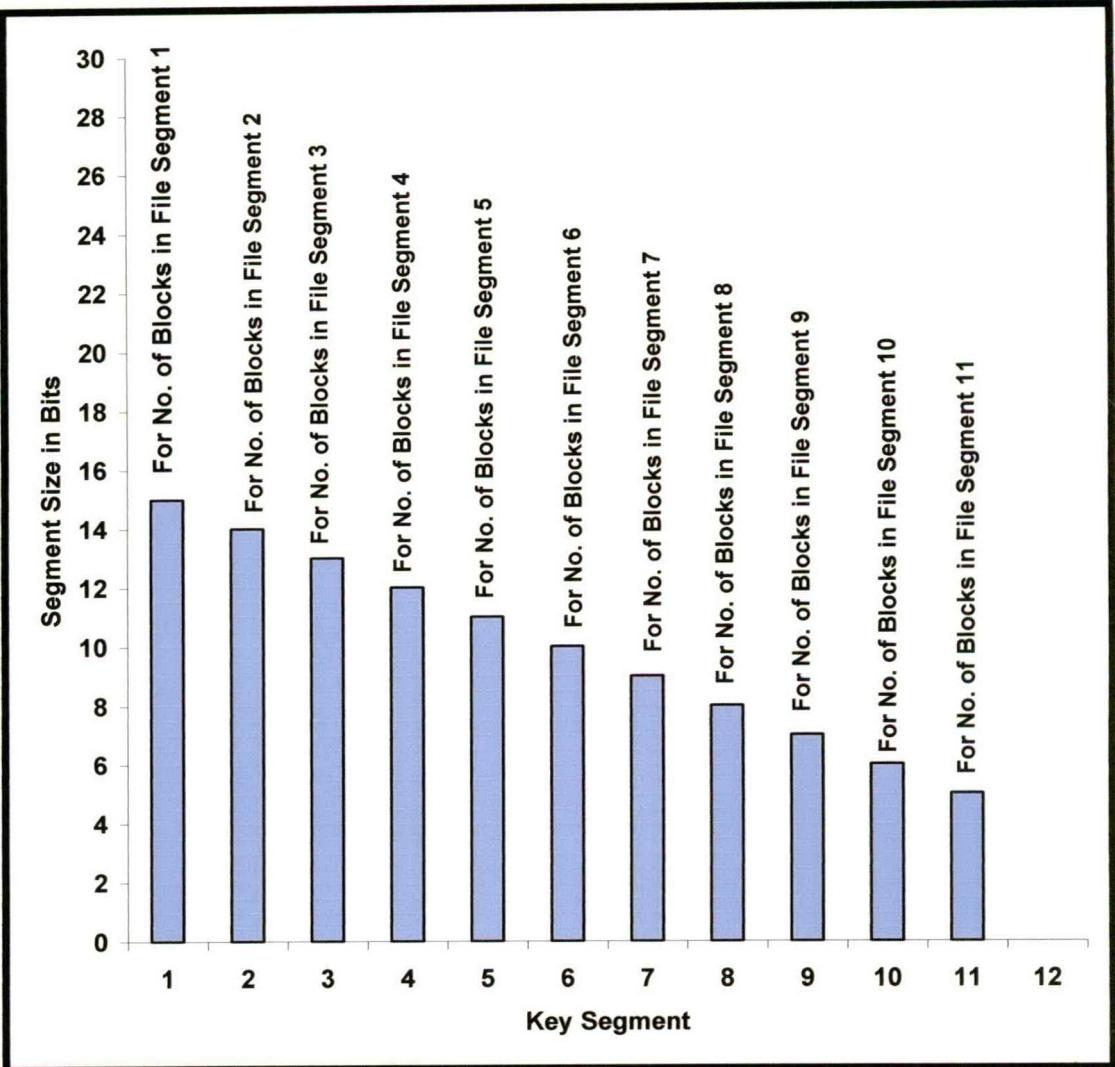
Segment with R=9 formed with the next maximum 64 blocks, each of size 64 bits;

Segment with R=10 formed with the next maximum 32 blocks, each of size 32 bits;

Segment with R=11 formed with the next maximum 16 blocks, each of size 16 bits;

With such a structure, the key space becomes of 110 bits long and a file of the maximum size of around 44.74 MB can be encrypted using either of the RPMS and the RSBM techniques.

Figure 8.2.4.1 presents this structure. This figure to a large extend matches with figure 8.2.3.1. But here since the source file size is not needed to be stored, there does not exist any pillar for that purpose.



**Figure 8.2.4.1**  
**110-bit Key Format with 11 Segments for RPMS and RSBM Technique**

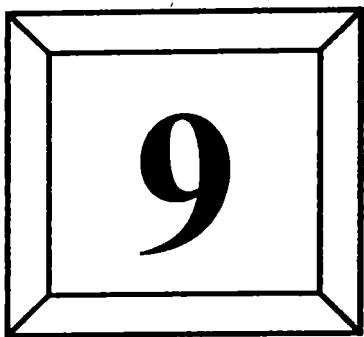
### 8.3 Conclusion

Table 8.3.1 summarizes proposed key structures for different proposed techniques.

**Table 8.3.1**  
**Proposed Key Structures for Different Proposed Techniques**

<b>Proposed Techniques</b>	<b>Proposed Key Structure</b>
<b>RPSP</b>	<b>No fixed length is proposed, the use of linked list is suggested</b>
<b>TE</b>	<b>180-bit key is proposed on the basis of a pre-fixed set of rules</b>
<b>RPPO</b>	<b>No fixed length is proposed, the use of linked list is suggested</b>
<b>RPMS</b>	<b>110-bit key is proposed that can encrypt files of upto 44.74 MB size</b>
<b>RSBP</b>	<b>123-bit key is proposed that can encrypt files of upto 11.18 MB size</b>
<b>RSBM</b>	<b>110-bit key is proposed that can encrypt files of upto 44.74 MB size</b>

A long key space enhances the security, but that does not necessarily mean that the key is to be constructed of excessively long size, because this, in turn, may be regarded as an overhead. Making a proper balance between these two issues, all the structures have been presented. But with the availability of more flexibility mainly in the issue of “blocks formation”, much longer key spaces can be constructed for different proposed techniques [12, 23, 37, 41].



## **Encryption Through Cascaded Implementation of The Proposed Techniques**

<b><u>Contents</u></b>	<b><u>Pages</u></b>
<b>9.1      Introduction</b>	<b>280</b>
<b>9.2      Implementation</b>	<b>282</b>
<b>9.3      Results</b>	<b>288</b>
<b>9.4      Analysis</b>	<b>293</b>
<b>9.5      Proposal of An Integrated Encryption System</b>	<b>293</b>
<b>9.6      Conclusion</b>	<b>300</b>

## **9.1 Introduction**

The approach of the cascaded implementation of the proposed techniques is an attempt to synchronize all the independent proposed techniques. It introduces a new dimension in the endeavour of ensuring data security to the highest possible level. Section 9.1.1 points out the basic principle of the cascaded approach. The strength of the approach is explained in section 9.1.2.

### **9.1.1 Basic Principle of Cascading**

The basic principle in the approach of implementing the proposed techniques in the cascaded manner without any repeated implementation of the same technique is as follows [36]:

1. On a source file under consideration, arbitrarily chosen one proposed encryption technique is to be implemented. There exist a total of six choices to select an encryption technique.
2. On the file generated through step 1, arbitrarily chosen one proposed encryption technique out of the remaining five is to be implemented.
3. On the file generated through step 2, arbitrarily chosen one proposed encryption technique out of the remaining four is to be implemented.
4. On the file generated through step 3, arbitrarily chosen one proposed encryption technique out of the remaining three is to be implemented.
5. On the file generated through step 4, arbitrarily chosen one proposed encryption technique out of the remaining two is to be implemented.
6. Finally, on the file generated through step 5, the only one left proposed encryption technique is to be implemented.

For a certain sequence of the encryption techniques, implemented during the process of the cascaded encryption, the same sequence is to be followed in exactly the reverse order for the purpose of decryption.

### **9.1.2 Strength of the Cascaded Approach**

The biggest advantage of using the cascaded approach is the unavoidable necessity of a long key space. Following are some mandatory specifications to be accommodated in the key space [36]:

- There exist a number of options for choosing a sequence of techniques. To be précis, there exist as many as 720 options to choose a sequence because six proposed techniques can be permuted in  $6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$  ways. Moreover, through a pre-transmission agreement between the sender and the receiver, if it is decided to allow cascaded implementation of the same technique, the number of options increases drastically.
- During cascading, for each technique, even if blocks are constructed of the same size, there exist a number of openings for choosing the unique block size. To enable constructing blocks of varying lengths, a high number of extra bits are to be placed in the key space.
- For the techniques like RPSP and RPPO, the number of iterations to be done during the encryption process is to be specified.
- For the technique like TE, the particular option to be followed during the encryption process is to be specified.
- For the technique of RSBP, the size of its source is to be specified.

Section 9.2 of this chapter shows a real-time implementation of this cascaded approach. Section 9.3 enlists all results obtained implementing the proposed techniques in the cascaded manner. Section 9.4 analyzes the results observed. Section 9.5 presents the proposal of a complete data encryption system, where it has been proposed to cascade the proposed techniques in a tactful way. Section 9.6 draws the conclusion on the entire activity.

## **9.2 Implementation**

For the purpose of presenting one implementation, one sample file TLIB.EXE has been chosen. This section discusses different intermediate phases of the implementation in a detailed manner with the related analysis of the approach.

Section 9.2.1 presents the encryption phase and the phase of decryption is presented in section 9.2.2. Section 9.2.3 analyzes this implementation.

### **9.2.1 Encrypting Source File using Proposed Techniques in Cascaded Manner**

Figure 9.2.1.1 exhibits the steps followed during the encryption process. The sample file TLIB.EXE is encrypted through 6-phased cascading approach to generate FOX6.EXE.

Here the source file TLIB.EXE is first encrypted using the RPSP encryption technique with the unique size of each block is inputted as 8 bits. As the result, FOX1.EXE is generated.

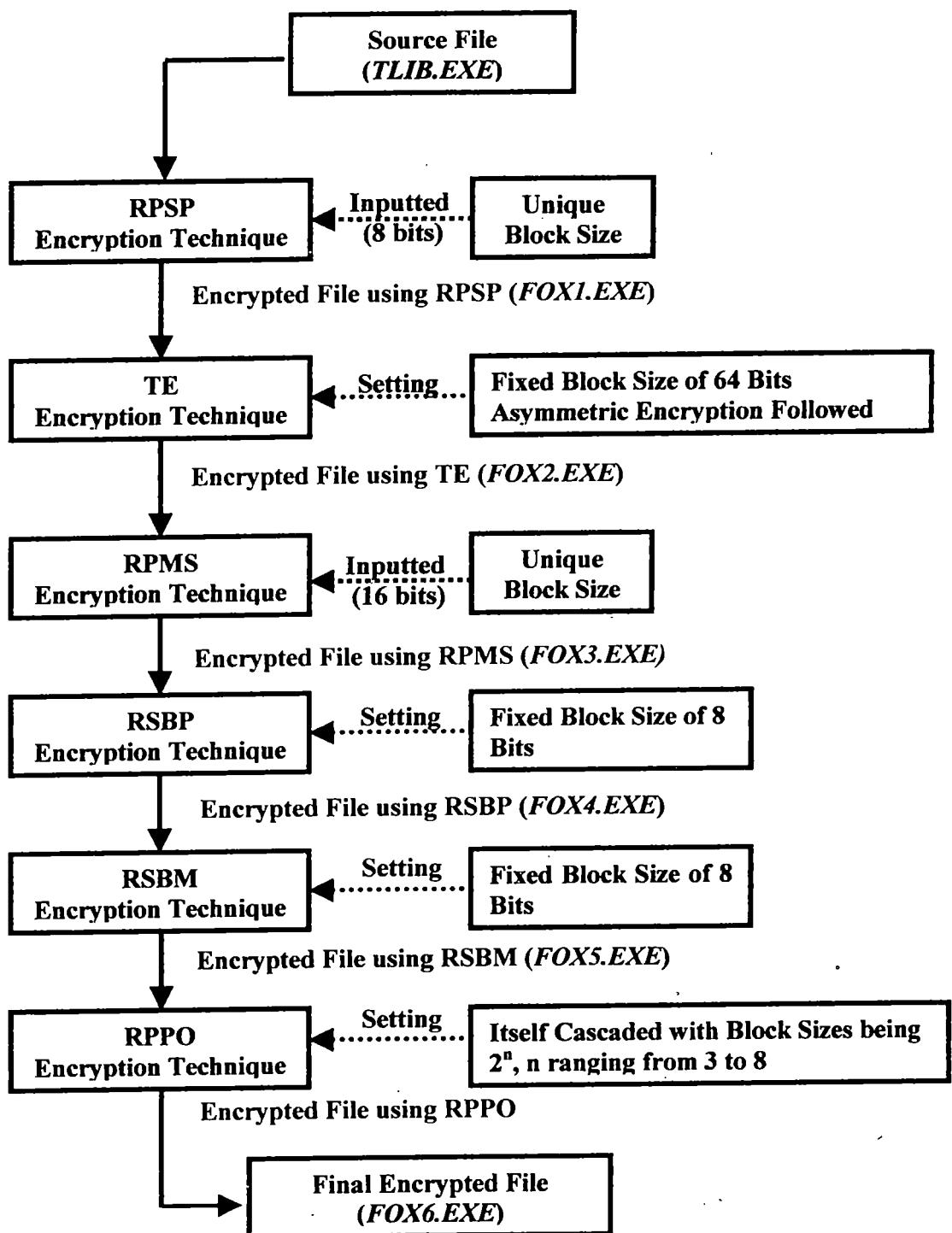
FOX1.EXE is then encrypted using the TE encryption technique with the unique block size being fixed as 64 bits. As the result, FOX2.EXE is generated.

Next FOX2.EXE is encrypted for inputted unique block size of 16 bits using the RPMS encryption technique. As the result, FOX3.EXE is generated.

FOX3.EXE is then fed into the RSBP encryption technique with the fixed block size of 8 bits. As the result, FOX4.EXE is generated.

Then the RSBM encryption technique is used to encrypt FOX4.EXE using the fixed block size of 8 bits. As the result, FOX5.EXE is generated.

Finally, FOX5.EXE is fed into the RPPO encryption technique, which is itself cascaded with block sizes being  $2^n$ , n ranging from 3 to 8. As the result, FOX6.EXE is generated, which is the final encrypted file in this cascaded approach.



**Figure 9.2.1.1**  
Steps followed during Cascaded Encryption.

### **9.2.2 Decrypting Encrypted File, Encrypted in Section 9.2.1**

Figure 9.2.2.1 exhibits the steps followed during the decryption process. The sample file FOX6.EXE is encrypted through 6-phased cascading approach to generate RAT6.EXE, which is in turn the source TLIB.EXE.

In this cascaded approach of decryption different proposed techniques are adopted exactly in the reverse order.

FOX6.EXE is first fed into the RPPO decryption technique, which is itself cascaded with block sizes being  $2^n$ , n ranging from 8 to 3. As the result, RAT1.EXE is generated.

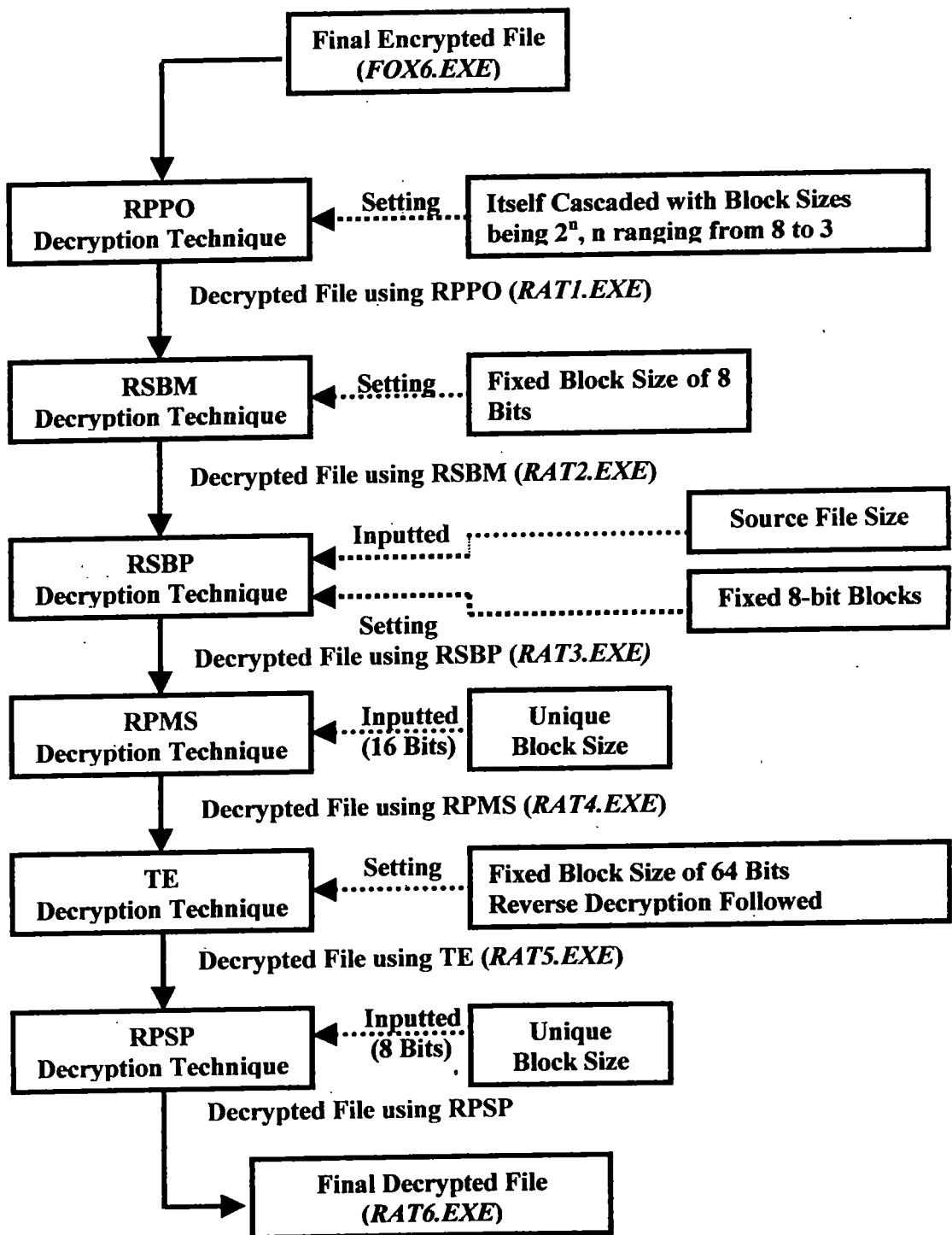
Then RAT1.EXE is decrypted using RSBM decryption technique with the fixed block size set at 8 bits. As the result, RAT2.EXE is generated.

Next the RSBP decryption technique is used to decrypt RAT2.EXE. For the decryption, fixed block size is set at 8 bits and the size of the original fine is inputted. Through the process of decryption. RAT3.EXE is generated.

RAT3.EXE is then fed into the RPMS decryption technique with the unique block size being inputted as 16 bits. As the result, RAT4.EXE is generated.

Then RAT4.EXE is decrypted using the TE decryption technique with the fixed block size being set at 64 bits. As the result, RAT5.EXE is generated.

Finally, the RPSP decryption technique is used to decrypt RAT5.EXE with the unique block size being inputted as 8 bits. As the result, RAT6.EXE is generated, which is in turn the original file TLIB.EXE.

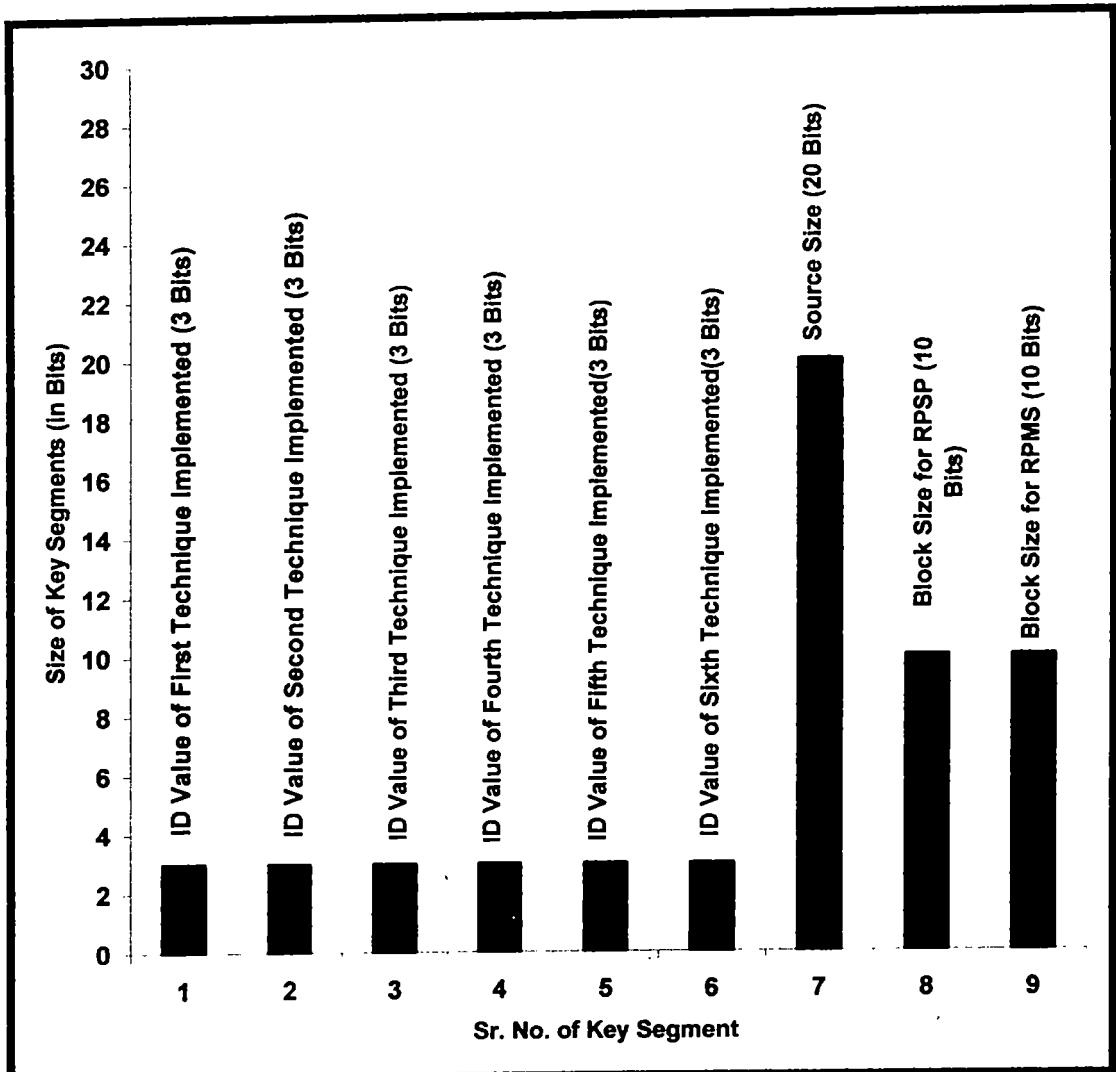


**Figure 9.2.2.1**  
Steps followed during Cascaded Decryption

### **9.2.3 Analysis of Implementation**

The strength of this cascaded implementation is not highlighted through the frequency distribution test or the chi square value test. As it is shown section 9.3, no universal conclusion can be drawn regarding this cascaded approach in terms of these two aspects. But the real strength lies in the possible formation of a large key space. As it is shown in figure 9.2.3.1, for a successful decryption, a 58-bit key must be available to the receiver of the encrypted message following this cascaded encryption. In section 9.5, it has been shown that with allowing the availability of much more flexibility in this cascaded approach, the key space increases drastically.

In figure 9.2.3.1, in the structure of the 58-bit key, a total of 9 segments exist. Out of these, the first 6 segments, each with 3 bits, are used to identify the exact sequence of techniques followed during encryption. As there are 6 techniques, a 3-bit segment is required to identify any of those. The 7<sup>th</sup> segment is of 20 bits of size, used to store the size of the source file in base-2 format, which is sufficient to accommodate the file size information of a file with 1.04 MB size approximately. This information is required while decrypting using the RSBP decryption technique. The final two segments, each of 10-bit size, are required to store the unique block sizes used during the implementation of the RPSP and the RPMS techniques. For the remaining four techniques, this information is not required here since those implementations have been done for fixed block sizes.



**Figure 9.2.3.1**  
**Structure of a 58-Bit Key with 9 Segments**

In this way, it is seen that even for such type of simple cascading, the key of the length of 58 bits is required for the purpose of having successful decryption. To avail the following options, many more bits are to be added in the key space:

- Not to fix the unique block size; to input it only during the implementation for all techniques
- Allowing source size to be more than 1.04 MB
- Allowing one technique to be implemented more than once
- Allowing larger unique size for blocks
- Allowing varying block lengths at least for some techniques

- Allowing the choosing of option in implementing the TE encryption technique
- Allowing the choosing of the number of iterations to be performed during the process of encryption using the RPPO and the RPSP techniques

### 9.3 Results

For the purpose of observing results, out of the 50 sample files that have been considered during implementing the proposed techniques independently, only 10, arbitrarily taking 2 from each category, have been considered. For each implementation, exactly the same approach as was mentioned in section 9.2 has been considered [36, 44].

Table 9.3.1 enlists these results. The left-most column of the table enlists names of different source files with their respective sizes mentioned in brackets. Files considered for this purpose are UNZIP.EXE, TCDEF.EXE, WIN.COM, KEYB.COM, HIDCI.DLL, PFPICK.DLL, USBD.SYS, IFSHLP.SYS, ARITH.CPP, and START.CPP. Their sizes are in the range of 3216 bytes to 58368 bytes [55, 56].

Column 2 to column 5 enlists names of intermediate files generated after applying the RPSP, the TE, the RPMS, and the RSBP encryption techniques respectively. Below each entry in brackets the Chi Square value between the original file and the file last generated has been given.

During the cascaded implementation, after implementing the RSBP encryption technique, the file size alters. For each file, the corresponding altered size is given in column 6.

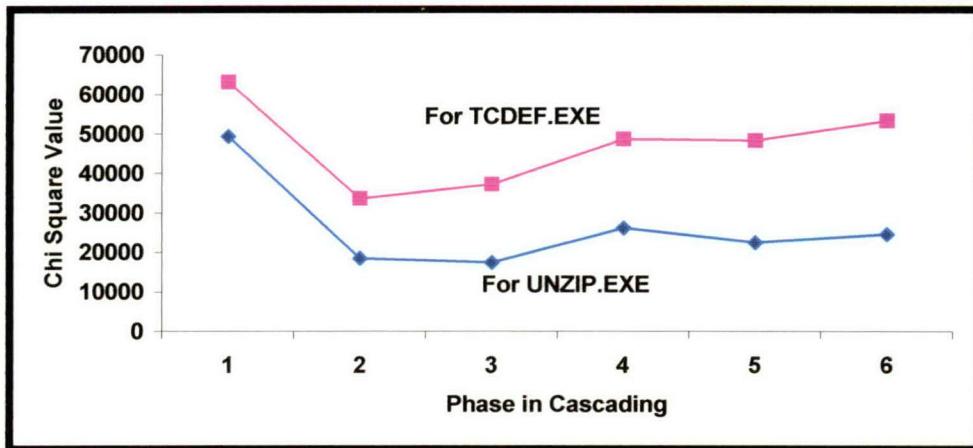
Column 7 and column 8 are similar in nature like column 2 to column 5. Here results after implementing the RSBM and the RPPO encryption techniques have been enlisted.

**Table 9.3.1**  
**Result of Cascaded Implementation**

Source File (Size)	File applying RPSP (C.S.V.)	File applying TE (C.S.V.)	File applying RPMS (C.S.V.)	File applying RSBP (C.S.V.)	Changed File Size	File applying RSBM (C.S.V.)	File applying RPPO (C.S.V.)
UNZIP.EXE (23044)	F11.EXE (49310)	F12.EXE (18417)	F13.EXE (17338)	F14.EXE (26068)	24380	F15.EXE (22351)	F16.EXE (24489)
TCDEF.EXE (11611)	F21.EXE (63115)	F22.EXE (33517)	F23.EXE (37118)	F24.EXE (48593)	12171	F25.EXE (48206)	F26.EXE (53202)
WIN.COM (24791)	F11.COM (99157)	F12.COM (52643)	F13.COM (45311)	F14.COM (400444)	24283	F15.COM (222935)	F16.COM (171505)
KEYB.COM (19927)	F21.COM (78263)	F22.COM (59081)	F23.COM (49197)	F24.COM (121713)	20332	F25.COM (77737)	F26.COM (101565)
HIDCI.DLL (3216)	F11.DLL (12914)	F12.DLL (9583)	F13.DLL (8841)	F14.DLL (9221)	3433	F15.DLL (8446)	F16.DLL (9553)
PFPICK.DLL (58368)	F21.DLL (210525)	F22.DLL (174781)	F23.DLL (169556)	F24.DLL (189747)	61350	F25.DLL (192864)	F26.DLL (244145)
USBD.SYS (18912)	F11.SYS (135449)	F12.SYS (96371)	F13.SYS (95886)	F14.SYS (98424)	20140	F15.SYS (92551)	F16.SYS (118467)
IFSHLP.SYS (3708)	F21.SYS (15685)	F22.SYS (7162)	F23.SYS (6297)	F24.SYS (15258)	3889	F25.SYS (10257)	F26.SYS (10566)
ARITH.CPP (9558)	F11.CPP (10842)	F12.CPP (16595)	F13.CPP (12375)	F14.CPP (13045)	10056	F15.CPP (12906)	F16.CPP (12081)
START.CPP (14557)	F21.CPP (80174)	F22.CPP (53041)	F23.CPP (19420)	F24.CPP (40349)	15293	F25.CPP (47043)	F26.CPP (32577)

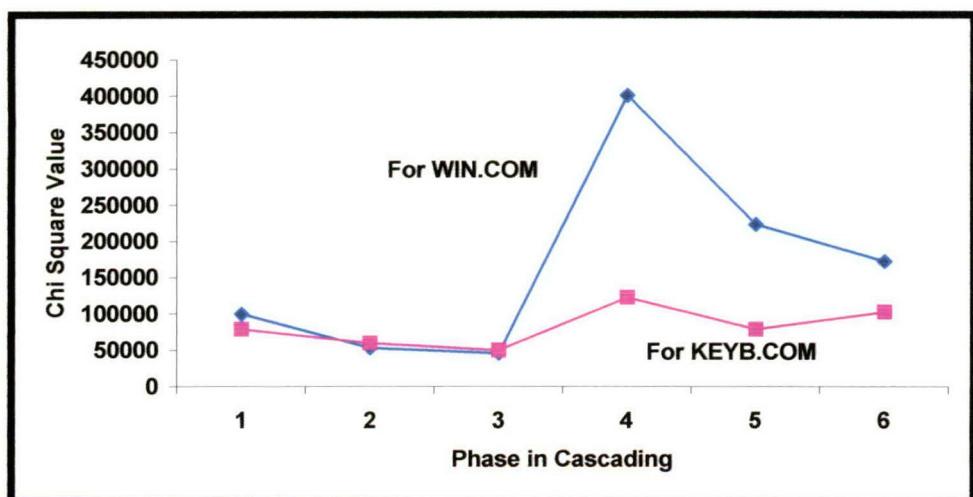
Figure 9.3.1 to figure 9.3.5 exhibit the variation in the chi square values after different intermediate and final steps during the encryption process.

In figure 9.3.1, the curve of color pink stands for the variation in Chi Square values for the file TCDEF.EXE, and the curve of color blue stands for the same for the file UNZIP.EXE. For TCDEF.EXE, the maximum Chi Square value is observed after phase 1 when it rises to 49310, so that on the average a steady decrement is observed at the end of the entire approach. The same is also true for UNZIP.EXE. Here the maximum Chi Square value is observed as 63115 after phase 1.



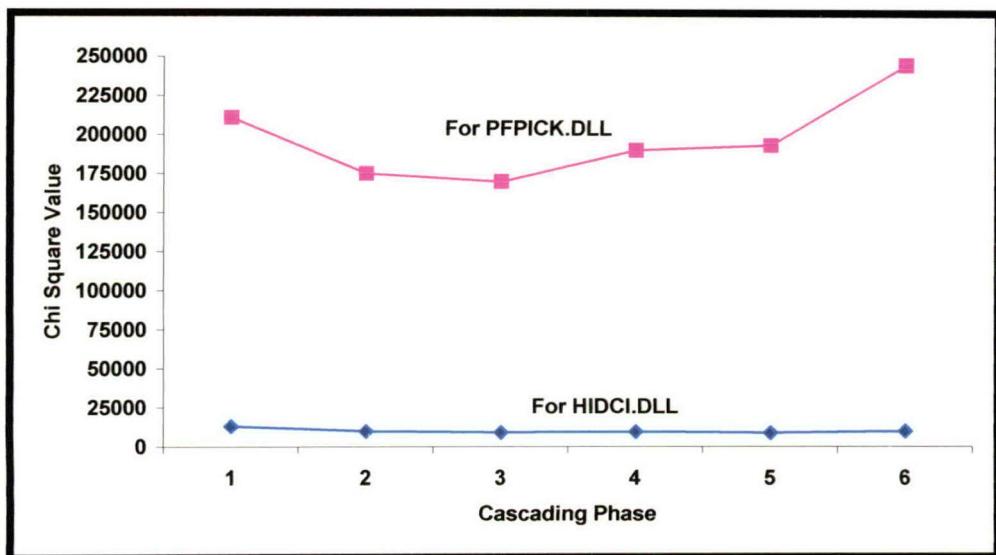
**Figure 9.3.1**  
**Variation in Chi Square Values during Cascaded Encryption for .EXE Files**

In figure 9.3.2, the curve of color pink stands for the variation in Chi Square values for the file KEYB.COM, and the curve of color blue stands for the same for the file WIN.COM. For KEYB.COM, the maximum Chi Square value is observed after phase 4 when it rises to 121713. The same is also true for WIN.COM. Here the maximum Chi Square value is observed as 400444 after phase 4. In each of the cases, in comparison to the Chi Square value obtained after phase 1, a higher value is obtained at the end of the cascaded approach.



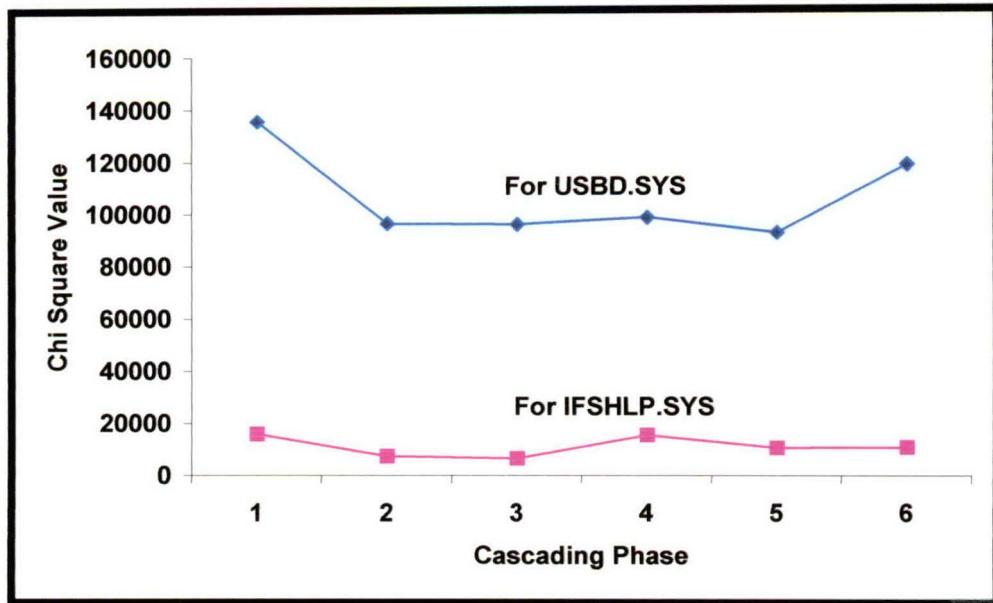
**Figure 9.3.2**  
**Variation in Chi Square Values during Cascaded Encryption for .COM Files**

In figure 9.3.3, the curve of color pink stands for the variation in Chi Square values for the file PFPICK.DLL, and the curve of color blue stands for the same for the file HIDCI.DLL. For PFPICK.DLL, the cascaded implementation is observed to be very effective as the maximum Chi Square value is observed after the final phase when it rises to 244145. For HIDCI.DLL, the case is opposite as the maximum Chi Square value is obtained after the phase 1, which is 12914. But here different Chi Square values are observed to be not of much difference, so that the curve generated looks roughly like a straight line.



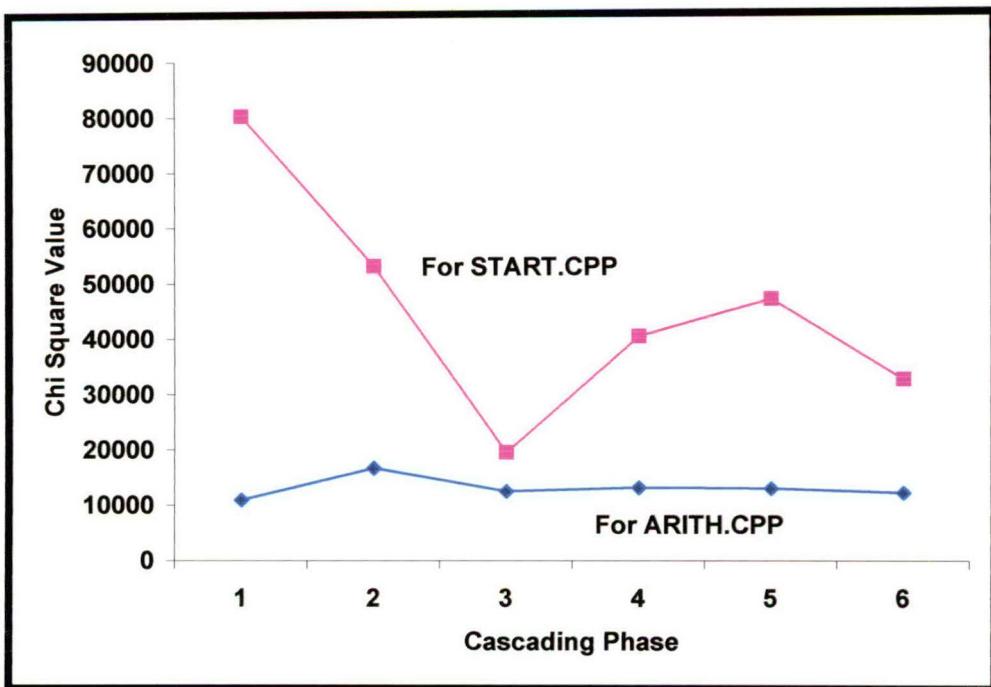
**Figure 9.3.3**  
**Variation in Chi Square Values during Cascaded Encryption for .DLL Files**

In figure 9.3.4, the curve of color pink stands for the variation in Chi Square values for the file IFSHLP.SYS, and the curve of color blue stands for the same for the file USBD.SYS. For IFSHLP.SYS, the cascaded implementation is observed to be very effective as the maximum Chi Square value is observed after the final phase when it rises to 15685, although different Chi Square values are observed to be not of much difference, so that the curve generated looks roughly like a straight line. For USBD.SYS, the case is opposite as the maximum Chi Square value is obtained after the phase 1, which is 135449. But here the Chi Square value obtained at the end of the cascaded approach does not differ too much from the maximum obtained value.



**Figure 9.3.4**  
**Variation in Chi Square Values during Cascaded Encryption for .SYS Files**

In figure 9.3.5, the curve of color pink stands for the variation in Chi Square values for the file START.CPP, and the curve of color blue stands for the same for the file ARITH.CPP. For START.CPP, the cascaded implementation is observed to be the least effective as the maximum Chi Square value is observed after phase 1 when it rises to 80174, and the subsequent values are observed to be much lesser than this value. For ARITH.CPP, the curve looks like a straight line, as there is not much difference in different Chi Square values obtained. The maximum value (16595) is obtained after phase 2.



**Figure 9.3.5**  
**Variation in Chi Square Values during Cascaded Encryption for .CPP Files**

#### 9.4 Analysis

All results obtained in section 9.3 establish the fact that there exists no tendency of a steady progress in chi square value between the source file and the file last generated during encryption. In each case, the Chi Square value obtained is very high in comparison with the standard value. Hence it can be said that the intermediate files generated are heterogeneous in nature with 1% uncertainty, and as a result, it can be concluded that the cascaded scheme ensures high degree of security. Moreover, as it was shown in section 9.2, even for this simple cascading, it requires a 58-bit key space to enable to have the correct decryption. Therefore it is easy to conclude that with the availability of having much more flexibility in the process of cascading, comparatively much longer key space may be generated, which is discussed in section 9.5.

#### 9.5 Proposal of An Integrated Encryption System

This section presents a proposal of an integrated data encryption system on the basis of the principle of cascading using the six proposed techniques. Section 9.5.1 points

out the principles of the proposed system. Different schematic characteristics of the system are enlisted in section 9.5.2. Section 9.5.3 mentions different operational characteristics of the system. Section 9.5.4 presents the structure of the secret key to be used in the system.

### **9.5.1 Principles of the Proposed Integrated System**

Following are the basic principles of the system:

- The fixation of the secret key is a run-time activity, during the process of encryption.
- Following the schematic restrictions (discussed in section 9.5.2) applicable to the encryption process, the task of encryption can be carried out and with each and every activity, the key space is upgraded
- During the process of decryption, the key generated at the end of encryption, is to be followed in each activity.

### **9.5.2 Schematic Characteristics of the Proposed Integrated System**

The proposed system is a 16-level cascaded approach. Following points simplify the proposed concept to be followed during the process of encryption:

- Different proposed encryption techniques could be implemented altogether a maximum total of 16 times, although the encryption authority is given the flexibility to deserve the right to terminate the process of encryption after any number of implementations.
- Each proposed encryption technique could be implemented for a maximum 3 times, ranging from “no implementation at all” to 3 implementations.
- For each technique, during each implementation, a unique block length is to be considered and that should not exceed 512 bits. This length should depend on the choice of the encryption authority, so that it is to be inputted to the system.

- For each technique, during different implementations, different unique block sizes could be considered, but each of those must not exceed 512 bits.
- While using the RPSP or the RPPO technique, the number of iterations to be performed during the process of encryption is exactly integral half of the total number of iterations required to complete the cycle.

### **9.5.3 Operational Characteristics of the Proposed Integrated System**

During the operational phase of the encryption process, the system is to provide following facilities:

- At any point of time during the cascaded encryption, following steps are to be used sequentially to finalize a particular technique to be implemented:
  - The encryption authority selects a technique.
  - The system provides the corresponding chi square value between the original file and the file that would be generated if the technique is applied.
  - The encryption authority, may be on judging the value, either finalizes this implementation or cancels it to move to any other technique.
- With the finalization of each step during this cascaded implementation, the secret key, which is stored in a data file, is upgraded and the encryption authority is given the option to view the key at any point of time.
- If the encryption authority attempts to violet any of the schematic characteristics during the cascaded implementation, the system rejects this attempt by visualizing the reason of this rejection.
- The option for the termination of the cascading process is always to be available in the system, so that at any time the encryption

authority can select this option, and, on confirmation, the termination can be granted.

#### **9.5.4 Structure of the Secret Key for the Integrated System**

Section 9.5.4.1 points out different criteria to be fulfilled for the formation of efficient secret key. Section 9.5.4.2 is a discussion on the different proposed segments in the secret key. Finally, section 9.5.4.3 shows the proposed format for the secret key.

##### **9.5.4.1 Criteria for an Efficient Key Generation**

For the purpose of constructing a secret key, a perfect key management is needed, the objective of which is not to unnecessarily increase the key space, but to accommodate all necessary information in the key in an efficient manner, so that the following set of criteria is satisfied [12, 23, 43]:

- **No ambiguity:** There should not be any mismatch between the schematic characteristics to be followed and the information stored in the key.
- **Easiness in accessing information:** All relevant information should be accommodated in the key as much adjacent as possible.
- **No repetition in providing information:** Directly or even indirectly there should not exist the repetition of any information.
- **Proper invalidation:** In certain situation, if any segment in the key becomes nonfunctioning, then that should be accommodated tactfully by perfectly invalid value.

Considering all these schematic characteristics and different criteria to be fulfilled, the structure of the 252-bit secret key has been proposed.

##### **9.5.4.2 Formation of Different Segments in the 252-bit key**

This section presents the way different segments in the 252-bit key are to be formed [12, 23, 43].

1. **Segment to identify the encryption technique:** Since there exist a total of 6 encryption techniques, a 3-bit segment is needed to

identify a technique. For instance, table 9.5.4.2.1 shows one arbitrarily fixed set of ID values.

**Table 9.5.4.2.1**  
**Arbitrarily chosen ID Values for Different Encryption Techniques**

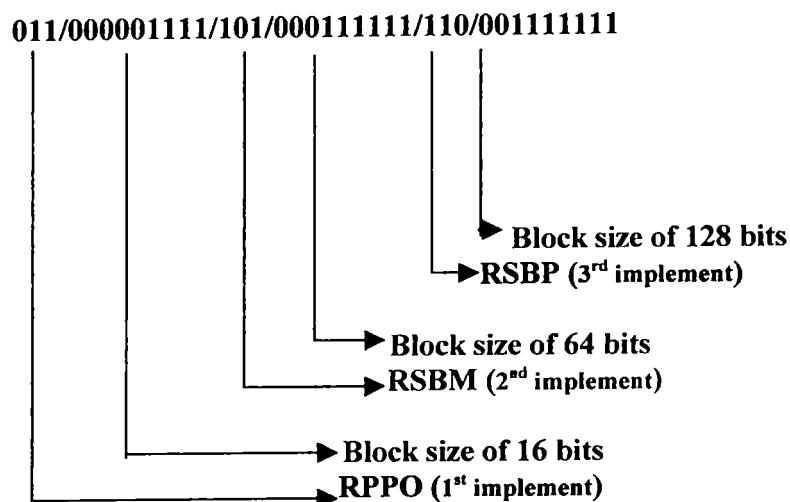
Encryption Technique	Assigned ID Value
RPSP	001
TE	010
RPPO	011
RPMs	100
RSBM	101
RSBP	110

An ID value of “000” indicates no implementation of any encryption technique.

2. **Segment to identify Block Size:** As per the schematic characteristic, since the maximum block size can be 512 bits, a 9-bit segment is needed to express the unique block size used, with the assumption that the size of a N-bit block is represented by the 9-bit binary value corresponding to  $(N-1)$ . For example, if the unique block size is 64 bits ( $N = 64$ ), the 9-bit binary value expressing this size would be “00111111”, which is the 9-bit binary value corresponding to 63 ( $N-1 = 63$ ). The formation of this segment is based on the principle that the block size of 0 bit is impossible.

The 252-bit proposed key, starting from the MSB position, first accommodates the segment 1 (of 3 bits) and the segment 2 (of 9 bits) for the first encryption technique implemented, and since there can be maximum 16 implementations, altogether the first  $16 \times (3+9) = 192$  positions are preserved for accommodating the values of these two segments for 16 implementations. “No implementation” is indicated by “000”. For example, if there are

only three implementations applied, in the sequence of RPPO, RSBM, and RSBP, with unique block sizes of 16 bits, 64 bits, and 128 bits respectively, then in the 192-bit structure, the first  $3 \times (3+9) = 36$  bits using table 9.5.4.2.1, would be as the following:



Now, the first segment for each of the remaining probable 13 implementations is to be accommodated by “000” to indicate “no implementation” and hence, the respective 2<sup>nd</sup> segments are all immaterial, although it is advisable to put 9-bit null value (“000000000”) in each of those segments, to avoid any sort of confusion.

3. **Segment to identify Source File Size:** During the process of decryption, the decryption authority must have the original file size to decrypt a file encrypted by applying the RSBP encryption technique. So, it is the responsibility during the process of encryption to fill in this 20-bit segment with the original file size.
4. **Segment to identify Size of File, on which RSBP is to be implemented Second Time:** Because each RSBP decryption technique requires the size of the file on which the corresponding RSBP encryption was implemented, another 20-bit segment is allocated for that purpose.

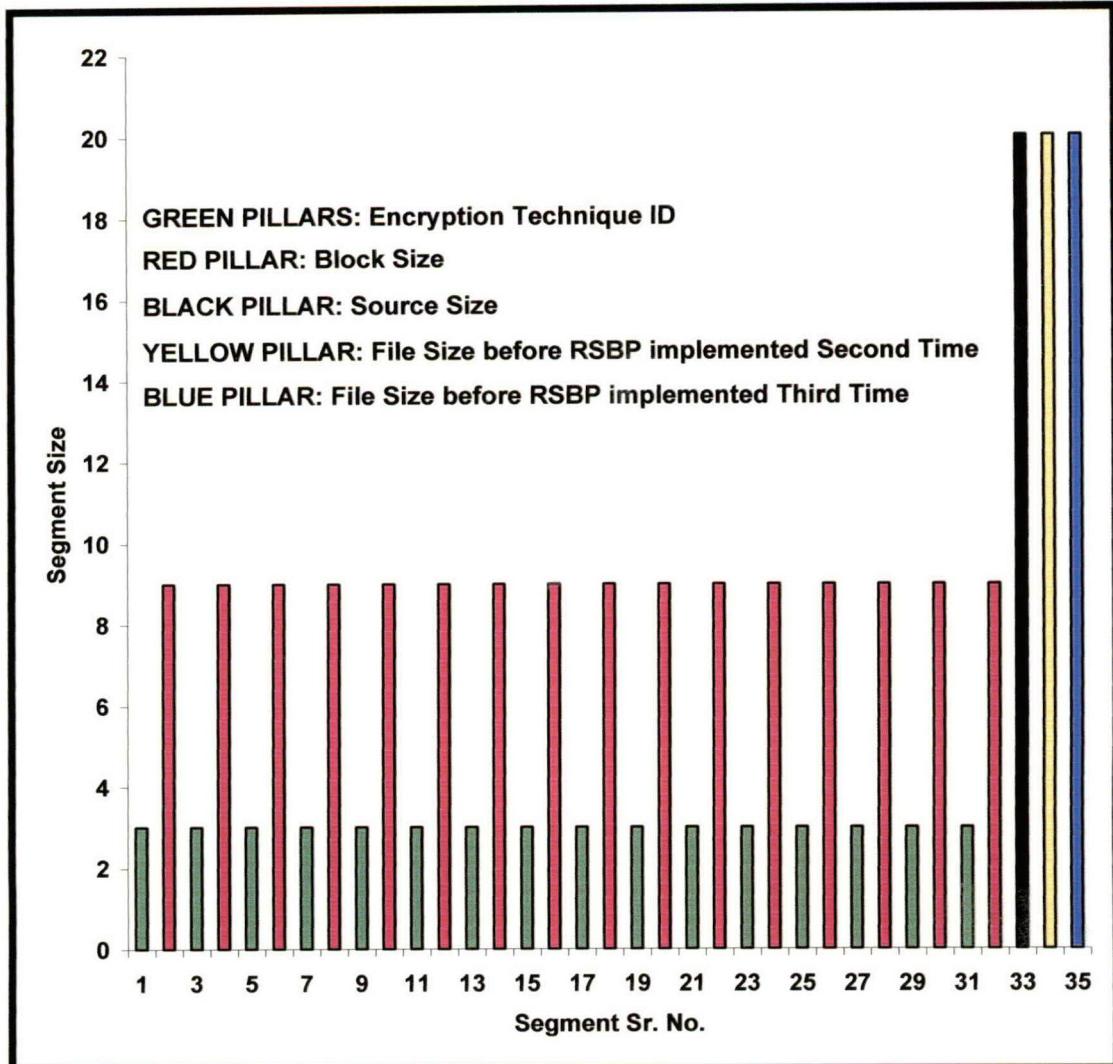
5. **Segment to identify Size of File, on which RSBP is to be implemented Third Time:** This 20-bit segment is allocated for the same reason as in 4.

Since the RSBP encryption technique, like all other techniques, can be implemented at most 3 times, there is no need of any more segment for storing size of any intermediate file.

Now, a 20-bit segment used for storing file size can store a file of size approximately upto 1.04 MB, and also the fact of "size alteration" following the RSBP encryption technique is to be considered. With this consideration, it can roughly be pointed out that this structure of key would work successfully for a source file of 1 MB size.

#### **9.5.4.3 The 252-bit Secret Key**

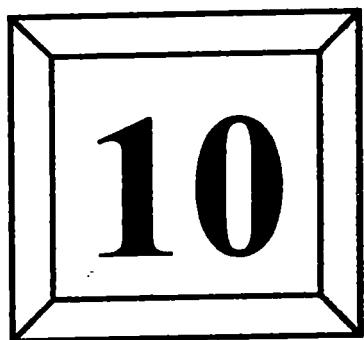
On the basis of the discussion in section 9.5.4.2, figure 9.5.4.3.1 shows the proposed format.



**Figure 9.5.4.3.1**  
**Proposed Format of 252-bit Secret Key with 35 Segments**

## 9.6 Conclusion

To make the process of encryption complicated and eventful, the role of cascading is inevitable, as it involves a series of operations, which plays a leading role in enhancing security. The correct implementation of the proposed integrated encryption system is expected to be an asset in the field of cryptography.



## **A Conclusive Discussion**

<u>Contents</u>	<u>Pages</u>
<b>10.1 Introduction</b>	<b>303</b>
<b>10.2 A Comparison of Different Implementations</b>	<b>303</b>
<b>10.3 A Conclusion on Proposed Implementation</b>	<b>310</b>

## **10.1 Introduction**

During the entire activity of developing the proposed encryption techniques, different techniques have been implemented in different ways. With comparing these implementations, it is not possible to compare the efficiencies of the techniques. Yet from the implementation point of view, performing this task of comparison has an effective significance. Section 10.2 attempts to perform this task.

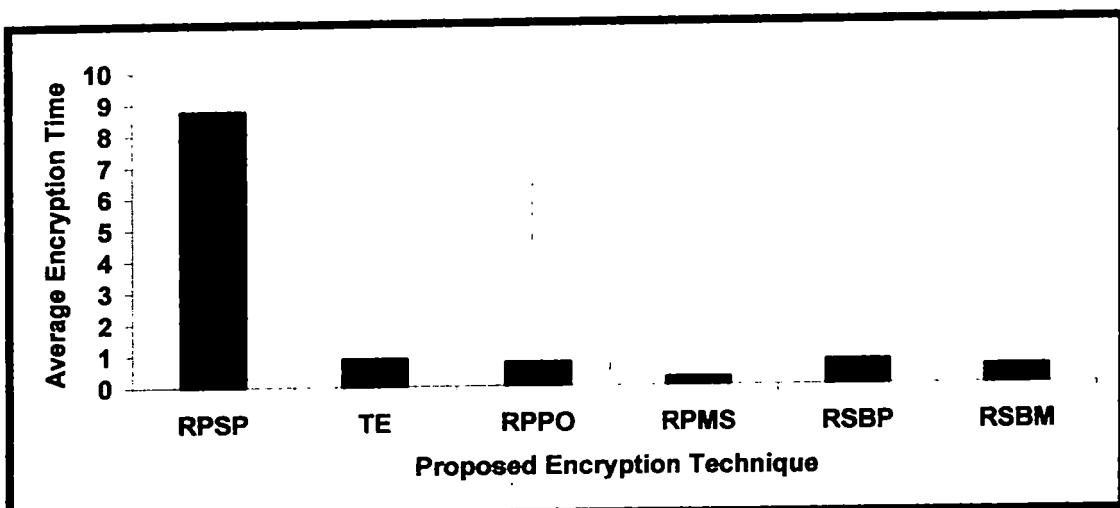
Apart from the real implementation, for each proposed technique, a model implementation is also proposed in the respective chapter and in chapter 8. One conclusion on this issue is drawn in section 10.3.

## **10.2 A Comparison among Different Implementations**

The policies adopted for the implementation of different techniques have been pointed out in the respective chapters. In this section, the comparison is done on the basis of the encryption time, the decryption time, and the chi square value. Since while implementing different proposed techniques, the same set of sample files have been considered, for each technique, the average encryption/decryption time and the chi square value have been computed, and on the basis of these results, the comparison has been performed. This entire activity has already been summarized in table 7.5.1 in chapter 7.

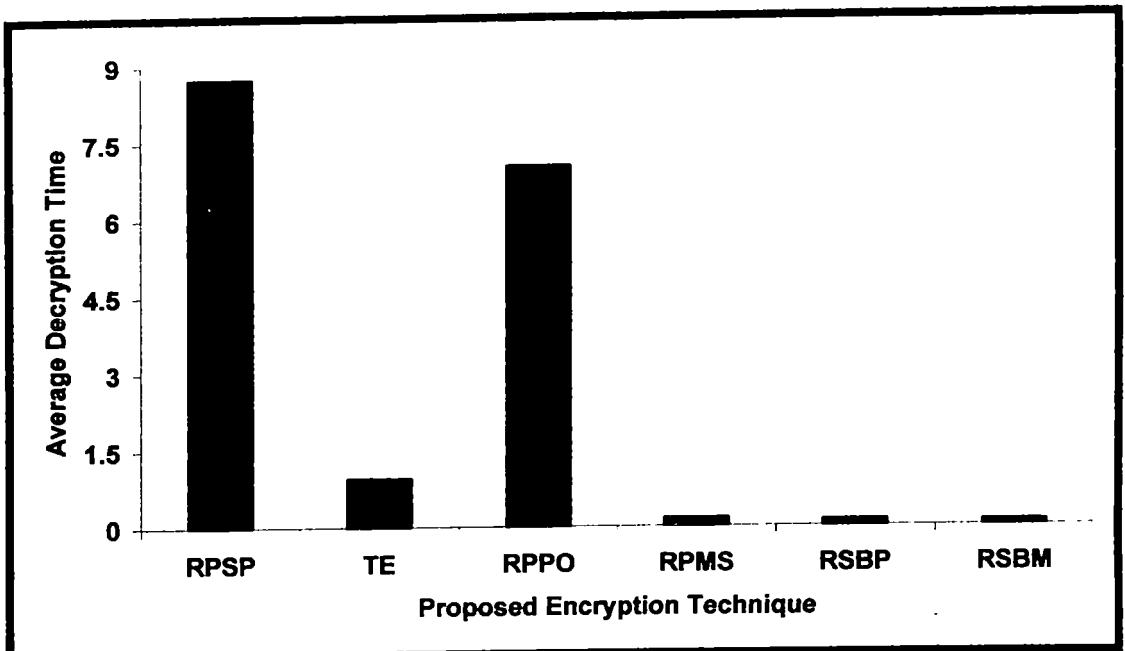
Using table 7.5.1, figure 10.2.1, figure 10.2.2, and figure 10.2.3 present diagrammatic comparisons respectively for the encryption time, the decryption time, and the chi square value.

In figure 10.2.1, there exist six vertical pillars standing for six proposed techniques. On the basis of the average encryption times required during implementing the different techniques for all fifty sample files, heights of the pillars have been settled. The left-most pillar stands for the RPSP technique, for which the average encryption time is 8.75713800 seconds. Followed by this, along the left-to-right direction, the remaining pillars respectively stand for the techniques of TE, RPPO, RPMS, RSBP, and RSBM, with average encryption times being 0.86703290 seconds, 0.73186806 seconds, 0.23659592 seconds, 0.74673470 seconds, and 0.55959400 seconds.



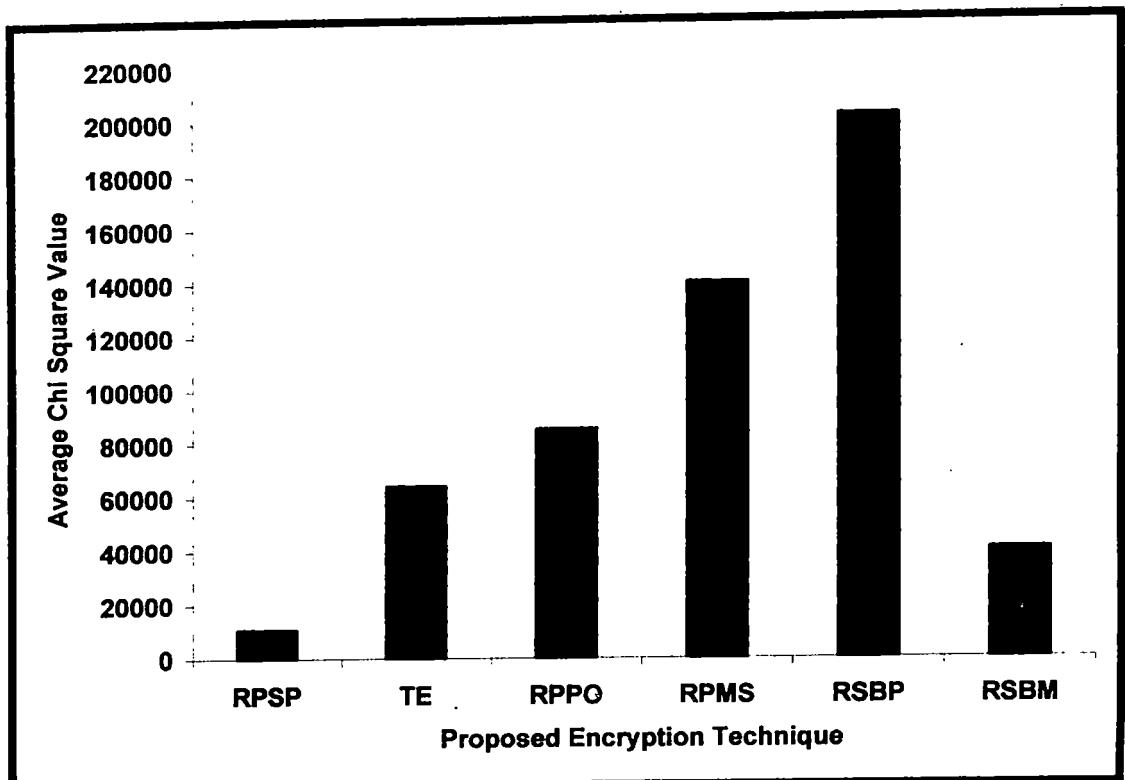
**Figure 10.2.1**  
**Comparison of Average Encryption Times for**  
**Different Proposed Techniques**

In figure 10.2.2, there exist six vertical pillars standing for six proposed techniques. On the basis of the average decryption times required during implementing the different techniques for all fifty sample files, heights of the pillars have been settled. The left-most pillar stands for the RPSP technique, for which the average decryption time is 8.73955200 seconds. Followed by this, along the left-to-right direction, the remaining pillars respectively stand for the techniques of TE, RPPO, RPMS, RSBP, and RSBM, with average decryption times being 0.94175818 seconds, 7.03076904 seconds, 0.15137143 seconds, 0.11040816 seconds, and 0.09020000 seconds.



**Figure 10.2.2**  
**Comparison of Average Decryption Times for**  
**Different Proposed Techniques**

In figure 10.2.3, there exist six vertical pillars standing for six proposed techniques. On the basis of the average Chi Square values obtained after implementing the different techniques for all fifty sample files, heights of the pillars have been settled. The left-most pillar stands for the RPSP technique, for which the average Chi Square value is 10701.70. Followed by this, along the left-to-right direction, the remaining pillars respectively stand for the techniques of TE, RPPO, RPMS, RSBP, and RSBM, with average values being 64188.04, 85350.94, 140196.94, 201990.76, and 40581.68.



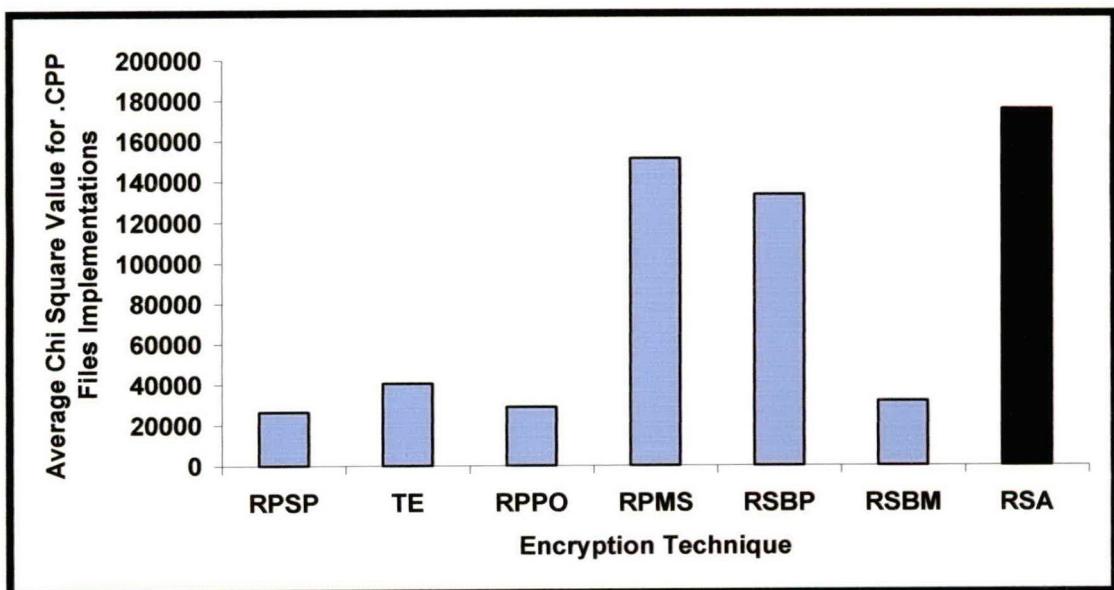
**Figure 10.2.3**  
**Comparison of Average Chi Square Values obtained for  
 Different Proposed Techniques**

To compare implementations of all proposed techniques with that of the RSA technique, the average chi square value has been computed for implementations of all CPP sample source files, and the result is enlisted in table 10.2.1. Here the observed value for the existing RSA technique is 175993.4. The values that are most closely compatible with this value are 151357.8, which is obtained for the RPMS technique, and 133624.5, which is obtained for the RSBP technique. Values obtained for the other proposed techniques of RPSP (26511.1), TE (40427.4), RPPO (28902.4), and RSBM (31781.9) are good enough to conclude that files are heterogeneous in nature with only 1% uncertainty.

**Table 10.2.1**  
**Average Chi Square Value for Different Proposed Techniques and Existing RSA Technique implementing .CPP Files**

	Proposed RPSP Technique	Proposed TE Technique	Proposed RPPO Technique	Proposed RPMS Technique	Proposed RSBP Technique	Proposed RSBM Technique	Existing RSA Technique
<b>Average Chi Square Value</b>	26511.1	40427.4	28902.4	151357.8	133624.5	31781.9	175993.4

The corresponding graphical relationship is shown in figure 10.2.4. Vertical pillars of color blue stand for results corresponding to different proposed techniques mentioned in the figure, whereas the black pillar is corresponding to the result for the RSA technique.



**Figure 10.2.4**  
**Graphical Relationship of Average Chi Square Value for Different Proposed Techniques and Existing RSA Technique implementing .CPP Files**

Table 10.2.2 points out those instances where results in terms of the Chi Square values are observed to be better than the corresponding results using the existing RSA technique. It is observed that results for PROJECT.CPP, START.CPP, CHARTCOM.CPP, and MAINC.CPP are better at the implementation through the RPSP

and the RSBP techniques. In case of the encryption through the TE technique, the result is found better for MAINC.CPP. The same is true for the RPPO and the RSBM implementations also. By implementing the RPMS technique for PROJECT.CPP, START.CPP, and MAINC.CPP Chi Square results are observed to be better than those using RSA implementation.

**Table 10.2.2**  
**List of Files generating Better Result in Proposed Technique than Existing RSA Technique**

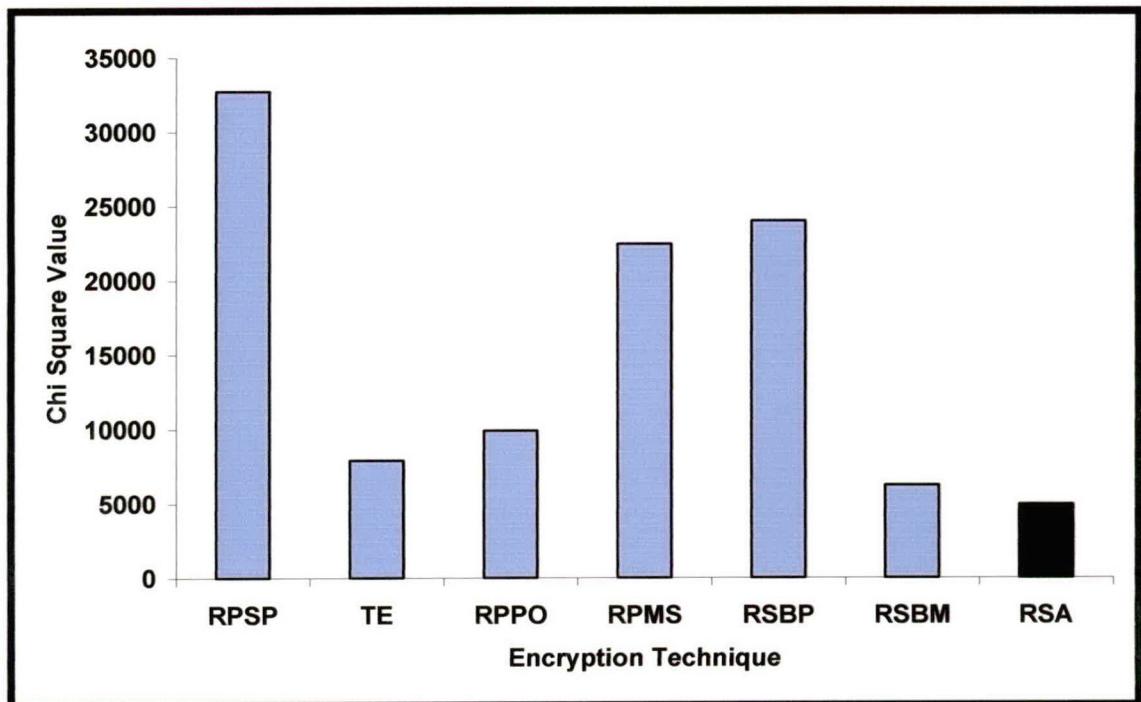
<b>Proposed Technique</b>	<b>Source File(s) with Better Results</b>
<b>RPSP</b>	<b>PROJECT.CPP START.CPP CHARTCOM.CPP MAINC.CPP</b>
<b>TE</b>	<b>MAINC.CPP</b>
<b>RPPO</b>	<b>MAINC.CPP</b>
<b>RPMS</b>	<b>PROJECT.CPP START.CPP MAINC.CPP</b>
<b>RSBP</b>	<b>PROJECT.CPP START.CPP CHARTCOM.CPP MAINC.CPP</b>
<b>RSBM</b>	<b>MAINC.CPP</b>

As per table 10.2.2, the file for which the result of implementation in terms of the chi square value is observed better for all proposed techniques than the existing RSA technique is MAINC.CPP. Table 10.2.3 enlists chi square values obtained after encrypting MAINC.CPP using all proposed techniques and the RSA technique. The Chi Square value of 4964 is generated through the implementation of the RSA technique, whereas all the proposed techniques produce higher values. These higher values include 32724 (for the RPSP technique), 7916 (for the TE technique), 9920 (for RPPO technique), 22485 (for the RPMS technique), 24048 (for the RSBP technique), and 6245 (for RSBM technique).

**Table 10.2.3**  
**Comparison of Chi Square Values obtained encrypting MAINC.CPP using All Proposed Techniques and Existing RSA Technique**

	Proposed RPSP Technique	Proposed TE Technique	Proposed RPPO Technique	Proposed RPMS Technique	Proposed RSBP Technique	Proposed RSBM Technique	Existing RSA Technique
Chi Square Value	32724	7916	9920	22485	24048	6245	4964

Graphically this comparison is shown in figure 10.2.5. Here also blue vertical pillars stand for results for MAINC.CPP after implementations through different proposed techniques mentioned in the figure, whereas the black pillar stands for the same after the implementation through the RSA technique.



**Figure 10.2.5**  
**Graphical Comparison of Chi Square Values obtained encrypting MAINC.CPP using All Proposed Techniques and Existing RSA Technique**

On the basis of the entire process of implementations and comparison, it is not a wise point to perform the final evaluation of different proposed schemes. All techniques being block ciphers, it is the issue of decomposition of the source stream of bits that also

plays a vital role in enhancing the security. Evaluation through the chi square value is only one accepted model for the purpose of evaluation. But the real strength lies in the proposed key structures, the concluding remark on which is made in section 10.3.

### **10.3 Conclusion on Different Model Implementations**

On the basis of the entire activity of development and simple implementation of different proposed encryption policies, and comparison of these implementations with the well-accepted RSA system of encryption, the following steady conclusion can be drawn:

**If each of the proposed encryption techniques is implemented independently following the protocol of the model implementation, presented in chapter 8; or if the proposed techniques are implemented in the cascaded manner following the protocol of the model implementation, presented in chapter 9, perfect, computationally secure cipher files can be generated; and on the basis of all possible kinds of factors normally used for evaluation, this can be proved to be well-compatible with the existing encryption systems.**

# **Appendix A**

## **References**

1. William Stallings, "Cryptography and Network Security", Third Edition, Pearson Education.
2. Bruce Schneier, *Applied Cryptograph*. Second Edition, Protocols, Algorithms, and Source Code in C, John Wiley & Sons Inc., 1996.
3. L.M. Adleman, C. Pomerance and R. S. Rumely, "On Distinguishing Prime Numbers from Composite Numbers", Annals of Mathematics, v. 117, n. 1, 1983, pp. 173-206.
4. H. Fiestel, "Cryptography and Computer Privacy," Scientific American, v. 228, n.5, May 1973, pp. 15-23.
5. G. B. Agnew, "Random Sources of Cryptographic Systems," Advances in Cryptology: – EURCRYPT '87 Proceedings, Springer Verlag, 1988, pp. 77-81.
6. S. G. AK1 and H. Meijer, "A Fast Pseudo-Random Permutation Generator with Applications to Cryptology," Advances in Cryptology: – EURCRYPT '84 Proceedings, Springer Verlag, 1985, pp. 269-275.
7. W. Alexi, B.Z. Chor, O. Goldreich and C. P. Schnorr. "RSA and Rabin Functions: Certain Parts are as Hard as the Whole", SIAM Journal on Computing, v. 17, n.2, Apr 1988, pp. 194 -209.
8. W. Alexi, B.Z. Chor, O. Goldreich and C. P. Schnorr, "RSA and Rabin Functions: Certain Parts are as Hard as the Whole" Proceedings of the 25<sup>th</sup> IEEE Symposium on the Foundation of Computer Science, 1984, pp. 449-457.
9. R. J. Anderson, "Why Cryptosystems Fail," 1<sup>st</sup> ACM Conference on Computer and Communications Security ACM Press, 1993, pp. 215-227.
10. R. J. Anderson, "Why Cryptosystems Fail," Communications of the ACM, v.37, n. 11, Nov 1994, pp. 32-40.
11. J. Anderson, and R. Needham, "Robustness of Principles for Public Key Protocols," Advances in Cryptology: – EURCRYPT '95 Proceedings, Springer Verlag, 1995.
12. C. Asmuth and J. Bloom, "A Modular Approach to Key Safeguarding," IEEE Transactions on Information Theory, v.IT 29, n. 2. Mar 1983, pp. 208-210.
13. S. K. Banerjee, "High Speed Implementation of DES," Computers & Security, v.1, 1982, pp. 261-267.

14. E. Biham and A. Biryukov, "How to Strengthen DES Using Existing Hardware," *Advances in Cryptology – ASIACRYPT '94 Proceedings*, Springer-Verlag, 1995.
15. E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Crytosystems," *Advances in Cryptology – CRYPTO '90 Proceedings*, Springer-Verlag, 1991, pp. 2-21.
16. E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Crytosystems," *Journal of Cryptology*, v. 4, n. 1, 1991, pp. 3-72.
17. E. Biham and A. Shamir, "Differential Cryptanalysis of the Full 16-Round DES," *Advances in Cryptology – CRYPTO '92 Proceedings*, Springer-Verlag, 1993, pp. 487-496.
18. G. Brassard, "A Note on Complexity of Cryptography," *IEEE Transactions on Information Theory*, v. IT 25, n. 2, Mar 1979, pp. 232-233.
19. G. Brassard, "Relativized Cryptography," *Proceedings of the IEEE 20<sup>th</sup> Annual Symposium on the Foundations of Computer Science*, 1979, pp.383-391.
20. G. Brassard, "Relativized Cryptography," *IEEE Transactions on Information Theory*, v. IT 29, n. 6, Nov 1983, pp. 877-894.
21. E. F. Brickell, and A.M. Odlyzko, "Cryptanalysis: A Survey of Recent Results," *Proceedings of the IEEE* v. 76, n. 5, May 1988. pp. 578-593.
22. E.F. Brikell, and A.M. Odlyzko, "Cryptanalysis: A Survey of Recent Results," *Contemporary Cryptology: The Science of Information Integrity*, G.J. Simmons, ed., IEEE Press, 1991, pp. 501-540.
23. T.R. Cain, and A.T. Sherman, "How to Break Gifford's Cipher," *Proceedings of 2<sup>nd</sup> Annual ACM Conferences on Computer and Communication Security*, ACM Press, 1994, pp. 198-209.
24. J.M. Carroll, "Do-it Yourself Cryptography," *Computers and Security*, v. 9, n.7, Nov 1990, pp. 613-619.
25. C. Connell, "An Analysis of New-DES: A Modified version of DES," *Cryptologia*, v. 14, n. 3, July 1990, pp. 217-223.
26. R.H. Cooper, and W. Patterson, "A generalization of Knapsack Method Using Galois Fields," *Cryptologia*, v. 8, n. 4, Oct 1984, pp. 343-347.
27. D. Coppersmith, "Data Encryption Standard (DES) and its Strength against Attacks," *Technical Report RC 18613*, IBM T.J. Watson Center, Dec 1992.

28. D. Coppersmith, "Data Encryption Standard (DES) and its Strength against Attacks," IBM Journal of Research and Development, v. 38, n. 3, May 1994, pp. 243-250.
29. C. Cauvreur and J.J. Quisquater, "An Introduction to Fast Generation of Large Prime Numbers," Philips Journal Research, v. 37, n. 5-6, 1982, pp. 231-264.
30. C. Cauvreur and J.J. Quisquater, "An Introduction to Fast Generation of Large Prime Numbers," Philips Journal Research, v. 38, 1983, pp. 77.
31. J. Daemen, and J. Vandewalle, "Block Cipher Based on Modular Arithmetic," Proceedings of Third Symposium on State and Progress of Research in Cryptography, Rome Italy, 15-16 Feb 1993, pp. 80-89.
32. J. Daemen, R. Govaerts, and J. Vandewalle, "A New Approach to Block Cipher Design," Fast Software Encryption, Cambridge Security Workshop Proceedings, Springer-Verlag, 1994, pp. 18-32.
33. D. E. Denning, "Data Encryption Standard: Fifteen Years of Public Scrutiny", Proceedings Sixth Annual Computer Security Applications Conference, IEEE Computer Society Press, 1990.
34. A. Di Porto and W. Wolfowicz, "VINO: A Block Cipher Including Variable Permutations," Fast Software Encryption Cambridge Security Workshop Proceedings, Springer-Verlag, 1994, pp. 205-210.
35. S. Even and O. Goldreich, "DES-Like Functions Can Generate the Alternating Group," IEEE Transactions on Information Theory, v. IT-29. n. 6, Nov 1983, pp. 863-865.
36. S. Even and O. Goldreich, "On the Power of Cascade Ciphers," ACM Transactions on Computer Systems, v. 3, n. 2, May 1985, pp. 108-116.
37. H. Gustafson, E. Dawson and B. Caelli, "Comparison of Block Ciphers," Advances in Cryptology – AUSCRYPT '90 Proceedings, Springer-Verlag 1990, pp. 208-220.
38. X. Lai and J. Massey, "A Proposal for New Block Encryption Standard," Advances in Cryptology – EUROCRYPT '90 Proceedings, Springer-Verlag, 1991, pp. 389-404.
39. Y. Ohnishi "A Study of Data Security," Master's Thesis Tohoku University, Japan 1988.
40. R.L. Rivest, "A Description of a Single-Chip Implementation of RSA Cipher," LAMDA Magazine, v.1, n. 3, 1980, pp. 14- 18.

41. M. J. B. Robshaw, "Block Ciphers" Technical Report TR-601, RSA Laboratories, Jul 1994.
42. B. Schneier, "One-Way Hash Functions," Dr. Dobb's Journal, v.16, n. 9, Sep 1991, pp. 148-151.
43. B. Schneier, "Description of a New Variable-length Key, 64-bit Block Cipher (Blowfish)," Fast software Encryption, Cambridge Security Workshop Proceedings, Springer-Verlag, 1994, pp. 191-204.
44. N. G. Das, "Statistical Methods in Commerce, Accountancy & Economics (Part II)", M. Das & Co.
45. Mandal J. K. and Dutta S., "A Space-Efficient Universal Encoder for Secured Transmission", International Conference on Modelling and Simulation (MS' 2000-Egypt), Cairo, April 11-14, 2000, pp-193-201.
46. Mandal J. K. and Dutta S., "A Universal Encryption Technique", Proceedings of the National Conference of Networking of Machines, Microprocessors, IT and HRD-Need of the Nation in the Next Millennium, Kalyani-741 235, Dist. Nadia, West Bengal, India, November 25-26,1999, pp-B114-B120.
47. Mandal J.K. and Dutta S., "A Universal Bit-Level Encryption Technique", Proceedings of the 7th State Science and Technology Congress, Jadavpur University, West Bengal, India, February 28 - March 1, 2000, pp-INFO2.
48. Dutta S., Mandal J. K., Mal S., "A Multiplexing Triangular Encryption Technique – A move towards enhancing security in E-Commerce", Proceedings of IT Conference (organized by Computer Association of Nepal), 26 and 27 January, 2002, BICC, Kathmandu.
49. Dutta S., Mandal J. K., A Microprocessor Based Cascaded Technique of Encryption, Proceedings of XXXVI Annual Convention, CSI 2001, Science City, Kolkata, November 20-24, 2001, pp. C269-275
50. J K Mandal, S Mal and S Dutta, " Security in E-Business – A Strategic Issue", National Seminar on Emerging Issues and Strategic Options Before Business in the Liberalized Regime", 7<sup>th</sup> March, 2001, pp 5-6
51. J K Mandal, S Mal and S Dutta, "Aspects of Storage Efficient Security in GIS Data" Workshop on Remote Sensing and GIS for Sustainable Development and Management in the Himalayas and Adjoining Areas" at NBU, West Bengal by Indian Society of Remote Sensing, Kolkata, Chapter, March 8-9, 2002, pp-24

52. S Mal, J K Mandal and S Dutta, "A Microprocessor Based Encoder for Secured Transmission" Conference on Intelligent Computing on VLSI, Kalyani Govt. Engg. College, 1-17 Feb, 2001, pp 164-169
53. S Mal, J K Mandal and S Dutta, "A Cryptographic Model for Secured Transmission of Messages", Proceedings of the National Conference on Applicable Mathematics, WMVC-2001, A.C. College of Commerce, Jalpaiguri, March 17-19, 2001, pp-18-21.
54. S. Mal, J K Mandal and S Dutta, "A Microprocessor Based Generalized Recursive Pair Parity Encoder for Secured Transmission", J. Tech., July 2003, Vol. XXXVII, Nos. 1-2, pp. 11-20
55. Mandal J. K., Mal S., Dutta S., A 256 Bit Recursive Pair Parity Encoder for Encryption, accepted for publication in AMSE Journal, France, 2003
56. Michael Welschenbach, "Cryptography in C and C++", APRESS

## **Appendix B**

### **List of Publications**

1. Dutta S. and Mandal J. K., "A Space-Efficient Universal Encoder for Secured Transmission", International Conference on Modelling and Simulation (MS' 2000-Egypt), Cairo, April 11-14, 2000
2. Mandal J. K., Mal S., Dutta S., A 256 Bit Recursive Pair Parity Encoder for Encryption, accepted for publication in AMSE Journal, France, 2003
3. Dutta S., Mandal J. K., Mal S., "A Multiplexing Triangular Encryption Technique – A move towards enhancing security in E-Commerce", Proceedings of IT Conference (organized by Computer Association of Nepal), 26 and 27 January, 2002, BICC, Kathmandu
4. Dutta S. and Mandal J. K., "A Universal Encryption Technique", Proceedings of the National Conference of Networking of Machines, Microprocessors, IT and HRD-Need of the Nation in the Next Millennium, Kalyani-741 235, Dist. Nadia, West Bengal, India, November 25-26, 1999
5. Dutta S. and Mandal J. K., "A Universal Bit-Level Encryption Technique", Proceedings of the 7th State Science and Technology Congress, Jadavpur University, West Bengal, India, February 28 - March 1, 2000
6. Dutta S., Mandal J. K., A Microprocessor Based Cascaded Technique of Encryption, Proceedings of XXXVI Annual Convention, CSI 2001, Science City, Kolkata, November 20-24, 2001, pp. C269-275
7. J K Mandal, S Mal and S Dutta, " Security in E-Business – A Strategic Issue", National Seminar on Emerging Issues and Strategic Options Before Business in the Liberalized Regime", 7<sup>th</sup> March, 2001, pp 5-6
8. J K Mandal, S Mal and S Dutta, "Aspects of Storage Efficient Security in GIS Data" Workshop on Remote Sensing and GIS for Sustainable Development and Management in the Himalayas and Adjoining Areas" at NBU, West Bengal by Indian Society of Remote Sensing, Kolkata, Chapter, March 8-9, 2002, pp-24
9. S Mal, J K Mandal and S Dutta, "A Microprocessor Based Encoder for Secured Transmission" Conference on Intelligent Computing on VLSI, Kalyani Govt. Engg. College, 1-17 Feb, 2001, pp 164-169

10. **S Mal, J K Mandal and S Dutta**, “A Cryptographic Model for Secured Transmission of Messages”, Proceedings of the National Conference on Applicable Mathematics, WMVC-2001, A.C. College of Commerce, Jalpaiguri, March 17-19, 2001, pp-18-21
11. **S. Mal, J K Mandal and S Dutta**, “A Microprocessor Based Generalized Recursive Pair Parity Encoder for Secured Transmission”, J. Tech., July 2003, Vol. XXXVII, Nos. 1-2, pp. 11-20

# **Appendix C**

## **Listing of Source Codes**

## Contents in Sequence

- **The RPSP Encryption Code**
- **The RPSP Decryption Code**
- **The TE Encryption Code**
- **The TE Decryption Code**
- **The RPPO Encryption Code**
- **The RPPO Decryption Code**
- **The RPMS Encryption Code**
- **The RPMS Decryption Code**
- **The RSBP Encryption Code**
- **The RSBP Decryption Code**
- **The RSBM Encryption Code**
- **The RSBM Decryption Code**
- **The Code to Calculate Pearsonian Chi Square Value**
- **The Code to Test Frequency Distribution**

```

//THE RPSP ENCRYPTION CODE
#include<stdio.h>
#include<math.h>
#include<time.h>
#include<string.h>

// Function for converting binary to corresponding ascii value.
int equiasci(int tar[])
{
    int asc=0;
    int i;
    for(i=0;i<=7;i++)
        asc+=tar[i] * pow(2,(7-i));
    return asc;
}

// Function for converting a character read to its equivalent binary.
void binary(char ch,int mid[])
{
    int i,mask,k;
    for(i=7;i>=0;i--)
    {
        mask=1<<i; // left shift 1 i times.
        k=ch & mask; // Extract the ith bit of ch;
        mid[7-i]=(k==0)?0:1; // And store it in array mid for future use.
    }
}

//Function will take an integer value as argument, then it will test whether
//it is prime or not. If prime return true otherwise false.
int prime(int num)
{
    int d;
    if(num==1)
        return(0);
    else
    {
        for(d=2;d<=sqrt(num);d++)
        {
            if(num%d==0)
                return(0);
        }
        return(1);
    }
}

```

```

//This function will take index value of mid array, and it will calculate
//the index of tar array where the bit will assign.
int tarindex(int mindex,int nb)
{
    int temp,index;
    if(mindex==nb)
        return(mindex);

    temp=nb-mindex;
    if(prime(temp))
    {
        if(temp==2)
            temp=nb+1;
        for(index=temp-1;index>=2;index--)
        {
            if(prime(index))
                return(index);
        }
    }
    return(temp);
}

// Number of iteration required to come back to original
// for block size nb
long int iteration(int nb)
{
    int *s,*t;
    int i,mi;
    long int itr;
    static int flag;
    s=(int *)calloc(sizeof(int),nb);
    t=(int *)calloc(sizeof(int),nb);
    for(i=0;i<nb;i++)
        s[i]=i;
    itr=0;
    flag=0;
    while(!flag)
    {
        for(i=0;i<nb;i++)
        {
            mi=tarindex(i+1,nb);
            t[mi-1]=s[i];
        }
        itr++;
        i=0;
        flag=1;
    }
}

```

```

        while(i<nb&&flag)
        {
            if(t[i]!=i)
                flag=0;
            i++;
        }
        for(i=0;i<nb;i++)
            s[i]=t[i];
    }
    return(itr);
}

// nb bits in s[]. After one iteration result in t[];
void one_primeop(char s[],char t[],int nb)
{
    int i,mi;
    for(i=0;i<nb;i++)
    {
        mi=tarindex(i+1,nb);
        t[mi-1]=s[i];
    }
}

//operate n times on the bit array
void nprimeop(char s[],char t[],int nb,long int n)
{
    int i,mi,itr;
    itr=0;
    while(itr<n)
    {
        one_primeop(s,t,nb);
        for(i=0;i<nb;i++)
            s[i]=t[i];
        itr++;
    }
}

void pimeoponfile(char sfile[],char tfile[],int nb,long int itr)
{
    FILE *fs,*ft;
    char *t_array, *s_array;
    int i,j,k,nbby8;
    long int it;
    int temp[8],asc,rem;
}

```

```

unsigned long int size,count,dec,outloop;
char ch;
fs=fopen(sfile,"rb"); //Open the source file.
ft=fopen(tfile,"wb"); // Open the target file.
if(fs==NULL || ft==NULL)
{
    printf("\n File Opening Error. ");
    exit();
}

fseek(fs,0,SEEK_END);
size=f.tell(fs);
fseek(fs,0,SEEK_SET);
s_array=(char *)calloc(nb,sizeof(char));
t_array=(char *)calloc(nb,sizeof(char));
nbby8=nb/8;
outloop=size/nbby8;
rem=size%nbby8;

for(count=0;count<outloop;count++)
{
    k=0;
    for(i=0;i<nbby8;i++)
    {
        ch=getc(fs); // Read each character from the source file.
        binary(ch,temp); // Call the funct. binary.
        for(j=0;j<8;j++)
            s_array[k++]=temp[j];
    }

    nprimeop(s_array,t_array,nb,itr);

    for(i=0,j=0;i<nb;i++)
    {
        *(temp+j++)=t_array[i];
        if(j%8==0)
        {
            j=0;
            asc=equiasci(temp); // Call the funct. equiasci
            // Write the corresponding character in the target file.
            putc(asc,ft);
        }
    }
}

for(count=0;count<rem;count++)

```

```

    {
        ch=getc(fs);
        putc(ch,ft);
    }
    fclose(fs);
    fclose(ft);
    free(s_array);
    free(t_array);

}

void main()
{
    int nb;
    int startblk,endblk;
    unsigned long int itr;
    char source[15],target[15];
    clock_t start,end;

    float time;

    printf("\n Enter Source file name :");
    scanf("%s",source);

    printf("\n Enter Target file name :");
    scanf("%s",target);

    printf("\n Enter stream size : ");
    scanf("%d",&nb);
    start=clock();
    itr=iteration(nb);
    itr=itr/2;
    primeonfile(source,target,nb, itr);
    end=clock();
    time=(end-start)/(float)CLK_TCK;

    printf("\nEncrypted file is \" %s \",target);
    //printf("\nFile size = %lu ",size);
    printf("\nTime taken for encryption = %.4f",time);
    getch();
}

```

## //THE RPSP DECRYPTION CODE

```
#include<stdio.h>
#include<math.h>
#include<time.h>
#include<string.h>

// Function for converting binary to corresponding ascii value.
int equiasci(int tar[])
{
    int asc=0;
    int i;
    for(i=0;i<=7;i++)
        asc+=tar[i] * pow(2,(7-i));
    return asc;
}

// Function for converting a character read to its equivalent binary.
void binary(char ch,int mid[])
{
    int i,mask,k;
    for(i=7;i>=0;i--)
    {
        mask=1<<i; // left shift 1 i times.
        k=ch & mask; // Extract the ith bit of ch;
        mid[7-i]=(k==0)?0:1; // And store it in array mid for future use.
    }
}

//Function will take an integer value as argument, then it will test whether
//it is prime or not. If prime return true otherwise false.
int prime(int num)
{
    int d;
    if(num==1)
        return(0);
    else
    {
        for(d=2;d<=sqrt(num);d++)
        {
            if(num%d==0)
                return(0);
        }
        return(1);
    }
}
```

```

}

//This function will take index value of mid array, and it will calculate
//the index of tar array where the bit will assign.
int tarindex(int mindex,int nb)
{
    int temp,index;
    if(mindex==nb)
        return(mindex);

    temp=nb-mindex;
    if(prime(temp))
    {
        if(temp==2)
            temp=nb+1;
        for(index=temp-1;index>=2;index--)
        {
            if(prime(index))
                return(index);
        }
    }
    return(temp);
}

// Number of iteration required to come back to original
// for block size nb
long int iteration(int nb)
{
    int *s,*t;
    int i,mi;
    long int itr;
    static int flag;
    s=(int *)calloc(sizeof(int),nb);
    t=(int *)calloc(sizeof(int),nb);
    for(i=0;i<nb;i++)
        s[i]=i;
    itr=0;
    flag=0;
    while(!flag)
    {
        for(i=0;i<nb;i++)
        {
            mi=tarindex(i+1,nb);
            t[mi-1]=s[i];
        }
        itr++;
        i=0;
    }
}

```

```

        flag=1;
        while(i<nb&&flag)
        {
            if(t[i]!=i)
                flag=0;
            i++;
        }
        for(i=0;i<nb;i++)
            s[i]=t[i];
    }
    return(itr);
}

// nb bits in s[]. After one iteration result in t[];
void one_primeop(char s[],char t[],int nb)
{
    int i,mi;
    for(i=0;i<nb;i++)
    {
        mi=tarindex(i+1,nb);
        t[mi-1]=s[i];
    }
}

//operate n times on the bit array
void nprimeop(char s[],char t[],int nb,long int n)
{
    int i,mi,itr;
    itr=0;
    while(itr<n)
    {
        one_primeop(s,t,nb);
        for(i=0;i<nb;i++)
            s[i]=t[i];
        itr++;
    }
}

void pimeoponfile(char sfile[],char tfile[],int nb,long int itr)
{
    FILE *fs,*ft;
    char *t_array, *s_array;
    int i,j,k,nbby8;
    long int it;
}

```

```

int temp[8],asc,rem;
unsigned long int size,count,dec,outloop;
char ch;
fs=fopen(sfile,"rb"); //Open the source file.
ft=fopen(tfile,"wb"); // Open the target file.
if(fs==NULL || ft==NULL)
{
    printf("\n File Opening Error. ");
    exit();
}

fseek(fs,0,SEEK_END);
size=ftell(fs);
fseek(fs,0,SEEK_SET);
s_array=(char *)calloc(nb,sizeof(char));
t_array=(char *)calloc(nb,sizeof(char));
nbby8=nb/8;
outloop=size/nbby8;
rem=size%nbby8;

for(count=0;count<outloop;count++)
{
    k=0;
    for(i=0;i<nbby8;i++)
    {
        ch=getc(fs); // Read each character from the source file.
        binary(ch,temp); // Call the funct. binary.
        for(j=0;j<8;j++)
            s_array[k++]=temp[j];
    }

    nprimeop(s_array,t_array,nb,itr);

    for(i=0,j=0;i<nb;i++)
    {
        *(temp+j++)=t_array[i];
        if(j%8==0)
        {
            j=0;
            asc=equiasci(temp); // Call the funct. equiasci
        }
        // Write the corresponding character in the target file.
        putc(asc,ft);
    }
}

```

```

for(count=0;count<rem;count++)
{
    ch=getc(fs);
    putc(ch,ft);
}
fclose(fs);
fclose(ft);
free(s_array);
free(t_array);

}

void main()
{
    int nb;
    int startblk,endblk;
    unsigned long int itr;
    char source[15],target[15];
    clock_t start,end;

    float time;

    printf("\n Enter Source file name :");
    scanf("%s",source);

    printf("\n Enter Target file name :");
    scanf("%s",target);

    printf("\n Enter stream size :");
    scanf("%d",&nb);
    start=clock();
    itr=iteration(nb);
    itr -=itr/2;
    primeoponfile(source,target,nb, itr);
    end=clock();
    time=(end-start)/(float)CLK_TCK;

    printf("\nDecrypted file is : %s \",target);
    //printf("\nFile size = %lu ",size);
    printf("\nTime taken for decryption = %.4f",time);
    getch();

}

```

```

//THE TE ENCRYPTION CODE
#include<stdio.h>
#include<math.h>
#include<time.h>
#include<string.h>

// Function for converting binary to corresponding ascii value.
int equiasci(int tar[])
{
    int asc=0;
    int i;
    for(i=0;i<=7;i++)
        asc+=tar[i] * pow((double)2,(double)(7-i));
    return asc;
}

// Function for converting a character read to its equivalent binary.
void binary(char ch,int mid[])
{
    int i,mask,k;
    for(i=7;i>=0;i--)
    {
        mask=1<<i; // left shift 1 i times.
        k=ch & mask; // Extract the ith bit of ch;
        mid[7-i]=(k==0)?0:1; // And store it in array mid for future use.
    }
}

//triangle-encription function
long float tri_enc(int s_array[],int t_array[],int nb)
{
    int i,j;
    long float nop;

    // binary(ch,s_array);
    for(i=0;i<nb;i++)
    {
        t_array[nb-i-1]=s_array[0];
        for(j=0;j<nb-i-1;j++)
        {
            s_array[j]=s_array[j]^s_array[j+1];
        }
    }
}

```

```

        }
        nop=(long float)nb*(nb-1)/2;
        return nop;
    }

//triangle decrption function
long float tri_dec(int s_array[],int t_array[],int nb)
{
    int i,j,loop;
    long float nop;

    // binary(ch,s_array);
    for(i=0;i<nb;i++)
    {
        t_array[i]=s_array[nb-i-1];
        loop=nb-i-1;
        for(j=0;j<loop;j++)
        {
            s_array[j]=s_array[j]^s_array[j+1];
        }

    }
    nop=(long float)nb*(nb-1)/2;
    return nop;
}

long float tri_enco(char source[],char dest[], int nb)
{
    FILE *fs,*ft;
    int *t_array, *s_array,i,asc;
    int temp[8];
    long int size,count,rem;
    int nbby8,j,k;
    static long float nop,totalop;
    char ch;

    fs=fopen(source,"rb"); //Open the source file.
    ft=fopen(dest,"wb"); // Open the target file.
    if(fs==NULL || ft==NULL)
    {
        printf("\n File Opening Error. ");
        exit();
    }

    fseek(fs,0,SEEK_END);
}

```

```

size=fteill(fs);
fseek(fs,0,SEEK_SET);
s_array=(int *)calloc(nb,sizeof(int));
t_array=(int *)calloc(nb,sizeof(int));
nbby8=nb/8;
rem=size/(nbby8);
for(count=0;count<rem;count++)
{
//This loop reads nb/8 characters or nb bits from source and store it in mid
//array.
    for(i=0,j=0;i<nbby8;i++)
    {
        ch=getc(fs);// Read each character from the source file.
        binary(ch,temp); // Call the funct. binary.
//mid array is used to store the binary value of all the characters read from
//file
        for(k=0;k<8;k++)
            *(s_array+j++)=temp[k];
    }
    nop=tri_enc(s_array,t_array,nb);
    totalop += nop;
//temp array is used to store 8-bits at a time, find thier equivalent ascii
// and write it to target file.
    for(i=0,j=0;i<nb;i++)
    {
        *(temp+j++)=t_array[i];
        if(j%8==0)
        {
            j=0;
            asc=equiasci(temp); // Call the funct. equiasci
// Write the corresponding character in the target file.
            putc(asc,ft);
        }
    }
}
// Calculate the remaining characters
rem=size%nbby8;
for(count=0;count<rem;count++)
{
    ch=getc(fs);
    putc(ch,ft);
}
fclose(fs);
fclose(ft);
free(s_array);

```

```

        free(t_array);

        return totalop;
    }
long float tri_deco(char source[],char dest[],int nb)
{
    FILE *fs,*ft;
    int *t_array, *s_array,i,asc;
    int temp[8];
    long int size,count,rem;
    int nbby8,j,k;
    static long float nop,totalop;
    char ch;
    fs=fopen(source,"rb"); //Open the source file.
    ft=fopen(dest,"wb"); // Open the target file.
    if(fs==NULL || ft==NULL)
    {
        printf("\n File Opening Error. ");
        exit();
    }

    fseek(fs,0,SEEK_END);
    size=f.tell(fs);
    fseek(fs,0,SEEK_SET);
    s_array=(int *)calloc(nb,sizeof(int));
    t_array=(int *)calloc(nb,sizeof(int));
    nbby8=nb/8;
    rem=size/nbby8;
    for(count=0;count<rem;count++)
    {

//This loop reads nb/8 characters or nb bits from source and store it in mid
//array.
        for(i=0,j=0;i<nbby8;i++)
        {
            ch=getc(fs);// Read each character from the source file.
            binary(ch,temp); // Call the funct. binary.
//mid array is used to store the binary value of all the characters read from
//file
            for(k=0;k<8;k++)
                *(s_array+j++)=temp[k];
        }
        nop=tri_dec(s_array,t_array,nb);
        totalop += nop;
//temp array is used to store 8-bits at a time, find thier equivalent ascii
    }
}

```

```

// and write it to target file.
    for(i=0,j=0;i<nb;i++)
    {
        *(temp+j++)=t_array[i];
        if(j%8==0)
        {
            j=0;
            asc=equiasci(temp); // Call the funct. equiasci
        }
        // Write the corresponding character in the target file.
        putc(asc,ft);
    }
}

// Calculate the remaining characters
rem=size%nbby8;
for(count=0;count<rem;count++)
{
    ch=getc(fs);
    putc(ch,ft);
}
fclose(fs);
fclose(ft);
free(s_array);
free(t_array);

return totalop;
}

// Define a structure for writing info. about the src file,coded file,decoded
// file, their file size, coding-decoding time, in another file.
struct tablerec
{
    char fsource[25];
    char fmtdtar[15];
    // char fttarget[15];
    long int ssize;
    // long int msize;
    // long int tsize;
    float entime;
    float detime;
    long float enoprt;
    long float deoprt;
    // int bsize;
};

void main()
{

```

```

int ans=1,nb;
static float nop,totalop;
int startblk,endblk;
char source[15],midtar[15],target[15];
// char tempfile[]="temp.txt"; //These files are used for temporarily
char source1[]{"source1.txt"};//holding the mid-term files generated
// char source2[]{"source2.txt"}// in both the phases.
FILE *fs,*ft,*fm,*fp;
clock_t start,end;

float time;
struct tablerec rec;

while(ans)
{
    printf("\n Enter Source file name :");
    scanf("%s",source);
    strcpy(rec.fsource,source);
    printf("\n Enter Middle target file name :");
    scanf("%s",midtar);
    strcpy(rec.fmidtar,midtar);
    // strcpy(rec.ftarget,target);
    //while(1)
    //{
        startblk=512;
        //printf("\nEnter starting block size : ");
        //scanf("%d",&startblk);
        endblk=512;
        //printf("\nEnter end block size : ");
        //scanf("%d",&endblk);

    // Encoding phase begins here. Start with 8-bit stream length and encode to
    // multiple of 8-bit stream length (like 16,32,64,...,256) for each character read
    // from the original file.

        nb=startblk;
        start=clock();
        totalop=tri_enco(source,source1,nb);
    /*
        for(nb=2*startblk;nb<=endblk;nb += nb)
        {
            nop=tri_enco(source1,source2,nb);
            totalop += nop;
            strcpy(tempfile,source1);//Swap the name of the low-bit
            strcpy(source1,source2);//& hi-bit file so that hi-bit
            strcpy(source2,tempfile);//file acts as the new source
        }
    */
}

```

```

*/
//Now encoded file is in source1.I want to store encoded file in midtar
//Hence I rename it.
    rename(source1,midtar);
    end=clock();
    time=(end-start)/(float)CLK_TCK;
// time in micro second
    rec.enoprt=totalop;
    rec.entime=time;

// Decoding phase begins here. Start with higher stream length and decode to
// 8-bit stream length for each character read from the encoded file to get
// the original source .
/*nop=0.0;
totalop=0.0;
start=clock();
nb=endblk;
totalop=tri_deco(midtar, source1,nb);*/
/*
for(nb=endblk/2;nb>=startblk;nb = nb/2)
{
    nop=tri_deco(source1,source2,nb);
    totalop += nop;
    strcpy(tempfile,source1);
    strcpy(source1,source2);
    strcpy(source2,tempfile);
}
*/
/*rename(source1,target);
end=clock();
time=(end-start)/(float)CLK_TCK;

rec.deoprt=totalop;
rec.detime=time;*/

fs=fopen(source,"rb");// Open the different files used in
// fm=fopen(midtar,"rb");// encoding & decoding phases to
// ft=fopen(target,"rb");// calculate their respective sizes.

// Open the info. file in binary append mode to write different information
// gathered about the files used in en-de coding phases.
//rec.bsize=startblk;
fp=fopen("data11.txt","a");
if(fp==NULL)
{
    printf("\n File Opening Error. ");
}

```

```

        exit();
    }

    fseek(fs,0,SEEK_END);
    rec.ssize=f.tell(fs);
    fseek(fs,0,SEEK_SET);
    printf("\n%ld",rec.ssize);
    // fseek(fm,0,SEEK_END); // different files used
    //rec.msize=f.tell(fm);

    // fseek(ft,0,SEEK_END); // and copy them to the rec.
    // rec.tsize=f.tell(ft);

    fwrite(&rec,sizeof(rec),1,fp); // Write to info. file
    fclose(fs);
    // fclose(fm);
    //fclose(ft);
    fclose(fp);
    printf("\n Do You Want To Continue ?(1/0)");
    scanf("%d",&ans);
}

/*
// printf("\n Number of Operation= %Lf ",noofop);
fp=fopen("data11.txt","rb"); // Open the info. file for reading the
printf("sFile eFile sSize tSize Comp-time Dcomp-time copr dopr \n");
while(fread(&rec,sizeof(rec),1,fp)==1) // different runs conducted.
{
    printf("\n%-12s %-8s ",rec.fsource,rec.fmtdtar); //rec.ftarget);
    printf("%ld %ld",rec.ssize,rec.tsize); //,rec.msize,rec.tsize);
    printf("%6f %6f ",rec.entime,rec.detime);
    printf("%0f %0f ",rec.enoprt,rec.deoprt);
}
fclose(fp);
*/
}

//THE TE DECRYPTION CODE
#include<stdio.h>
#include<math.h>
#include<time.h>
#include<string.h>

// Function for converting binary to corresponding ascii value.
int equiasci(int tar[])
{

```

```

int asc=0;
int i;
for(i=0;i<=7;i++)
    asc+=tar[i] * pow((double)2,(double)(7-i));
return asc;
}

// Function for converting a character read to its equivalent binary.
void binary(char ch,int mid[])
{
    int i,mask,k;
    for(i=7;i>=0;i--)
    {
        mask=1<<i; // left shift 1 i times.
        k=ch & mask; // Extract the ith bit of ch;
        mid[7-i]=(k==0)?0:1; // And store it in array mid for future use.
    }
}

//triangle-encription function
long float tri_enc(int s_array[],int t_array[],int nb)
{
    int i,j;
    long float nop;

    // binary(ch,s_array);
    for(i=0;i<nb;i++)
    {
        t_array[nb-i-1]=s_array[0];
        for(j=0;j<nb-i-1;j++)
        {
            s_array[j]=s_array[j]^s_array[j+1];
        }
    }
    nop=(long float)nb*(nb-1)/2;
    return nop;
}

//triangle decription function
long float tri_dec(int s_array[],int t_array[],int nb)
{
    int i,j,loop;

```

```

long float nop;

// binary(ch,s_array);
for(i=0;i<nb;i++)
{
    t_array[i]=s_array[nb-i-1];
    loop=nb-i-1;
    for(j=0;j<loop;j++)
    {
        s_array[j]=s_array[j]^s_array[j+1];
    }

}
nop=(long float)nb*(nb-1)/2;
return nop;
}

long float tri_enco(char source[],char dest[], int nb)
{
    FILE *fs,*ft;
    int *t_array, *s_array,i,asc;
    int temp[8];
    long int size,count,rem;
    int nbby8,j,k;
    static long float nop,totalop;
    char ch;
    fs=fopen(source,"rb"); //Open the source file.
    ft=fopen(dest,"wb"); // Open the target file.
    if(fs==NULL || ft==NULL)
    {
        printf("\n File Opening Error. ");
        exit();
    }

    fseek(fs,0,SEEK_END);
    size=f.tell(fs);
    fseek(fs,0,SEEK_SET);
    s_array=(int *)calloc(nb,sizeof(int));
    t_array=(int *)calloc(nb,sizeof(int));
    nbby8=nb/8;
    rem=size/(nbby8);
    for(count=0;count<rem;count++)
    {

//This loop reads nb/8 characters or nb bits from source and store it in mid

```

```

//array.
    for(i=0,j=0;i<nbby8;i++)
    {
        ch=getc(fs); // Read each character from the source file.
        binary(ch,temp); // Call the funct. binary.
    }
    //mid array is used to store the binary value of all the characters read from
    //file
    for(k=0;k<8;k++)
        *(s_array+j++)=temp[k];
    }
    nop=tri_enc(s_array,t_array,nb);
    totalop += nop;
    //temp array is used to store 8-bits at a time, find thier equivalent ascii
    // and write it to target file.
    for(i=0,j=0;i<nb;i++)
    {
        *(temp+j++)=t_array[i];
        if(j%8==0)
        {
            j=0;
            asc=equiasci(temp); // Call the funct. equiasci
            // Write the corresponding character in the target file.
            putc(asc,ft);
        }
    }
    // Calculate the remaining characters
    rem=size%nbby8;
    for(count=0;count<rem;count++)
    {
        ch=getc(fs);
        putc(ch,ft);
    }
    fclose(fs);
    fclose(ft);
    free(s_array);
    free(t_array);

    return totalop;
}
long float tri_deco(char source[],char dest[],int nb)
{
    FILE *fs,*ft;
    int *t_array, *s_array,i,asc;
    int temp[8];
    long int size,count,rem;

```

```

int nbby8,j,k;
static long float nop,totalop;
char ch;
fs=fopen(source,"rb"); //Open the source file.
ft=fopen(dest,"wb"); // Open the target file.
if(fs==NULL || ft==NULL)
{
    printf("\n File Opening Error. ");
    exit();
}

fseek(fs,0,SEEK_END);
size=f.tell(fs);
fseek(fs,0,SEEK_SET);
s_array=(int *)calloc(nb,sizeof(int));
t_array=(int *)calloc(nb,sizeof(int));
nbby8=nb/8;
rem=size/nbby8;
for(count=0;count<rem;count++)
{
    //This loop reads nb/8 characters or nb bits from source and store it in mid
    //array.
    for(i=0,j=0;i<nbby8;i--)
    {
        ch=getc(fs); // Read each character from the source file.
        binary(ch,temp); // Call the funct. binary.
    }
    //mid array is used to store the binary value of all the characters read from
    //file
    for(k=0;k<8;k++)
        *(s_array+j++)=temp[k];
    }
    nop=tri_dec(s_array,t_array,nb);
    totalop += nop;
    //temp array is used to store 8-bits at a time, find thier equivalent ascii
    // and write it to target file.
    for(i=0,j=0;i<nb;i++)
    {
        *(temp+j++)=t_array[i];
        if(j%8==0)
        {
            j=0;
            asc=equiasci(temp); // Call the funct. equiasci
        }
        // Write the corresponding character in the target file.
        putc(asc,ft);
    }
}

```

```

        }
    }

// Calculate the remaining characters
rem=size%nbby8;
for(count=0;count<rem;count++)
{
    ch=getc(fs);
    putc(ch,ft);
}
fclose(fs);
fclose(ft);
free(s_array);
free(t_array);

return totalop;
}

// Define a structure for writing info. about the src file,coded file,decoded
// file, their file size, coding-decoding time, in another file.
struct tablerec
{
    char fsouce[25];
    char fmtdtar[15];
//    char fttarget[15];
    long int ssize;
//    long int msize;
//    long int tsize;
    float entime;
    float detime;
    long float enoprt;
    long float deoprt;
//    int bsize;
};

void main()
{
    int ans=1,nb;
    static float nop,totalop;
    int startblk,endblk;
    char source[15],midtar[15],target[15];
//    char tempfile[]="temp.txt"; //These files are used for temporarily
    char source1[]="source1.txt";//holding the mid-term files generated
//    char source2[]="source2.txt";// in both the phases.
    FILE *fs,*ft,*fm,*fp;
    clock_t start,end;
}

```

```

float time;
struct tablerec rec;

while(ans)
{
    /*printf("\n Enter Source file name :");
    scanf("%s",source);
    strcpy(rec.fsource,source);*/
    printf("\n Enter Encrypted file name :");
    scanf("%s",midtar);
    strcpy(rec.fmidtar,midtar);
    printf("\n Enter Decrypted file name :");
    scanf("%s",target);
    // strcpy(rec.ftarget,target);
    //while(1)
    //{
    startblk=512;
    //printf("\nEnter starting block size : ");
    //scanf("%d",&startblk);
    endblk=512;
    //printf("\nEnter end block size : ");
    //scanf("%d",&endblk);

// Encoding phase begins here. Start with 8-bit stream length and encode to
// multiple of 8-bit stream length (like 16,32,64,...,256) for each character read
// from the original file.

/*nb=startblk;
start=clock();
totalop=tri_enco(source,source1,nb);*/
/*
for(nb=2*startblk;nb<=endblk;nb += nb)
{
    nop=tri_enco(source1,source2,nb);
    totalop += nop;
    strcpy(tempfile,source1);//Swap the name of the lw-bit
    strcpy(source1,source2);//& hi-bit file so that hi-bit
    strcpy(source2,tempfile);//file acts as the new source
}

*/
//Now encoded file is in source1.I want to store encoded file in midtar
//Hence I rename it.
/* rename(source1,midtar);
end=clock();
time=(end-start)/(float)CLK_TCK;
// time in micro second
rec.enoprt=totalop;

```

```

rec.entime=time;*/

// Decoding phase begins here. Start with higher stream length and decode to
// 8-bit stream length for each character read from the encoded file to get
// the original source .
    nop=0.0;
    totalop=0.0;
    start=clock();
    nb=endblk;
    totalop=tri_deco(midtar, source1,nb);

/*
for(nb=endblk/2;nb>=startblk;nb = nb/2)
{
    nop=tri_deco(source1,source2,nb);
    totalop += nop;
    strcpy(tempfile,source1);
    strcpy(source1,source2);
    strcpy(source2,tempfile);
x
}
*/
rename(source1,target);
end=clock();
time=(end-start)/(float)CLK_TCK;

rec.deoprt=totalop;
rec.detime=time;

// fs=fopen(source,"rb");// Open the different files used in
// fm=fopen(midtar,"rb");// encoding & decoding phases to
// ft=fopen(target,"rb");// calculate their respective sizes.

// Open the info. file in binary append mode to write different information
// gathered about the files used in en-de coding phases.
//rec.bsize=startblk;
/* fp=fopen("data11.txt","a");
if(fp==NULL)
{
    printf("\n File Opening Error. ");
    exit();
}

fseek(fs,0,SEEK_END);
rec.ssize=f.tell(fs);
fseek(fs,0,SEEK_SET);
printf("\n%ld",rec.ssize); */

```

```

// fseek(fm,0,SEEK_END); // different files used
// rec.msize=fstell(fm);

// fseek(ft,0,SEEK_END); // and copy them to the rec.
// rec.tsize=fstell(ft);

// fwrite(&rec,sizeof(rec),1,fp); // Write to info. file
// fclose(fs);
// fclose(fm);
// fclose(ft);
// fclose(fp);
printf("\n Do You Want To Continue ?(1/0)");
scanf("%d",&ans);
}

/*
// printf("\n Number of Operation= %Lf ",noofop);
fp=fopen("data11.txt","rb"); // Open the info. file for reading the
printf("sFile eFile sSize tSize Comp-time Dcomp-time copr dopr \n");
while(fread(&rec,sizeof(rec),1,fp)==1) // different runs conducted.
{
    printf("\n%-12s %-8s ",rec.fsource,rec.fmtdtar); // rec.ftarget);
    printf("%ld %ld",rec.ssize,rec.tsize); // ,rec.msize,rec.tsize);
    printf("%.6f %.6f ",rec.entime,rec.detime);
    printf("%.0f %.0f ",rec.enoprt,rec.deoprt);
}
fclose(fp);
*/
}

//THE RPPO ENCRYPTION CODE
// RPPO Encoding Scheme.
// Recursive Pair Parity Operation.

#include<stdio.h>
#include<math.h>
#include<time.h>
#include<string.h>
//double noofop=0.0; // Counter to caluclate no. of operations

```

```

// Function for converting binary to corresponding ascii value.
int equiasci(int tar[])
{
    int asc=0;
    int i;
    for(i=0;i<=7;i++)
        asc+=tar[i] * pow((double)2,(double)(7-i));

```

```

        return asc;
    }

// Function for converting a character read to its equivalent binary.
void binary(char ch,int mid[])
{
    int i,mask,k;
    for(i=7;i>=0;i--)
    {
        mask=1<<i; // left shift 1 i times.
        k=ch & mask; // Extract the ith bit of ch;
        mid[7-i]=(k==0)?0:1; // And store it in array mid for future use.
    }
}

long float operation(char source[],char dest[],int nb,int iteratio)
{
    FILE *fs,*ft;
    char ch;
    int i,j,k;
    long int size,count,rem;
    int temp[8];
    int *mid,*tar;
    int asc=0;
    static long float noofop;
    fs=fopen(source,"rb"); //Open the source file.
    ft=fopen(dest,"wb"); // Open the target file.
    if(fs==NULL || ft==NULL)
    {
        printf("\n File Opening Error. ");
        exit();
    }

    fseek(fs,0,SEEK_END);
    size=f.tell(fs);
    fseek(fs,0,SEEK_SET);
    mid=(int *)calloc(nb,sizeof(int));
    tar=(int *)calloc(nb,sizeof(int));
    rem=size/(nb/8);
    for(count=0;count<rem;count++)
    {

//This loop reads nb/8 characters or nb bits from source and store it in mid
//array.

```

```

for(i=0,j=0;i<nb/8;i++)
{
    ch=getc(fs); // Read each character from the source file.
    binary(ch,temp); // Call the funct. binary.
//mid array is used to store the binary value of all the characters read from
//file
    for(k=0;k<8;k++)
        *(mid+j++)=temp[k];
}
// Manipulate the bits & store it in array for future use according
// to the rule
// tar[0]=mid[0]
// tar[i]=mid[i] EXOR tar[i-1]
    for(j=1;j<=iteratio;j++)
    {
        tar[nb-1]=mid[nb-1];
        for(i=nb-2;i>=0;i--)
        {
            tar[i]=mid[i] ^ tar[i+1];
            noofop+=1;
        }
    }
//This loop copies the contents of tar into mid for performing the same type
//of operation.
    for(i=0;i<nb;i++)
        mid[i]=tar[i];
}
//temp array is used to store 8-bits at a time, find their equivalent ascii
// and write it to target file.
    for(i=0,j=0;i<nb;i++)
    {
        *(temp+j++)=tar[i];
        if(j%8==0)
        {
            j=0;
            asc=equiasci(temp); // Call the funct. equiasci
// Write the corresponding character in the target file.
            putc(asc,ft);
        }
    }
}
// Calculate the remaining characters
rem=size%(nb/8);
for(count=0;count<rem;count++)
{
    ch=getc(fs);
    putc(ch,ft);
}

```

```

    }
    fclose(fs);
    fclose(ft);
    free(mid);
    free(tar);
    return noofop;
}

// Define a structure for writing info. about the src file,coded file,decoded
// file, their file size, coding-decoding time, in another file.
struct tablerec
{
    char fsource[25];
    char fmtdtar[15];
    //char fttarget[15];
    long int ssize;
    //long int msize;
    //long int tsize;
    float entime;
    float detime;
    long float enoprt;
    long float deoprt;
};

void main()
{
    int ans=1,nb;
    static long float nop,totalop;
    int startblk,endblk,iteratio,deiter;
    char source[30],midtar[30],target[30];
    char tempfile[]="temp.txt"; //These files are used for temporarily
    char source1[]="source1.txt";//holding the mid-term files generated
    char source2[]="source2.txt";// in both the phases.
    FILE *fs,*ft,*fm,*fp;
    clock_t start,end;

    float time;
    struct tablerec rec;

    while(ans)
    {
        printf("\n Enter Source file name :");
        scanf("%s",source);
        strcpy(rec.fsource,source);
        printf("\n Enter Encrypted file name :");

```

```

        scanf("%s",midtar);
        strcpy(rec.fmidtar,midtar);
/* printf("\n Enter Final Target file name :");
scanf("%s",target);*/
// strcpy(rec.ftarget,target);
//while(1)
//{
//{
startblk=8;
//printf("\nEnter starting block size : ");
//scanf("%d",&startblk);
endblk=256;
//printf("\nEnter end block size : ");
//scanf("%d",&endblk);
iteratio=1;
//printf("\nEnter number of iteration : ");
//scanf("%d",&iteratio);
/*if(iteratio>=startblk)
{
    printf("\nWrong Value For Iteration.");
    continue;
}
else
    break;
} */
// Encoding phase begins here. Start with 8-bit stream length and encode to
// multiple of 8-bit stream length (like 16,32,64,...,256) for each character read
// from the original file.

start=clock();
totalop=operation(source,source1,startblk,iteratio);

for(nb=2*startblk;nb<=endblk;nb += nb)
{
    nop=operation(source1,source2,nb,iteratio);
    totalop += nop;
    strcpy(tempfile,source1);//Swap the name of the low-bit
    strcpy(source1,source2);// & hi-bit file so that hi-bit
    strcpy(source2,tempfile);//file acts as the new source
}

//Now encoded file is in source1.I want to store encoded file in midtar
//Hence I rename it.
rename(source1,midtar);
end=clock();
time=(end-start)/(float)CLK_TCK;
// time in micro second
rec.enoprt=totalop;

```

```

rec.entime=time;

// Decoding phase begins here. Start with higher stream length and decode to
// 8-bit stream length for each character read from the encoded file to get
// the original source .
/*
    nop=0.0;
    totalop=0.0;
    start=clock();
    deiter=endblk-iteratio;
    totalop=operation(midtar,source1,endblk,deiter);
    for(nb=endblk/2;nb>=startblk;nb = nb/2)
    {
        deiter=nb-iteratio;
        nop=operation(source1,source2,nb,deiter);
        totalop += nop;
        strcpy(tempfile,source1);
        strcpy(source1,source2);
        strcpy(source2,tempfile);
    }
    rename(source1,target);
    end=clock();
    time=(end-start)/(float)CLK_TCK;

    rec.deoprt=totalop;
    rec.detime=time;

    fs=fopen(source,"rb");// Open the different files used in
    // fm=fopen(midtar,"rb");// encoding & decoding phases to
    // ft=fopen(target,"rb");// calculate their respective sizes.

// Open the info. file in binary append mode to write different information
// gathered about the files used in en-de coding phases.
    fp=fopen("data1.txt","a");
    if(fp==NULL)
    {
        printf("\n File Opening Error. ");
        exit();
    }

    fseek(fs,0,SEEK_END);// Calculate the size of
    rec.ssize=f.tell(fs);

    //fseek(fm,0,SEEK_END);// different files used
    //rec.msize=f.tell(fm);

```

```

//fseek(ft,0,SEEK_END);// and copy them to the rec.
//rec.tsiz=ftell(ft);

fwrite(&rec,sizeof(rec),1,fp);// Write to info. file
fclose(fs);
fclose(fm);
fclose(ft);
fclose(fp);/*
printf("\n Do You Want To Continue ?(1/0)");
scanf("%d",&ans);
}

/*
printf("\n Number of Operation= %Lf ",noofop);
fp=fopen("data11.txt","rb"); // Open the info. file for reading the
printf("sFile eFile sSize tSize Comp-time Dcomp-time copr dopr \n");
while(fread(&rec,sizeof(rec),1,fp)==1) // different runs conducted.
{
    printf("\n%-12s %-8s ",rec.fsource,rec.fmidtar);//rec.ftarget);
    printf("%ld %ld",rec.ssize,rec.tsiz); //,rec.msize,rec.tsiz);
    printf("%.6f %.6f ",rec.entime,rec.detime);
    printf("%.0f %.0f ",rec.enoprt,rec.deoprt);
}
fclose(fp);
*/
}

//THE RPPO DECRYPTION CODE

#include<stdio.h>
#include<math.h>
#include<time.h>
#include<string.h>

// Function for converting binary to corresponding ascii value.
int equiasci(int tar[])
{
    int asc=0;
    int i;
    for(i=0;i<=7;i++)
        asc+=tar[i] * pow(2,(7-i));
    return asc;
}

// Function for converting a character read to its equivalent binary.
void binary(char ch,int mid[])
{

```

```

int i,mask,k;
for(i=7;i>=0;i--)
{
    mask=1<<i; // left shift 1 i times.
    k=ch & mask; // Extract the ith bit of ch;
    mid[7-i]=(k==0)?0:1; // And store it in array mid for future use.
}
}

//Function will take an integer value as argument, then it will test whether
//it is prime or not. If prime return true otherwise false.
int prime(int num)
{
    int d;
    if(num==1)
        return(0);
    else
    {
        for(d=2;d<=sqrt(num);d++)
        {
            if(num%d==0)
                return(0);
        }
        return(1);
    }
}

//This function will take index value of mid array, and it will calculate
//the index of tar array where the bit will assign.
int tarindex(int mindex,int nb)
{
    int temp,index;
    if(mindex==nb)
        return(mindex);

    temp=nb-mindex;
    if(prime(temp))
    {
        if(temp==2)
            temp=nb+1;
        for(index=temp-1;index>=2;index--)
        {
            if(prime(index))
                return(index);
        }
    }
    return(temp);
}

```

```

}

// Number of iteration required to come back to original
// for block size nb
long int iteration(int nb)
{
    int *s,*t;
    int i,mi;
    long int itr;
    static int flag;
    s=(int *)calloc(sizeof(int),nb);
    t=(int *)calloc(sizeof(int),nb);
    for(i=0;i<nb;i++)
        s[i]=i;
    itr=0;
    flag=0;
    while(!flag)
    {
        for(i=0;i<nb;i++)
        {
            mi=tarindex(i+1,nb);
            t[mi-1]=s[i];
        }
        itr++;
        i=0;
        flag=1;
        while(i<nb&&flag)
        {
            if(t[i]!=i)
                flag=0;
            i++;
        }
        for(i=0;i<nb;i++)
            s[i]=t[i];
    }
    return(itr);
}

```

```

// nb bits in s[]. After one iteration result in t[];
void one_primeop(char s[],char t[],int nb)
{
    int i,mi;
    for(i=0;i<nb;i++)
    {
        mi=tarindex(i+1,nb);
        t[mi-1]=s[i];
    }
}

```

```

        }

    }

//operate n times on the bit array
void nprimeop(char s[],char t[],int nb,long int n)
{
    int i,mi,itr;
    itr=0;
    while(itr<n)
    {
        one_primeop(s,t,nb);
        for(i=0;i<nb;i++)
            s[i]=t[i];
        itr++;
    }
}

void pimeoponfile(char sfile[],char tfile[],int nb,long int itr)
{
    FILE *fs,*ft;
    char *t_array, *s_array;
    int i,j,k,nbby8;
    long int it;
    int temp[8],asc,rem;
    unsigned long int size,count,dec,outloop;
    char ch;
    fs=fopen(sfile,"rb"); //Open the source file.
    ft=fopen(tfile,"wb"); // Open the target file.
    if(fs==NULL || ft==NULL)
    {
        printf("\n File Opening Error. ");
        exit();
    }

    fseek(fs,0,SEEK_END);
    size=ftell(fs);
    fseek(fs,0,SEEK_SET);
    s_array=(char *)calloc(nb,sizeof(char));
    t_array=(char *)calloc(nb,sizeof(char));
    nbby8=nb/8;
    outloop=size/nbby8;
    rem=size%nbby8;

    for(count=0;count<outloop;count++)

```

```

{
    k=0;
    for(i=0;i<nbby8;i++)
    {
        ch=getc(fs); // Read each character from the source file.
        binary(ch,temp); // Call the funct. binary.
        for(j=0;j<8;j++)
            s_array[k++]=temp[j];
    }

    nprimeop(s_array,t_array,nb,itr);

    for(i=0,j=0;i<nb;i++)
    {
        *(temp+j++)=t_array[i];
        if(j%8==0)
        {
            j=0;
            asc=equiasci(temp); // Call the funct. equiasci
            // Write the corresponding character in the target file.
            putc(asc,ft);
        }
    }
}

for(count=0;count<rem;count++)
{
    ch=getc(fs);
    putc(ch,ft);
}
fclose(fs);
fclose(ft);
free(s_array);
free(t_array);

}

void main()
{
    int nb;
    int startblk,endblk;
    unsigned long int itr;
    char source[15],target[15];
    clock_t start,end;
}

```

```

float time;

printf("\n Enter Source file name :");
scanf("%s",source);

printf("\n Enter Target file name :");
scanf("%s",target);

printf("\n Enter stream size : ");
scanf("%d",&nb);
start=clock();
itr=iteration(nb);
itr -=itr/2;
pimeoponfile(source,target,nb, itr);
end=clock();
time=(end-start)/(float)CLK_TCK;

printf("\nDecrypted file is \" %s \"",target);
//printf("\nFile size = %lu ",size);
printf("\nTime taken for decryption = %.4f",time);
getch();

}

//THE RPMS ENCRYPTION CODE

```

```

#include<stdio.h>
#include<math.h>
#include<time.h>
//#include<string.h>

// Function for converting binary to corresponding decimal value. Maximum
// value of nb can be 32
unsigned long int equideci(int s[],int nb)
{
    unsigned long int dec=0;
    int i;
    for(i=0;i<nb;i++)
        dec+=s[i] * pow((double)2,(double)(nb-i-1));
    return dec;
}

// Function for converting a character read to its equivalent binary.
void binary(unsigned int dec,int mid[],int nb)

```

```

{
    int i,k;
    unsigned long int mask;
    for(i=nb-1;i>=0;i--)
    {
        mask=1<<i; // left shift 1 i times.
        k=dec & mask; // Extract the ith bit of ch;
        mid[nb-1-i]=(k==0)?0:1; // And store it in array mid for future use.
    }
}

// encrypt the equivalent decimal value of source stream
void oddeven_en(unsigned long int dec,int t[],int nb)
{
    int i;
    //unsigned long int dec;
    //dec=equideci(s,nb);
    //t=calloc(sizeof(int),nb);
    for(i=0;i<nb;i++)
    {
        if(dec & 1)
        {
            t[i]=1;
            dec=dec/2+1;
        }
        else
        {
            t[i]=0;
            dec=dec/2;
        }
    }
}

void oddeven_de(int s[],int msbchanged,int t[],int nb)
{
    unsigned long int dec;
    int i;
    if(s[nb-1]==0)
        dec=2;
    else
        dec=1;
    for(i=nb-2;i>=0;i--)
    {
        if(s[i]==0)
            dec +=dec;
        else
            dec +=dec-1;
    }
}

```

```

        }
        binary( dec,t, nb);
        if(msbchanged)
            t[0]=t[0]^1;
    }

void oddeven_enco(char source[],char dest[], int nb)
{
    FILE *fs,*ft;
    int *t_array, *s_array;
    int i;
    int temp[8],asc,rem;
    unsigned long int size,count,dec,outloop;
    int nbby8,j,k;
    //static long float nop,totalop;
    char ch;
    fs=fopen(source,"rb"); //Open the source file.
    ft=fopen(dest,"wb"); // Open the target file.
    if(fs==NULL || ft==NULL)
    {
        printf("\n File Opening Error. ");
        exit();
    }

    fseek(fs,0,SEEK_END);
    size=f.tell(fs);
    fseek(fs,0,SEEK_SET);
    s_array=(int *)calloc(nb,sizeof(int));
    t_array=(int *)calloc(nb,sizeof(int));
    nbby8=nb/8;
    outloop=size/nbby8;
    rem=size%nbby8;
    for(count=0;count<outloop;count++)
    {
        k=0;
        for(i=0;i<nbby8;i++)
        {
            ch=getc(fs); // Read each character from the source file.
            //dec=ch;
            binary(ch,temp,8); // Call the funct. binary.
            for(j=0;j<8;j++)
                s_array[k++]=temp[j];
        }
        dec=equideci(s_array,nb);
        oddeven_en( dec,t_array,nb);
    }
}

```

```

        for(i=0,j=0;i<nb;i++)
        {
            *(temp+j++)=t_array[i];
            if(j%8==0)
            {
                j=0;
                asc=equideci(temp,8); // Call the funct. equiasci
                // Write the corresponding character in the target file.
                putc(asc,ft);
            }
        }
        for(count=0;count<rem;count++)
        {
            ch=getc(fs);
            putc(ch,ft);
        }
        fclose(fs);
        fclose(ft);
        free(s_array);
        free(t_array);
    }

void main()
{
    int nb;
    int startblk,endblk;
    char source[15],target[15];
    clock_t start,end;

    float time;

    printf("\n Enter Source file name :");
    scanf("%s",source);

    printf("\n Enter Target file name :");
    scanf("%s",target);

    printf("\n Enter stream size : ");
    scanf("%d",&nb);
    start=clock();
    oddeven_enco(source,target,nb);
    end=clock();
    time=(end-start)/(float)CLK_TCK;
}

```

```

        printf("\nEncrypted file is \' %s \'",target);
        //printf("\nFile size = %lu ",size);
        printf("\nTime taken for encryption = %.4f",time);
        getch();
    }

```

### **//THE RPMS DECRYPTION CODE**

```

#include<stdio.h>
#include<math.h>
#include<time.h>
#include<string.h>

// Function for converting binary to corresponding decimal value. Maximum
// value of nb can be 32
unsigned long int equideci(int s[],int nb)
{
    unsigned long int dec=0;
    int i;
    for(i=0;i<nb;i++)
        dec+=s[i] * pow((double)2,(double)(nb-i-1));
    return dec;
}

// Function for converting a character read to its equivalent binary.
void binary(unsigned int dec,int mid[],int nb)
{
    int i,k;
    unsigned long int mask;
    for(i=nb-1;i>=0;i--)
    {
        mask=1<<i; // left shift 1 i times.
        k=dec & mask; // Extract the ith bit of ch;
        mid[nb-1-i]=(k==0)?0:1; // And store it in array mid for future use.
    }
}

void oddeven_de(int s[],int msbchanged,int t[],int nb)
{
    unsigned long int dec;
    int i;
    if(s[nb-1]==0)
        dec=2;

```

```

else
    dec=1;
for(i=nb-2;i>=0;i--)
{
    if(s[i]==0)
        dec +=dec;
    else
        dec +=dec-1;
}
binary( dec,t, nb);
if(msbchanged)
    t[0]=t[0]^1;
}
void oddeven_deco(char source[],char dest[], int nb)
{
    FILE *fs,*ft;
    int *t_array, *s_array;
    int i;
    int temp[8],asc,rem;
    unsigned long int size,count,dec,outloop;
    int nbby8,j,k;
    //static long float nop,totalop;
    char ch;
    fs=fopen(source,"rb"); //Open the source file.
    ft=fopen(dest,"wb"); // Open the target file.
    if(fs==NULL || ft==NULL)
    {
        printf("\n File Opening Error. ");
        exit();
    }

    fseek(fs,0,SEEK_END);
    size=f.tell(fs);
    fseek(fs,0,SEEK_SET);
    s_array=(int *)calloc(nb,sizeof(int));
    t_array=(int *)calloc(nb,sizeof(int));
    nbby8=nb/8;
    outloop=size/nbby8;
    rem=size%nbby8;
    for(count=0;count<outloop;count++)
    {
        k=0;
        for(i=0;i<nbby8;i++)
        {
            ch=getc(fs); // Read each character from the source file.

```

```

        binary(ch,temp,8); // Call the funct. binary.
        for(j=0;j<8;j++)
            s_array[k++]=temp[j];
    }
    oddeven_de(s_array, 0, t_array, nb);
    //dec=equideci(t_array,nb);

    for(i=0,j=0;i<nb;i++)
    {
        *(temp+j++)=t_array[i];
        if(j%8==0)
        {
            j=0;
            asc=equideci(temp,8); // Call the funct. equiasci
        // Write the corresponding character in the target file.
            putc(asc,ft);
        }
    }
}
for(count=0;count<rem;count++)
{
    ch=getc(fs);
    putc(ch,ft);
}
fclose(fs);
fclose(ft);
free(s_array);
free(t_array);
}

}

```

```

void main()
{
    int nb;
    int startblk,endblk;
    char source[15],target[15];
    clock_t start,end;

    float time;

    printf("\n Enter Source file name :");
    scanf("%s",source);

    printf("\n Enter Target file name :");
    scanf("%s",target);

```

```

printf("\n Enter stream size : ");
scanf("%d",&nb);
start=clock();
oddeven_deco(source,target,nb);
end=clock();
time=(end-start)/(float)CLK_TCK;

printf("\nEncrypted file is \'%s\'",target);
printf("\nTime taken for encryption = %.4f",time);
getch();

}

//THE RSBP ENCRYPTION CODE
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<iostream.h>
#include<time.h>
void main()
{
FILE *f1,*f2;
int i=1,c=1,d=1,prm[500],nprm[500],n,j,k,t,ad,flag,cd,l,
    cd[60000],m,x[10],temp[10],sm,u,pr=0,npr=0;
long int bin[60000],l;
clock_t start,end;
float time;
char str[490000],st[15],st1[15];
printf("Enter the input file name");
scanf("%s",st);
printf("Enter the encrypted file name");
scanf("%s",st1);
start=clock();
for(n=0;n<=255;n++)
{
    k=0;
    for(j=2;j<n;j++)
    {
        if(n%j==0)
            k=1;
    }
    if(k==0)
    {
        prm[c]=n;
        c++;
    }
}

```

```

    else
    {
        nprm[d]=n;
        d++;
    }
}
f1=fopen(st,"rb");
if((f1=fopen(st,"rb"))==NULL)
    printf("can not open file");
f2=fopen(st1,"wb");
pr=0;
i=1;
while((n=getc(f1))!=EOF)
{
    for(j=1;j<c;j++)
    {
        if(prm[j]==n)
        {
            cd[i]=1;
            pr++;
            fprintf(f2,"%d",cd[i]);
            m=j;
        }
    }
    for(j=1;j<d;j++)
    {
        if(nprm[j]==n)
        {
            cd[i]=0;
            npr++;
            fprintf(f2,"%d",cd[i]);
            m=j;
        }
    }
    j=0;
    while(m>0)
    {
        j++;
        x[j]=m%2;
        m=m/2;
    }
    bin[i]=0;
    for(k=j;k>0;k--)
    {
        bin[i]=bin[i]*10+x[k];
    }
}

```

```

i++;
}
printf("\nThe size of source file is %d\n",i-1);
sm=pr*7+npr*9;
ad=8-(sm%8);
flag=0;
pr=1;
    while(pr<=ad)
    {
        fprintf(f2,"%d",flag);
        pr++;
    }
for(j=i-1;j>0;j--)
{
    if(cd[j]==1)
    {
        k=5;
        l=bin[j];
        while(k>=0)
        {
            temp[k]=l%10;
            k--;
            l=l/10;
        }
        k=0;
        while(k<=5)
        {
            fprintf(f2,"%d",temp[k]);
            k++;
        }
    }
    else
    {
        k=7;
        l=bin[j];
        while(k>=0)
        {
            temp[k]=l%10;
            k--;
            l=l/10;
        }
        k=0;
        while(k<=7)
        {
            fprintf(f2,"%d",temp[k]);
            k++;
        }
    }
}

```

```

        }
    }

}

fclose(f1);
fclose(f2);
f2=fopen(st1,"rb");
i=1;
//flag=pr+npr;
while((str[i]==getc(f2))!=EOF)
    i++;
fclose(f2);
f2=fopen(st1,"wb");
n=i/8;
//n=pr+npr;
//printf("enter the size of source file in KB");
//scanf("%d")
k=1;
for(j=1;j<=n;j++)
{
    cd1=1;
    while(cd1<=8)
    {
        if(str[k]=='1')
            x[cd1]=1;
        if(str[k]=='0')
            x[cd1]=0;
        k++;
        cd1++;
    }
    sm=0;
    cd1--;
    while(cd1>0)
    {
        if(x[cd1]==1)
        {
            t=1;
            u=1;
            while(u<=(8-cd1))
            {
                u++;
                t=t*2;
            }
            sm=sm+t;
        }
    }
}

```

```

        cd1--;
    }

    fprintf(f2,"%c",sm);
}

fclose(f2);
// getch();
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("Time Required=%.8f second",time);
fclose(f2);

}

//THE RSBP DECRYPTION CODE
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<iostream.h>
#include<time.h>
void main()
{
    FILE *f3,*f2;
    int m,i=1,c=1,d=1,prm[500],nprm[500],n,j,k,t,rd,pr=0,npr=0,b,a,s=0;
    clock_t start,end;
    float time;
    char str[490000],st2[15],st1[15],temp1[10];
    printf("Enter the encrypted file name\n");
    scanf("%s",st1);
    printf("Enter the decrypted file name");
    scanf("%s",st2);
    printf("Enter the size of source file in KB");
    scanf("%d",&n);
    start=clock();
    for(m=0;m<=255;m++)
    {
        k=0;
        for(j=2;j<m;j++)
        {
            if(m%j==0)
                k=1;
        }
        if(k==0)
        {
            prm[c]=m;
            c++;
        }
    }
}

```

```

        else
        {
            nprm[d]=m;
            d++;
        }
    }

f2=fopen(st1,"rb");
f3=fopen(st2,"wb");
i=0;
k=0;
while((rd=getc(f2))!=EOF)
{
    j=7;
    while(j>=0)
    {
        if(rd%2==0)
            temp1[j]='0';
        else
            temp1[j]='1';
        rd=rd/2;
        j--;
    }
    j=0;
    while(j<8)
    {
        str[k]=temp1[j];
        j++;
        k++;
    }
}

b=k-1;
for(j=0;j<n;j++)
{
    if(str[j]=='1')
    {
        a=0;
        c=5;
        while(a<6)
        {
            temp1[a]=str[b-c];
            a++;
            c--;
        }
        b=b-6;
        s=0;
    }
}

```

```

for(i=5;i>=0;i--)
{
    if(temp1[i]=='1')
    {
        k=1;
        t=1;
        while(k<=(5-i))
        {
            t=t*2;
            k++;
        }
        s=s+t;
    }
}
else
{
    a=0;
    c=7;
    while(a<8)
    {
        temp1[a]=str[b-c];
        a++;
        c--;
    }
    b=b-8;
    s=0;
    for(i=7;i>=0;i--)
    {
        if(temp1[i]=='1')
        {
            k=1;
            t=1;
            while(k<=(7-i))
            {
                t=t*2;
                k++;
            }
            s=s+t;
        }
    }
}
if(str[j]=='1')
s=prm[s];
if(str[j]=='0')

```

```

        s=nprm[s];
        fprintf(f3,"%c",s);

    }
end=clock();
time=(end-start)/(float)CLK_TCK;
printf("Time Required=%.8f second",time);
fclose(f2);
fclose(f3);
}

//THE RSBM ENCRYPTION CODE
#include<stdio.h>
#include<ctype.h>
#include<time.h>
void main()
{
FILE *f1,*f2,*f3;
int i=1,j,n,cd,m,k,x[10],temp[10].a,b,s,t,sm,u;
long int bin[80000],l;
char c,str[640000],temp1[10],st1[15],st1[15],st2[15];
clock_t start1,end1,start2,end2;
float time_en,time_de;
printf("Enter the input file name: ");
scanf("%s",st);
printf("Enter the encrypted file name: ");
scanf("%s",st1);
/*printf("Enter the decrypted file name: ");
scanf("%s",st2);*/
start1=clock();
f1=fopen(st,"rb");
f2=fopen(st1,"wb");
while((n=getc(f1))!=EOF)
{
    if(n%2==0)
    {
        cd=0;
        fprintf(f2,"%d",cd);
        m=n/2;
    }
    else
    {
        cd=1;
        fprintf(f2,"%d",cd);
        m=n/2+1;
    }
    j=0;
}

```

```

while(m>0)
{
    j++;
    x[j]=m%2;
    m=m/2;
}
bin[i]=0;
for(k=j;k>0;k--)
    bin[i]=bin[i]*10+x[k];
i++;
}
for(j=i-1;j>0;j--)
{
k=6;
l=bin[j];
while(k>=0)
{
temp[k]=l%10;
k--;
l=l/10;
}
k=0;
while(k<=6)
{
fprintf(f2,"%d",temp[k]);
k++;
}
}
fclose(f1);
fclose(f2);
f2=fopen(st1,"rb");
i=1;
while((str[i]==getc(f2))!=EOF)
    i++;
fclose(f2);
f2=fopen(st1,"wb");
n=i/8;
k=1;
for(j=1;j<=n;j++)
{
cd=1;
while(cd<=8)
{
if(str[k]=='1')
    x[cd]=1;
if(str[k]=='0')
    x[cd]=0;
cd++;
}
}

```

```

        x[cd]=0;
        k++;
        cd++;
    }
    sm=0;
    cd--;
    while(cd>0)
    {
        if(x[cd]==1)
        {
            t=1;
            u=1;
            while(u<=(8-cd))
            {
                u++;
                t=t*2;
            }
            sm=sm+t;
        }
        cd--;
    }
    fprintf(f2,"%c",sm);
}
fclose(f2);
end1=clock();
time_en=(end1-start1)/(float)CLK_TCK;
/*start2=clock();
f2=fopen(st1,"rb");
f3=fopen(st2,"wb");
i=0;
k=0;
while((n=getc(f2))!=EOF)
{
    j=8;
    while(j>0)
    {
        if(n%2==0)
            temp1[j]='0';
        else
            temp1[j]='1';
        n=n/2;
        j--;
    }
    j++;
    while(j<=8)
    {

```

```

        str[k]=temp1[j];
        j++;
        k++;
    }
}
n=k/8;
b=k;
for(j=0;j<n;j++)
{
a=0;
c=7;
while(a<7)
{
    temp1[a]=str[b-c];
    a++;
    c--;
}
b=b-7;
s=0;
for(i=6;i>=0;i--)
{
    if(temp1[i]=='1')
    {
        k=1;
        t=1;
    while(k<=(6-i))
    {
        t=t*2;
        k++;
    }
    s=s+t;
    }
}
if(str[j]=='0')
s=s*2;
else
s=s*2-1;
fprintf(f3,"%c",s);
}
fclose(f2);
fclose(f3);
end2=clock();
time_de=(end2-start2)/(float)CLK_TCK;*/
printf("\nEncryption time: %.4f",time_en);
//printf("\nDecryption time: %.4f",time_de);
printf("\n\n");

```

```

}

//THE RSBM DECRYPTION CODE
#include<stdio.h>
#include<ctype.h>
#include<time.h>
void main()
{
FILE *f1,*f2,*f3;
int i=1,j,n,cd,m,k,x[10],temp[10],a,b,s,t,sm,u;
long int bin[80000],l;
char c,str[640000],temp1[10],st[15],st1[15],st2[15];
clock_t start1,end1,start2,end2;
float time_en,time_de;
/*printf("Enter the input file name: ");
scanf("%s",st);*/
printf("Enter the encrypted file name: ");
scanf("%s",st1);
printf("Enter the decrypted file name: ");
scanf("%s",st2);
/*start1=clock();
f1=fopen(st,"rb");
f2=fopen(st1,"wb");
while((n=getc(f1))!=EOF)
{
    if(n%2==0)
    {
        cd=0;
        fprintf(f2,"%d",cd);
        m=n/2;
    }
    else
    {
        cd=1;
        fprintf(f2,"%d",cd);
        m=n/2+1;
    }
    j=0;
    while(m>0)
    {
        j++;
        x[j]=m%2;
        m=m/2;
    }
    bin[i]=0;
    for(k=j;k>0;k--)
        bin[i]=bin[i]*10+x[k];
}
}

```

```

        i++;
}
for(j=i-1;j>0;j--)
{
k=6;
    l=bin[j];
    while(k>=0)
    {
        temp[k]=l%10;
        k--;
        l=l/10;
    }
k=0;
    while(k<=6)
    {
        fprintf(f2,"%d",temp[k]);
        k++;
    }
}
fclose(f1);
fclose(f2);
f2=fopen(st1,"rb");
i=1;
while((str[i]==getc(f2))!=EOF)
    i++;
fclose(f2);
f2=fopen(st1,"wb");
n=i/8;
k=1;
for(j=1;j<=n;j++)
{
    cd=1;
    while(cd<=8)
    {
        if(str[k]=='1')
            x[cd]=1;
        if(str[k]=='0')
            x[cd]=0;
        k++;
        cd++;
    }
    sm=0;
    cd--;
    while(cd>0)
    {
        if(x[cd]==1)

```

```

    {
        t=1;
        u=1;
        while(u<=(8-cd))
        {
            u++;
            t=t*2;
        }
        sm=sm+t;
    }
    cd--;
}
fprintf(f2,"%c",sm);
}
fclose(f2);
end1=clock();
time_en=(end1-start1)/(float)CLK_TCK;*/
start2=clock();
f2=fopen(st1,"rb");
f3=fopen(st2,"wb");
i=0;
k=0;
while((n=getc(f2))!=EOF)
{
    j=8;
    while(j>0)
    {
        if(n%2==0)
            temp1[j]='0';
        else
            temp1[j]='1';
        n=n/2;
        j--;
    }
    j++;
    while(j<=8)
    {
        str[k]=temp1[j];
        j++;
        k++;
    }
}
n=k/8;
b=k;
for(j=0;j<n;j++)
{

```

```

a=0;
c=7;
while(a<7)
{
    temp1[a]=str[b-c];
    a++;
    c--;
}
b=b-7;
s=0;
for(i=6;i>=0;i--)
{
    if(temp1[i]=='1')
    {
        k=1;
        t=1;
        while(k<=(6-i))
        {
            t=t*2;
            k++;
        }
        s=s+t;
    }
}
if(str[j]=='0')
s=s*2;
else
s=s*2-1;
fprintf(f3,"%c",s);
}
fclose(f2);
fclose(f3);
end2=clock();
time_de=(end2-start2)/(float)CLK_TCK;
//printf("\nEncryption time: %.4f",time_en);
printf("\nDecryption time: %.4f",time_de);
printf("\n\n");
}
// THE CODE TO CALCULATE PEARSONIAN CHI SQUARE VALUE
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<conio.h>
void main(int argc,char *argv[])
{
    clrscr();

```

```

if(argc!=3)
{
    printf("Error in arguments.");
    exit(1);
}
long int temp1[256],temp2[256];
int c;
for(int i=0;i<256;i++)
{
    temp1[i]=0;
    temp2[i]=0;
}
FILE *f1,*f2;
f1=fopen(argv[1],"rb");
while((c=fgetc(f1))!=EOF)
{
    for(i=0;i<256;i++)
    {
        if(c==i)
            temp1[i]++;
    }
}
fclose(f1);
f2=fopen(argv[2],"rb");
while((c=fgetc(f2))!=EOF)
{
    for(i=0;i<256;i++)
    {
        if(c==i)
            temp2[i]++;
    }
}
fclose(f2);
float chivalue=0.0;
int fr=0;
for(i=0;i<256;i++)
{
    unsigned long int a=pow((temp2[i]-temp1[i]),2);
    if(temp1[i]!=0)
    {
        fr++;
        chivalue=chivalue+a/temp1[i];
    }
}
clrscr();

```

```

/*printf("Following is the list of frequencies of respectively all characters in the source
file:\n");
for(i=0;i<256;i++)
{
printf("%d ",temp1[i]);
sum+=temp1[i];
}
printf("\n%ld",sum);
sum=0;
printf("\nFollowing is the list of frequencies of respectively all characters in the
encrypted file:\n");
for(i=0;i<256;i++)
{
printf("%d ",temp2[i]);
sum+=temp2[i];
}
printf("\n%ld",sum);*/
printf("\nThe Chi Square Value is %.4f.\n",chisquare);
printf("Degree of Freedom: %d",fr-1);
}

//THE CODE FOR TESTING FREQUENCY DISTRIBUTION
#include<stdio.h>
#include<graphics.h>
#include<conio.h>
void main(argc,argv)
int argc;
char *argv[];
{
    FILE *fp,*fp1;
    int n;
    int i,j,p,t,l;
//    unsigned long int temp[256],temp1[256];
    int gd=DETECT,gm;
    initgraph(&gd,&gm,"c:\\tc\\bgi");
    setbkcolor(WHITE);
    setcolor(BLUE);
    if(argc!=3)
        outtextxy(50,50,"Improper Input");
    line(50,10,50,425);
    line(50,425,600,425);
    settextstyle(0,0,1);
    outtextxy(595,422,>"");
    settextstyle(0,1,1);
    outtextxy(55,7,>"");
    long int temp1[256],temp2[256];
    int c;
}

```

```

for(i=0;i<256;i++)
{
    temp1[i]=0;
    temp2[i]=0;
}
FILE *f1,*f2;
f1=fopen(argv[1],"rb");
while((c=fgetc(f1))!=EOF)
{
    for(i=0;i<256;i++)
    {
        if(c==i)
            temp1[i]++;
    }
}
fclose(f1);
f2=fopen(argv[2],"rb");
while((c=fgetc(f2))!=EOF)
{
    for(i=0;i<256;i++)
    {
        if(c==i)
            temp2[i]++;
    }
}
fclose(f2);

t=55;
setcolor(BLUE);
for(l=0;l<256;l++)
{
    for(i=420;i>(420-temp1[l]);i--)
    {
        outtextxy(t,i,".");
    }
    t+=4;
}
t=57;
setcolor(RED);
for(l=0;l<256;l++)
{
    for(i=420;i>(420-temp2[l]);i--)
    {
        outtextxy(t,i,".");
    }
    t+=4;
}

```

```
}

setcolor(BLUE);
settextstyle(0,0,1);
outtextxy(150,440,"Characters");
settextstyle(0,1,1);
outtextxy(35,220,"Freequency");
getch();
closegraph();
restorecrtmode();

}
```

## **Appendix D**

## **Bibliography**

1. W F Friedman, "The Index of Coincidence and Its Applications in Cryptography", Riverbank Publication No. 22, Riverbanks Labs, 1920. Reprinted by Aegean Park Press, 1987.
2. E. H. Hebern, "Electronic Coding Machine," U.S. Patent #1,510,441, 30 Sep 1924.
3. C. E. Shannon, "Communication Theory of Secrecy Systems," Bell System Technical Journal, v.28, n.4, 1949, pp. 656 – 715.
4. D. Kahn, *The Codebreakers: The story of Secret Writing*, New York: Macmillan Publishing Co., 1983.
5. J. L. Smith, "The Design of Lucifer A Cryptographic Device for Data Communication," IBM Research Report RC3326, 1971.
6. J. L. Smith, W. A. Notz, and P.R. Osseck, "An Experimental Application of Cryptography to a Remotely Accessed Data System," Proceedings of ACM Annual Conference, August 1972, pp. 282-290.
7. W. Diffie and M.E. Hellman, "New Directions in Cryptography," IEEE Transactions on Information Theory, v. IT-22, n. 6, Nov 1976, pp. 644 – 654.
8. W. Diffie and M.E. Hellman, "Privacy and Authentication: An Introduction to Cryptography," Proceedings of the IEEE, v. 67, n. 3, Mar 1979, pp. 397-427.
9. L. R. Knudsen, "Block Ciphers – Analysis, Design, Applications," Ph. D. Dissertation, Aarhus University, Nov 1994.
10. P. Wayner, "Mimic Functions," Cryptologia, v. 16, n. 3, Jul 1992. pp. 193-214.
11. P. Wayner, "Mimic Functions and Tractability," draft manuscript, 1993.
12. W. F. Friedman, *Elements of Cryptanalysis*. Laguna Hills, CA: Aegean Park Press, 1976.
13. H. F. Gines, *Cryptanalysis*. American Photographic Press, 1937, (Reprinted by Dover Publications, 1956.)
14. E. A. Williams. *An Invitation to Cryptograms*, New York: Simon and Schuster, 1959.
15. R. Ball, *Mathematical Recreations and Essays*. New York: MacMillan, 1960.
16. A. Sinkov, *Elementary Cryptanalysis*, Mathematical Association of America, 1996.

17. A. Peleg and A. Rosenfield, "Breaking Substitution Ciphers using a Relaxation Algorithm," *Communications of ACM*, v.22, n. 11, Nov 1979, pp. 598-605.
18. A. G. Konheim, *Cryptography: A Primer*. New York: John Wiley & Sons, 1981.
19. G. W. Hart, "To Decode Short Cryptograms," *Communications of ACM*, v. 37, n. 9, Sep 1994, pp. 102-108.
20. D. Kahn, *The Codebreakers: The Story of Secret Writing*, New York: Macmillan Publishing Co., 1967.
21. F. Pratt, *Secret and Urgent*, Blue Ribbon Books, 1942.
22. W. F. Friedman, *Methods for the Solutions of Running-Key Ciphers*, Riverbank Publication No. 16, Riverbank Labs, 1981.
23. A. Scherbius, "Ciphering Machine," U.S. Patent #1,657,411. 24 Jan 1928.
24. W. G. Barker, Cryptanalysis of the Hagelin Cryptograph, Aegean Park Press, 1977.
25. C.A. Deavours and L. Kruh, *Machine Cryptography and Modern Cryptanalysis*, Norwood MA: Artech House, 1985.
26. W. Diffie and M.E. Hellman, "Privacy and Authentication: An Introduction to Cryptography," *Proceedings of the IEEE*, v. 67, n. 3, Mar 1979, pp. 397-427.
27. C. A. Deavours, "Black Chamber: A Column; How the British Broke Enigma," *Cryptologia*, v. 4, n.3, Jul 1980, pp.129-132.
28. A. G. Konheim, *Cryptography: A Primer*. New York: John Wiley & Sons, 1981.
29. R.L. Rivest, "Statistical Analysis of Hagelin Cryptography," *Cryptologia*, v. 5, n.1, Jul 1981, pp.27-32.
30. G. Welchman, *The Hut Six Story: Breaking the Enigma Codes*, New York McGraw-Hill, 1982.
31. B. C. W. Hagelin, *The Story of the Hagelin Cryptos*. *Cryptologia*, v. 18, n.3, Jul 1994, pp.204-242.
32. M. Abadi, J. Feigenbaum, and J. Kilan, "On Hiding Information from an Oracle." *Proceedings of 19<sup>th</sup> ACM Symposium on the Theory of Computing*, 1987, pp. 195 – 203.
33. M. Abadi, J. Feigenbaum, and J. Kilan, "On Hiding Information from an Oracle." *Journal of Computer and System Sciences*, v.39, n. 1, Aug 1989, pp. 21-50.

34. M. Abadi, R. Needham, "Prudent Engineering Practice for Cryptographic Protocols", Research Report 125, Digital Equipment Corporation Systems Research centre, Jun 1994.
35. C. M. Adams, " Simple and Effective Key Scheduling for Effective ciphers, "Workshop on Selected Areas on Cryptography – Workshop Record, Kingston, Ontario, 5-6, May 1994, 129 -129.
36. C. M. Adams and S. E. Tavares, "The Structured Design of Cryptographically Good S-Boxes" Journal of Cryptology v. 3, n. 1, 1990, pp. 27-41.
37. W. Adams and D. Shanks, "Strong Primality Test That are Not Sufficient," Mathematics of Computation, v. 39, 1982, pp. 255-300.
38. B. S. Adiga and P. Shankar, "Modified Lulee Cryptosystem", Electronic Letters v. 21, n. 18, 29 Aug 1985, pp. 794-795.'
39. L. M. Adleman, "A Subexponential Algorithm for the Discrete Logarithm Problem with Application to Cryptography, "Proceedings of IEEE 20<sup>th</sup> Annual Symposium of Foundations of Computer Science 1979, pp. 55-60..
40. AT & T, "T7001 Random Number Generator", Data Sheet, August 1986.
41. M. Beale and M. F. Monaghan, "Encryption Using Random Boolean Functions," Cryptography and Coding, H. J. Beker and F. C. Piper, Eds., Oxford: Clarendon Press, 1989, pp. 219-230.
42. P. Beauchemin, G. Brassard, C. Crepeau, C. Goutier, and C. Pomerance, "The Generation of Random Numbers that are Probably Prime," Journal of Cryptology, v. 1, n. 1, 1988, pp. 53-64.
43. S. M. Bellovin and M. Merritt, " An Attack on the Interlock Protocol When Used for Authentication," IEEE Transactions on Information Theory, v. 40,n. 1, Jan 1994, pp. 273-275.
44. S. M. Bellovin and M. Merritt, "Cryptographic Protocol for Secure Communications," U.S. Patent #5, 241,599,31 Aug 1993.
45. J. C. Benaloh, "Cryptographic Capsules: A Disjunctive Primitive for Interactive Protocols," Advances in Cryptology – CRYPTO '86 Proceedings, Springer-Verlag, 1987, pp. 213-222.

46. J. C. Benaloh, "Secret Sharing Homomorphisms: Keeping Shares of a Secret Secret," Advances in Cryptology – CRYPTO '86 Proceedings, Springer-Verlag, 1987, 251-260.
47. A. Bender and G. Castagnoli, "On the implementation of Elliptic Curve Cryptosystems," Advances in Cryptology – CRYPTO '89 Proceedings, Springer-Verlag, 1990, pp. 186-192.
48. S. Berkovits, "How to Broadcast a Secret," Advances in Cryptology – CRYPTO '91 Proceedings, Springer-Verlag, 1991, pp. 535-541.
49. T. Berson, "Differential Cryptanalysis Mod  $2^{32}$  with Applications to MD5," Advances in Cryptology – EUROCRYPT '92 Proceedings, 1992, pp. 71-80.
50. E. Biham and P. C. Kocher, "A Known Plaintext Attack on the PKZIP Encryption," K. U. Leuven Workshop on Cryptographic Algorithms, Springer-Verlag, 1995.
51. M. Bishop, "An Application for a Fast Data Encryption Standard Implementation," Computing Systems, v. 1, n. 3, 1988, pp. 221-254.
52. M. Blaze, and B. Schneier, "The MacGuffin Block Cipher Algorithm," K. U. Leuven Workshops on Cryptographic Algorithms , Springer-Verlag, 1995.
53. M. Blum, and S. Micali, "How to Generate Cryptographically-Strong Sequences of Pseudo-Random Bits," SIAM Journal of Computing, v. 13, n. 4, Nov 1984, pp. 850-864.
54. H. Bonnenberg, A. Curiger, N. Felber, H. Kaeslin, and X Lai, "VLSI Implementation of a New Block Cipher," Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 91), Oct 1991, pp. 510-513.
55. A. Curiger, H. Zimmermann, N. Felber, H. Kaeslin and W. Fichtner, "VINCI: VLSI Implementation of New Block Cipher IDEA," Proceedings of IEEE CICC '93 San Diego, CA, May 1993, pp. 15.5.1 – 15.5.4.
56. A. Curiger and B. Stuber, "Specification for IDEA Chip" Technical Report No. 92/03, Institut fur Integrierte Systeme. ETH Zurich, Feb 1992.
57. G.I. Davida, "Inverse of Elements of a Galois Field," Electronics Letters, v. 8, n. 21, 19 Oct 1972, pp. 518-520.

58. R.C. Fairfield, A. Matusevich, and J. Plany, "AN LSI Digital Encryption Processor (DEP)," IEEE Communications, v. 23, n. 7, Jul 1985, pp. 30-41.
59. P. Finch, "Study of Blowfish Encryption," Ph. D. Dissertation, Department of Computer Science, City University of New York Graduate School and University Center, Feb 1995.
60. S. Goldwasser and J. Kilian, "Almost All Primes Can Be Quickly Certified," Proceedings of the 18<sup>th</sup> ACM Symposium on the Theory of Computing, 1986, pp. 316-329.
61. P. L'Ecuyer, "Random Numbers of Simulation," Communications of the ACM, v.33, n. 10, Oct 1990, pp. 85-97.
62. G. Mayhew, "A Low Cost, High Speed Encryption System and Method," Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy 1994, pp. 147-154.
63. S. B. Morris, "Escrow Encryption," Lecture at MIT Laboratory for Computer Science, 2 June 1994.
64. National Institute of Standards and Technology, "Capstone Chip Technology," 30 Apr 1993.
65. R.L. Rivest, "RSA Chips (Present/Past/Future)," Advances in Cryptology: Proceedings of Eurocrypt 84, Springer-Verlag 1985, pp. 159-168.
66. R.L. Rivest, "The MD4 Message Digest Algorithm," RFC 1321, Apr 1992.
67. R.L. Rivest, "The MD5 Message Digest Algorithm," RFC 1321, Apr 1992.
68. R.L. Rivest, "Dr. Ron Rivest on Difficulty of Factoring," Ciphertext: The RSA Newsletter, v.1, n. 1, 1993, pp. 6,8.
69. R.L. Rivest, "The RC5 Encryption Algorithm," Dr. Dobb's Journal, v.20, n. 1, Jan 1995, pp. 146-148.
70. R.L. Rivest, "The RC5 Encryption Algorithm," K.U. Leuven Workshops on Cryptographic Algorithms, Springer-Verlag, 1995.
71. B. Schneier, "The Blowfish Encryption Algorithm," Dr. Dobb's Journal, v.19, n. 4, Apr 1994, pp. 38-40.
72. B. Schneier, "Protect Your Macintosh, Peachpit Press, 1994.

73. B. Schneier, "The GOST Encryption Algorithm", Dr. Dobb's Journal, v.20, n. 1, Jan 1995, pp. 123-124.
74. B. Yee, "Using Secure Coprocessor," Ph. D. Dissertation, School of Computer Science, Carnegie University, May 1994.
75. D. J. Wheeler and R. Needham, "TEA, A Tiny Encryption Algorithm," Technical Report 355, "Two Cryptographic Notes," Computer Laboratory, University of Cambridge, Dec 1994, pp. 1-3.

