

# **Using Graphic Based Systems to Improve Cryptographic Algorithms**

ERIN PATRICIA CHAPMAN

Bachelor of Science (University of Auckland, NZ)

A thesis submitted to the graduate faculty of Design and Creative Technologies  
Auckland University of Technology  
in partial fulfilment of the  
requirements for the degree of  
Master of Computer and Information Sciences

School of Engineering, Computer and Mathematical Sciences

Auckland, New Zealand  
2016

## Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a University or other institution of higher learning, except where due acknowledgement is made in the acknowledgements.

A handwritten signature in black ink, appearing to read 'E. Chapman', written over a horizontal dotted line.

Erin Patricia Chapman

## **Acknowledgements**

I would like to thank Auckland University of Technology and my supervisor Dr Brian Cusack for both the opportunity and the support they have provided throughout the process of completing my degree. Dr Cusack's advice and experience has been invaluable in the development of this thesis.

I would also like to thank my family, for being supportive and understanding throughout the relentless insanity that is post-graduate research, especially my mother for her assistance in proof-reading. I would also like to extend many thanks to Johanna Quinn, for her blunt and honest appraisal of a multitude of different drafts and ideas; and Jerina Grewar and Ebony Sparrow, for wading through the final draft with great enthusiasm.

## Abstract

With the ever-expanding use of technology for communications, the demand for strong cryptographic methods is continually growing. The implementation of cryptographic algorithms in modern networked systems is crucial to ensure the security and confidentiality of data. Standardized encryption algorithms have emerged to allow users and developers a quantifiable and thoroughly tested level of security within their systems.

While much research has been done to improve the security of traditional ciphers such as the Advanced Encryption Standard (AES) and the now-defunct Rivest Cipher 4 (RC4), there are opportunities for the development and improvement of alternative ciphers based on graphic methods. Encryption using graphic methods, such as Visual Cryptography (VC) and Elliptic Curve Cryptography (ECC), give high levels of security, and demonstrate alternative approaches to achieve secure methods for the ever-expanding online world.

This thesis proposes an alternative word-oriented symmetric stream cipher based on graphic methods called Coordinate Matrix Encryption (CME), which offers quantifiably high levels of security and a non-singular mapping of plaintext to ciphertext. The focus of this thesis was to explore the security offered by alternative graphic methods, in comparison to traditional classical methods, as well as the difficulties faced in implementing these alternative systems. It is hypothesized that graphic-based methods would offer higher levels of security with lower overheads than classical methods, and that the proposed CME system would prove secure against attack.

The proposed system was implemented in Java along with four comparable algorithms, both graphic-based and traditional, which were AES, RC4, ECC, and VC. The algorithms were all tested for security and efficiency, and the comparative results show the high levels of security achievable by alternative graphic-based ciphers. The resistance of the proposed 8-bit CME system to brute force attacks was shown to be 157,899 orders of magnitude higher than that of a 128-bit key in traditional ciphers such as AES. Examination of the avalanche effect of the CME scheme showed that less than 0.5% of all bytes within the ciphertext remained in the same position when a single bit of the plaintext was altered. While the RC4 scheme offered the best efficiency in terms of time required to encrypt and decrypt the data, the CME scheme had lower memory requirements and was faster in the setup execution.

Further research into alternative graphic methods is required to explore the applications of alternative systems such as CME. The security offered by the proposed

CME scheme makes it an ideal candidate for post-quantum cryptographic research. The system's alternative key structure and non-singular mapping allow for resistance to known and chosen plaintext attacks, and these features require further exploration. Further comparative analysis between traditional and graphic-based ciphers is required to determine whether alternative graphic methods are able to offer higher security for lower overheads. Optimization of the CME scheme requires further testing, to ensure it has competitive advantage, and it is able to be implemented in application development. There is currently little standardisation in stream ciphers to replace RC4, and as such the opportunity exists for an optimized version of CME to assist in this particular space in applications such as TLS that utilize stream ciphers for encryption on a day-to-day basis.

## Table of Contents

<b>Table of Contents .....</b>	<b>v</b>
<b>Table of Figures .....</b>	<b>x</b>
<b>Chapter 1 Introduction .....</b>	<b>1</b>
<b>1.1 MOTIVATION FOR RESEARCH.....</b>	<b>2</b>
<b>1.2 RESEARCH APPROACH AND FINDINGS .....</b>	<b>2</b>
<b>1.3 STRUCTURE OF THESIS.....</b>	<b>3</b>
<b>Chapter 2 Literature Review .....</b>	<b>5</b>
<b>2.0 INTRODUCTION .....</b>	<b>5</b>
<b>2.1 CRYPTOGRAPHY .....</b>	<b>5</b>
2.1.1 Classical Symmetric Cryptography.....	6
2.1.2 Advanced Encryption Standard.....	7
2.1.3 Stream Ciphers and Rivest Cipher 4 .....	8
2.1.4 Asymmetric Cryptography .....	9
<b>2.2 ERROR CORRECTING CODES.....</b>	<b>10</b>
<b>2.3 GROUP THEORY IN CRYPTOGRAPHY .....</b>	<b>11</b>
2.3.1 Rings and Fields.....	12
2.3.2 Matrices and Graphs.....	12
<b>2.4 GRAPHIC METHODS IN CRYPTOGRAPHY .....</b>	<b>14</b>
2.4.1 Cryptography Based on Families of Graphs .....	14
2.4.2 Multivariate Cryptography .....	16
<b>2.5 ELLIPTIC CURVE CRYPTOGRAPHY .....</b>	<b>17</b>
2.5.1 Elliptic Curve Cryptography and RSA.....	17
2.5.2 The ECC Discrete Logarithm Problem .....	18
2.5.3 Applications and Research in ECC .....	19
<b>2.6 VISUAL CRYPTOGRAPHY .....</b>	<b>20</b>
2.6.1 Secret Sharing Schemes .....	21
2.6.2 Extended Visual Cryptography Schemes .....	22
2.6.3 Pixel Expansion and Contrast Constraints .....	23
2.6.4 Random Grid Visual Cryptography Schemes .....	23
2.6.5 Applications and Research in Visual Cryptography .....	24
<b>2.7 ISSUES AND PROBLEMS .....</b>	<b>25</b>
2.7.1 Issues in Elliptic Curve Cryptography .....	25
2.7.2 Issues in Visual Cryptography .....	27

2.7.3 Issues in Graph Based Cryptography .....	28
<b>2.8 CONCLUSIONS .....</b>	<b>29</b>
<b>Chapter 3 Methodology .....</b>	<b>30</b>
<b>3.0 INTRODUCTION .....</b>	<b>30</b>
<b>3.1 REVIEW OF SIMILAR STUDIES .....</b>	<b>30</b>
3.1.1 Jeeva, Palanisamy and Kanagaram (2012).....	30
3.1.2 Afzal, Kausar and Masood (2006) .....	32
3.1.3 Sharma, Garg and Dwivedi (2014).....	33
3.1.4 Kohafi, Turki and Khalid (2003).....	34
3.1.5 Masadeh, Aljawarneh, Turab and Abuerrub (2010).....	35
3.1.6 Thakur and Kumar (2011).....	35
3.1.7 Bhat, Ali and Gupta (2015) .....	36
3.1.8 Prachi, Dewan and Pratibha (2015).....	37
3.1.9 Singhal and Raina (2011).....	38
<b>3.2 RESEARCH DESIGN .....</b>	<b>38</b>
3.2.1 Summary of Similar Studies and Review of the Problems and Issues.....	39
3.2.2 Research Questions and Hypotheses .....	40
3.2.3 Research Phases & Algorithm Implementations .....	42
3.2.4 Coordinate Matrix Encryption Algorithm Design.....	43
<b>3.3 DATA REQUIREMENTS .....</b>	<b>46</b>
3.3.1 Algorithm Testing .....	47
3.3.2 Algorithm Analysis .....	48
3.3.3 Data Presentation.....	49
<b>3.4 LIMITATIONS.....</b>	<b>50</b>
<b>3.5 CONCLUSION .....</b>	<b>52</b>
<b>Chapter 4 Research Findings .....</b>	<b>53</b>
<b>4.0 INTRODUCTION .....</b>	<b>53</b>
<b>4.1 COORDINATE MATRIX ENCRYPTION .....</b>	<b>53</b>
4.1.1 Implementation Details for CME on Binary Strings.....	53
4.1.2 Implementation Details for CME based on Byte Arrays.....	55
4.1.3 Efficiency .....	56
4.1.4 Security .....	58
<b>4.2 ADVANCED ENCRYPTION STANDARD .....</b>	<b>63</b>
4.2.1 Implementation Details .....	64
4.2.2 Efficiency .....	64
4.2.3 Security .....	65
<b>4.3 ELLIPTIC CURVE CRYPTOGRAPHY .....</b>	<b>68</b>

4.3.1 Implementation Details .....	68
4.3.2 Efficiency .....	69
4.3.3 Security .....	69
<b>4.4 VISUAL CRYPTOGRAPHY .....</b>	<b>70</b>
4.4.1 Implementation Details .....	70
4.4.2 Efficiency .....	72
4.4.3 Security .....	73
<b>4.5 RC4 .....</b>	<b>75</b>
4.5.1 Implementation Details .....	75
4.5.2 Efficiency .....	75
4.5.3 Security .....	76
<b>4.6 COMPARATIVE RESULTS .....</b>	<b>79</b>
4.6.1 2-out-of-2 VC versus 4-bit CME.....	79
4.6.2 AES versus 8-bit CME Byte Scheme.....	82
4.6.3 ECC versus 8-bit CME Byte Scheme.....	86
4.6.4 RC4 versus 8-bit CME .....	87
<b>4.7 CONCLUSION .....</b>	<b>91</b>
<b>Chapter 5 Discussion and Analysis of Findings .....</b>	<b>92</b>
<b>5.0 INTRODUCTION .....</b>	<b>92</b>
<b>5.1 RESEARCH QUESTIONS AND HYPOTHESES .....</b>	<b>92</b>
5.1.1 Research Question 1: What are the security benefits of graphic based systems in comparison to classical block ciphers? .....	92
5.1.2 Research Question 2: What difficulties are faced in the implementation of graphic based systems? .....	93
5.1.3 Sub-Questions .....	94
5.1.4 Hypothesis 1: Graphic-based methods provide a better level of security with lower overheads than classical encryption techniques .....	96
5.1.5 Hypothesis 2: The proposed encryption system based around graphic methods is computationally secure against attacks .....	97
<b>5.2 DISCUSSION.....</b>	<b>98</b>
5.2.1 Testing Algorithms.....	98
5.2.2 Benefits and Applications of Graphic Based Ciphers .....	99
5.2.3 Difficulties and Optimizations in Implementation .....	101
<b>5.3 CONCLUSION .....</b>	<b>102</b>
<b>Chapter 6 Conclusion .....</b>	<b>103</b>
<b>6.0 INTRODUCTION .....</b>	<b>103</b>
<b>6.1 LIMITATIONS OF RESEARCH .....</b>	<b>103</b>



6.1.1 Programming Limitations .....	103
6.1.2 Comparing Asymmetric and Symmetric Systems.....	104
6.1.3 Binary Implementation of Visual Cryptography.....	105
<b>6.2 FUTURE RESEARCH.....</b>	<b>105</b>
<b>6.3 CONCLUSION .....</b>	<b>108</b>
<b>References .....</b>	<b>109</b>
<b>Appendix A: Glossary .....</b>	<b>120</b>
<b>Appendix B: Source Code .....</b>	<b>130</b>
<b>B-1: GENERATION OF PSEUDO-RANDOM BINARY STRINGS .....</b>	<b>130</b>
<b>B-2: AES AND RC4 CODE AND ANALYSIS PROGRAMS .....</b>	<b>130</b>
B-2i: AES implementation.....	130
B-2ii: RC4 implementation.....	135
B-2iii: AES/RC4 Frequency Analysis Program.....	139
B-2iv: AES/RC4 Avalanche Effect Program.....	143
B-2v: AES/RC4 Message to binary string conversion.....	144
<b>B-3: ELLIPTIC CURVE IMPLEMENTATION .....</b>	<b>145</b>
B-3i: Generate EC Key .....	145
B-3ii: Complete ECDH protocol.....	146
<b>B-4: VC IMPLEMENTATION.....</b>	<b>147</b>
B-4i: 2-out-of-2 VC Encryption scheme.....	147
B-4ii: VC Avalanche effect.....	151
<b>B-5: CME BYTE IMPLEMENTATION.....</b>	<b>152</b>
B-5i: CME Byte setup and ByteCE classes.....	152
B-5ii: CME Byte code.....	157
<b>B-6: CME STRING IMPLEMENTATION .....</b>	<b>164</b>
B-6i: CME String setup and Entry classes.....	164
B-6ii: CME string code .....	169
<b>B-7: CME ANALYSIS PROGRAMS .....</b>	<b>179</b>
B-7i: Frequency analysis.....	179
B-7ii: Avalanche effect .....	185
B-7iii: CME UTF-8 string to binary conversion.....	188
<b>Appendix C: Testing Data .....</b>	<b>190</b>
<b>C-1: DATA USED IN COMPARISON OF AES, RC4 AND CME.....</b>	<b>190</b>
<b>C-2: DATA USED IN COMPARISON OF CME AND VC (PSEUDORANDOM     BINARY STRINGS).....</b>	<b>191</b>
<b>Appendix D: Example Results .....</b>	<b>193</b>

<b>D-1: EXAMPLE RESULT FROM AES .....</b>	<b>193</b>
<b>D-2: EXAMPLE RESULT FROM RC4.....</b>	<b>193</b>
<b>D-3: EXAMPLE RESULT FROM ECDH .....</b>	<b>193</b>
<b>D-4: EXAMPLE RESULT FROM VC .....</b>	<b>193</b>
<b>D-5: EXAMPLE RESULT FROM BYTE CME .....</b>	<b>194</b>
<b>D-6: EXAMPLE RESULT FROM BIT-STRING CME.....</b>	<b>194</b>

## Table of Figures

Figure 2.1: The AES Encryption Process (Adapted from Stallings, 2014, p.133) .....	8
Figure 2.2: Stream ciphers versus block ciphers. (Martin, 2012, p. 107) .....	9
Figure 2.3: A simple Cayley graph, as described by Equation 2.4.1.ii (Davidoff, Sarnak & Valette, 2003, p. 119).....	15
Figure 3.1: Results from Jeeva et al., 2012, p. 3036 .....	31
Figure 3.2: Phases of research.....	42
Figure 3.3: A randomly generated key matrix for a 3-bit coordinate matrix scheme.....	43
Figure 3.4: Key matrix generation in the Coordinate Matrix Encryption scheme.....	44
Figure 3.5: Example plaintext ciphertext pair output from a 4-bit CME scheme.....	46
Figure 3.6: An example of frequency analysis on a 2-bit coordinate matrix scheme.....	48
Table 4.1: Mean encryption/decryption times for byte CME (3d.p.) .....	56
Table 4.2: Mean encryption/decryption times for 4-bit string CME (3d.p.).....	57
Table 4.3: Mean setup time and memory for byte CME (3d.p.).....	57
Table 4.4: Mean encryption/decryption memory for byte CME (3d.p.).....	57
Table 4.5: Mean setup time and memory for 4-bit string CME (3d.p.).....	58
Table 4.6: Mean encryption/decryption memory required for 4-bit string CME (3d.p.)	58
Table 4.7: Frequency analysis of ciphertext from an 8816-bit string. (3d.p.).....	62
Table 4.8: Frequency analysis of ciphertext from a 4048-bit chosen plaintext string. (3d.p.) .....	62
Table 4.9: Frequency analysis of ciphertext from a 4408-bit string. (3d.p.).....	63
Table 4.10: Avalanche effect in byte CME. (3d.p.).....	63
Table 4.11: Mean encryption/decryption times for 128-bit AES (3d.p.).....	64
Table 4.12: Mean setup time and memory required for 128-bit AES (3d.p.).....	65
Table 4.13: Mean memory required for encryption/decryption in 128-bit AES (3d.p.)	65
Table 4.14: Frequency analysis of ciphertext from a 8144-bit string in 128-bit AES (3d.p.) .....	66
Table 4.15: Frequency analysis of ciphertext from a 4048-bit single character string in 128-bit AES (3d.p.).....	67
Table 4.16: Frequency analysis of ciphertext from a 4408-bit string in 128-bit AES (3d.p.) .....	67
Table 4.17: Avalanche effect in 128-bit AES (3d.p.) .....	68
Table 4.18: Memory and time requirements for execution of ECDH protocol (3d.p.)...	69

Figure 4.1: A visual representation of the six possible subpixel states for the implemented VC scheme. ....	71
Table 4.19: Mean encryption/decryption times in bit-string VC. (3d.p.) .....	72
Table 4.20: Mean encryption/decryption memory requirements in bit-string VC. (3d.p.) .....	72
Table 4.21: Mean setup time and memory requirements in bit-string VC. (3d.p.).....	73
Table 4.22: Avalanche effect in bit-string VC. (3d.p.) .....	74
Table 4.23: Encryption and decryption times in RC4 (3d.p.) .....	76
Table 4.24: Set up requirements for RC4 (3d.p.).....	76
Table 4.25: Memory requirements for RC4 (3d.p.) .....	76
Table 4.26: Frequency analysis of an RC4 encrypted 8144-bit string (3d.p.) .....	77
Table 4.27: Frequency analysis of ciphertext from a 4048-bit chosen plaintext string (3d.p.) .....	78
Table 4.28: Frequency analysis of a 4408-bit string encrypted with RC4. (3d.p.) .....	78
Table 4.29: Avalanche effect in RC4 (3d.p.) .....	79
Table 4.30: Mean encryption and decryption times for differing bit string lengths in the VC and CME schemes. (3d.p.).....	80
Table 4.31: Mean setup for the VC and CME schemes. (3d.p.) .....	80
Table 4.32: Avalanche effect for differing bit string lengths in the VC and CME schemes. (3d.p.) .....	81
Table 4.33: Mean setup requirements for the AES and byte-level CME schemes. (3d.p.) .....	82
Table 4.34: Mean encryption/decryption time for the AES and byte-level CME schemes. (3d.p.) .....	83
Table 4.35: Mean encryption/decryption memory for the AES and byte-level CME schemes. (3d.p.).....	83
Table 4.36: Frequency analysis for 128-bit AES on ciphertext from an 8814-bit string. (3d.p.) .....	84
Table 4.37: Frequency analysis for byte-level CME on ciphertext from an 8814-bit string. (3d.p.) .....	85
Table 4.38: Frequency analysis for 128-bit AES on ciphertext from a 4048-bit chosen plaintext string. (3d.p.).....	85
Table 4.39: Frequency analysis for byte-level CME on ciphertext from a 4048-bit chosen plaintext string. (3d.p.).....	85
Table 4.40: Avalanche effect in 128-bit AES and byte-level CME schemes. (3d.p.) ....	86

Table 4.41: Setup requirements for byte-level CME and ECDH protocols. (3d.p.).....	86
Table 4.42: Comparative set up requirements for RC4 and 8-bit CME (3d.p.).....	87
Table 4.43: RC4 versus 8-bit CME encryption and decryption time requirements (3d.p.) .....	88
Table 4.44: RC4 versus CME memory requirements (3d.p.) .....	88
Table 4.45: Frequency analysis of ciphertext from an 8814-bit string in RC4 (3d.p.) ...	90
Table 4.46: Frequency analysis of ciphertext from an 8814-bit string in 8-bit CME. (3d.p.) .....	90
Table 4.47: Frequency analysis of ciphertext from a 4048-bit chosen plaintext in RC4 (3d.p.).....	90
Table 4.48: Frequency analysis of ciphertext from a 4048-bit chosen plaintext in 8-bit CME (3d.p.) .....	91
Table 4.49: Comparative avalanche effect in RC4 and 8-bit CME (3d.p.).....	91

# Chapter 1

## Introduction

### 1.0 BACKGROUND

The use of cryptography for securing information can be traced back to early human civilisations. Transforming information so as to prevent unauthorized access is a necessity in the digital age. The standardisation of algorithms such as AES (Advanced Encryption Standard) provides for a quantifiable level of security. The ability to rigorously prove the security of a standard algorithm allows users to have confidence in the security of their implementation. It also allows programmers and developers to build around predefined structures for secure systems. Standard algorithms such as AES have undergone many iterations of testing and research to provide the necessary confidence in their security.

Modern symmetric ciphers use a Feistel design. This involves multiple rounds of operations for encrypting blocks of data. These operations include substitutions and transpositions, as well as adding individual round keys. The security of these symmetric ciphers rests on the security of the key, usually a binary string of at least 128 bits. AES gives the option of 128, 192 or 256 bit keys. Due to the rising tide of research into quantum computing, and the introduction of Grover's Algorithm (Grover, 1996), it is now recommended that symmetric encryption systems use keys greater than 128. The effect of quantum computing on security is discussed in chapters 5 and 6.

The current security climate, stoked by events such as the release of Edward Snowden's files from NSA surveillance programs, and the subsequent increase in encryption implementation by firms such as Apple and Facebook, has thrust cryptographic research to the forefront of social consciousness. As such, the demand for better, stronger, faster encryption methods is increasing globally. To meet this demand, new cryptographic algorithms must be developed. On this basis, the research in this thesis revolved around the creation of an alternative symmetric stream cipher called Coordinate Matrix Encryption (CME), using a matrix based key structure. The implemented CME scheme gave a theoretical security to brute force attacks that outstripped the compared standardized algorithms, a more pronounced avalanche effect, and remained

competitively efficient in execution.

## **1.1 MOTIVATION FOR RESEARCH**

The use of encryption in technology underpins the security of modern life. The burgeoning Internet of Things has resulted in a high demand for secure algorithms to protect personal data, such as the integration of asymmetric encryption technologies into banking applications and email, the use of encrypted smart card chips in bank cards and industry access cards, and the need to secure newly networked devices from smartphones to wearables to electric bicycles. As computer technology increases in speed and performance, and radical developments such as Shor's Algorithm threatening the security of current public key systems (Shor, 1994), the importance of and demand for strong cryptography is growing rapidly.

The use of symmetric encryption algorithms such as the industry standard AES (Advanced Encryption Standard) for the security of data has been implemented, and traditional encryption methods built on Feistel cipher design have received numerous improvements and upgrades in recent years. However, the security possibilities proposed by alternative ciphers based on graphic methods, and those that use alternative key structures is under-developed in comparison.

The motivation of this study is to develop and evaluate the possibilities of security and efficiency offered by alternative graphic-based ciphers and key structures. The constant expansion of computing technology requires that researchers continually develop and test new methods of encryption. As such, the realm of ciphers based on graphic-methods and the security offered by alternative key structures such as graphs or polynomial curves is of high importance in cryptography. The strength of alternative key structures, such as the matrices employed in the proposed CME system, is in the dramatically increased key space, which is discussed in Chapters 3, 4 and 5. The size of the key space, and resistance to traditional attacks makes alternative key structures, such as those proposed in the CME scheme, a highly attractive prospect for future research and implementation.

## **1.2 RESEARCH APPROACH AND FINDINGS**

The research conducted in this study was performed through the analysis of the efficiency and security of four well-developed and researched algorithms (Visual Cryptography,

Elliptic Curve Cryptography, Rivest Cipher 4, and the Advanced Encryption Standard), as well as the proposed CME system. The tests were performed over many iterations to provide stable results, and the different algorithms were then compared in pairs. The result of this experimental design suggested that the proposed CME scheme offered a high level of security while remaining comparatively efficient, though more optimisation may be required to ensure a truly competitive design. The study was conducted using Java standard implementations of ECC, RC4 and AES, as well as a string-oriented version of VC specifically developed for the purpose of the experiment. All the algorithms were tested for efficiency and security, with criteria developed based on prior studies and reviewed literature. The memory requirements, the time required at each stage, and the key space were among the testing criteria. For the relevant algorithms, the avalanche effect and the frequency distribution of the ciphertext was also examined.

The research design was developed through the analysis of comparable studies, given in Chapter 3, and current literature, which is evaluated in Chapter 2. The current methodologies of graphic based systems and industry standards for encryption were explored, and recent developments in cryptography were discussed. This research then formed the basis of the research questions and the study design, which utilized both practical and theoretical analysis of the efficiency and security of the algorithms.

### **1.3 STRUCTURE OF THESIS**

The thesis is split into 6 chapters, followed by 4 appendices. The chapter structure is as follows: 1. Introduction; 2. Literature Review; 3. Methodology and Design; 4. Research Findings; 5. Research Discussion; 6. Conclusion. The Appendices are: A. Glossary of Terms; B. Source Code; C. Testing Data; D. Example Results. Prior to the appendices is a list of all texts and materials referenced within the thesis.

Chapter 2: Literature Review explores the current research available in graph-based cryptography, and gives an in depth background for the material contained in the study. It details the current standards for cryptography such as AES and RC4, and gives the mathematical foundations of graphic-based cryptography. The more widely explored graphic-based systems such as ECC and VC are detailed, as are those encryption systems based on multivariable equations and graphs. The history and design of error-correcting codes is also examined.

Chapter 3: Methodology and Design gives an in-depth analysis of similar studies and outlines the research design. Prior comparative algorithm analyses are discussed and



the benefits and limitations of their design are enumerated. The research questions and hypotheses are formulated, and the testing criteria are detailed.

Chapter 4: Research Findings details the results of the study. The results of the tests are given individually for each of the tested algorithms, and then the pairs of comparative algorithms are examined together. The string oriented Coordinate Matrix Encryption (CME) system and Visual Cryptography algorithm are compared, as are the byte-oriented CME scheme and AES; the byte-oriented CME and RC4; and byte-oriented CME and Elliptic Curve Cryptography (ECC) employing the Diffie-Hellman protocol.

Chapter 5: Research Discussion addresses the implications of the study. The research questions are answered based on the results given in Chapter 3, and the hypotheses are redressed given the findings. The difficulties faced in the implementation of the algorithms as well as the benefits and limitations of encryption systems based on graphic methods and alternative key structures are discussed.

Chapter 6: Conclusion enumerates the limitations of the study in design and execution. The ways these limitations may have impacted on the results are detailed. Then the opportunities for further research are explored, and recommendations for future study are given.

## **Chapter 2**

### **Literature Review**

#### **2.0 INTRODUCTION**

The literature review is an in-depth study of the selected elements of cryptography that will impact on this thesis. It has review of the origins and history of graphic based cryptographic methods, the current research undertaken in these areas, and concludes with an analysis of outstanding issues, problems, and unresolved challenges in the research area. The first section details the necessary cryptographic background, and covers current standards as well as classical encryption methods. The second section explains the mathematical theory underlying graphic based cryptography, such as matrices, vectors, fields, rings and groups. Section three reviews the more generic graphic based methods in cryptographic research, including multivariate cryptography and cryptography based on special graph families. Section four details Elliptic Curve Cryptography, a well-studied graphic method which offers an alternative to current asymmetric encryption technologies, while section five outlines Visual Cryptography, an image encryption method based around graph decomposition and matrix operations. Section six explores the issues and problems within each of the detailed graphic based methods, while section seven discusses the conclusions that can be drawn from this research.

#### **2.1 CRYPTOGRAPHY**

The use of cryptography and the encryption of data, provides for secure transmission while maintaining confidentiality and integrity (Chandra, Paira, Alam, & Sanyal, 2014). Modern encryption technologies such as asymmetric encryption like the Advanced Encryption Standard and Rivest, Shamir and Adleman (RSA), as well as security protocols such as WiFi Protected Access (WPA) are in widespread use across the globe, protecting web browsing, home networks, and personal devices. The most widespread

encryption algorithms, such as the Advanced Encryption Standard (AES), are based around classical substitution and transposition techniques. Cryptographic methods can also be broken down into block ciphers versus stream ciphers (Anderson, 2008). Section 2.1.1 gives an overview of symmetric cryptography, followed in section 2.1.2 by a description of the current standard block cipher, AES, the Advanced Encryption Standard. Section 2.1.3 then details the use of stream ciphers, including Rivest Cipher 4 (RC4). Finally section 2.1.4 explains asymmetric cryptography.

### **2.1.1 Classical Symmetric Cryptography**

Classical encryption, such as the early Caesar cipher, uses substitution and transposition methods to scramble a data stream, so as to render it meaningless without the corresponding key. In symmetric encryption, the same key is used to encrypt and decrypt the data. The breaking of symmetric encryption relies on the security and secrecy of the key – the algorithm used for encryption does not need to be kept confidential, as the encryption cannot be reversed without the key (Stallings, 2014). Symmetric encryption is used in most modern technologies, for the bulk of encrypted communication. Currently, encryption is used to secure much of the online world, such as banking transactions, secure email, website logins, and company data. In 2013, over 600 million people were making use of email services (Hosnieh, Martin von, & Christoph, 2013). With so many people utilizing the Internet for communication, the ability to ensure such communications remain private becomes of similar significance to a home owner being able to lock his/her front door. The steady increase in demand for encryption, especially with the surge in growth of the Internet of Things, has given rise to a new wave of cryptographic research. Being able to encrypt the transmissions of devices that link in to the web is of huge importance. Cars, air conditioning units, televisions, and many other household items are now becoming networked. Hence, it is necessary to ensure security, and prevent malicious attackers from manipulating these devices for their own ends. Self-driving cars require security to ensure that a malicious attacker is unable to take over the operating system and alter their functionality. The use and operation of drones adds another layer of importance to the use of encryption, as they are remotely controlled and can be highly weaponized. It is therefore exceedingly important to ensure that the commands received by drones are from a valid and securely-verified source.

Symmetric encryption is a classification of encryption methods based on a shared secret key, and is also known as secret key or shared key encryption. This typically relies on substitution and transposition ciphers. One example of symmetric encryption is the

Caesar cipher, the earliest known encryption algorithm, which shifted the alphabet 3 places to the right. For the Caesar cipher, the key for decryption is the shift – the number of places to the left the cipher alphabet must be moved to result in the plaintext.

$$\text{Equation 2.1.1.i} \quad C = E(k, p) = (p + k) \bmod 26 \quad (\text{Stallings, 2014, p.15})$$

The general equation for encryption in the Caesar shift cipher is shown in Equation 2.1.1.i, where  $p$  is the plaintext letter of the alphabet, and  $k$  is the key shift – a value between 1 and 25.

$$\text{Equation 2.1.1.ii} \quad p = D(k, C) = (C - k) \bmod 26 \quad (\text{Stallings, 2014, p.15})$$

The reversal algorithm for decryption is shown in Equation 2.1.1.ii, where  $C$  is the ciphertext letter. As there are only 25 possible shifts (if shifting to the original position is discounted), the Caesar cipher is not a secure method of encryption (Martin, 2012). The small key space of 25 means that each of the key possibilities can be calculated until the correct key shift is found.

## 2.1.2 Advanced Encryption Standard

Modern symmetric ciphers have significantly higher levels of security than prior classical methods. The Advanced Encryption Standard (AES) is the current standard for data encryption worldwide, and uses multiple rounds of substitutions, transpositions and keys to obfuscate plaintext into ciphertext. AES uses finite field arithmetic, with all operations performed over a finite Galois field  $\mathcal{GF}(2^8)$  (Stallings, 2014). This finite field arithmetic constrains any and all results from operations to within the 256 possible 8-bit bytes. The encryption process is shown in Figure 2.1. AES uses different numbers of rounds depending on the security level of the implementation. 128-bit AES uses 10 rounds, 192-bit has 12 rounds, and 256-bit AES uses 14 rounds. The original key is expanded, resulting in a key word for each round. The encryption process uses 4 transformation operations; substitute bytes, which swaps out the bytes of the current block with those in a predefined matrix; mix columns, which shifts the columns of the current block using modular arithmetic; shift rows, moving all rows within the block; and add round key, which performs a single XOR operation over the current block and round key. Each round makes use of these transformations, and once all rounds are completed, the final ciphertext is output.

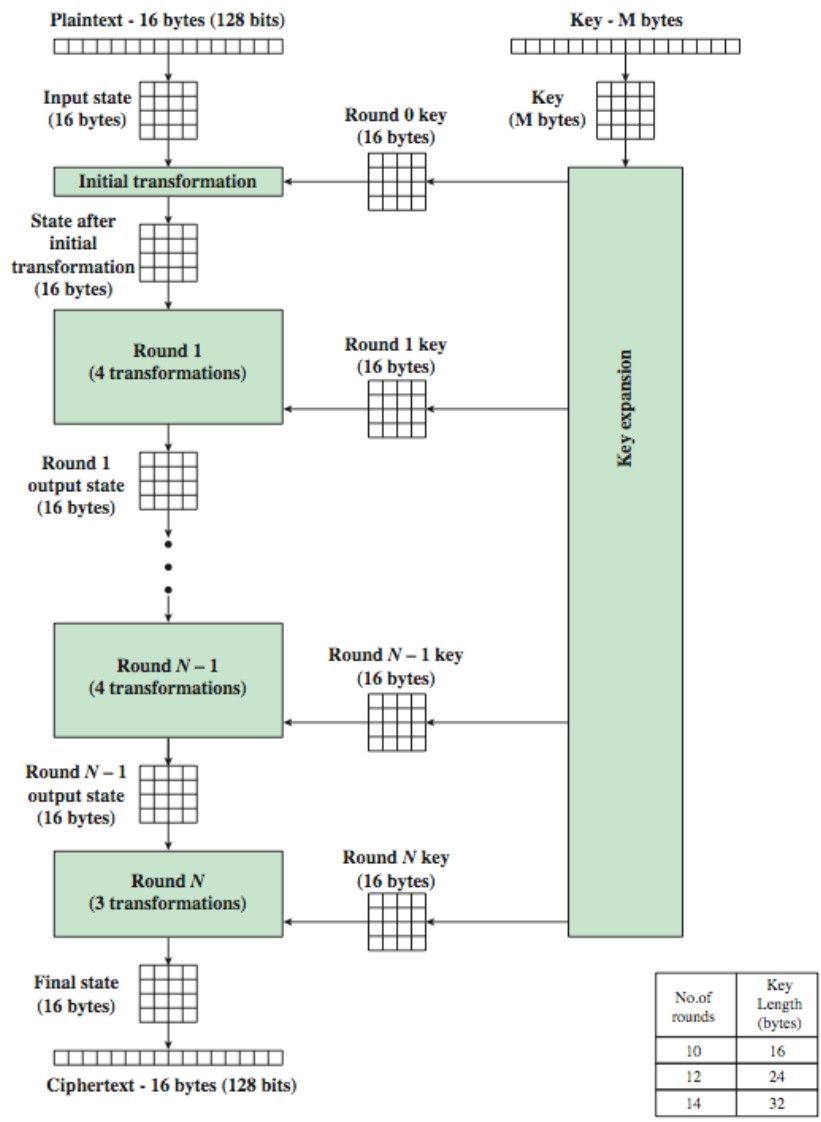


Figure 2.1: The AES Encryption Process (Adapted from Stallings, 2014, p.133)

### 2.1.3 Stream Ciphers and Rivest Cipher 4

The implementation of technologies such as TLS (Transport Layer Security) and SSL (Secure Sockets Layer) for use in website authentication required the use of fast encryption models that operated on streams of data. As such, it was necessary to design stream ciphers, encryption methods that operate on small pieces of the data sequentially. According to Martin (2012), a stream cipher can be described as a variant of the block cipher, which has a designated block size of less than 64 bits. The general model of a stream cipher encrypts data byte by byte, or 8 bits at a time (Stallings, 2014). Figure 2.2 gives a visual comparison of stream versus block ciphers.

Rivest Cipher 4 (RC4), was developed in 1987 by Ron Rivest to address this need for secure stream ciphers in web technologies. While it has since been proven insecure, it was, as of 2014, the most widely implemented stream cipher (Rivest & Schuldt, 2014). RC4 operates by permuting the data using a keystream of up to 256 bytes (2048 bits) and

algorithmic access to a state vector  $S$  which contains all possible 8-bit bytes (Stallings, 2014).

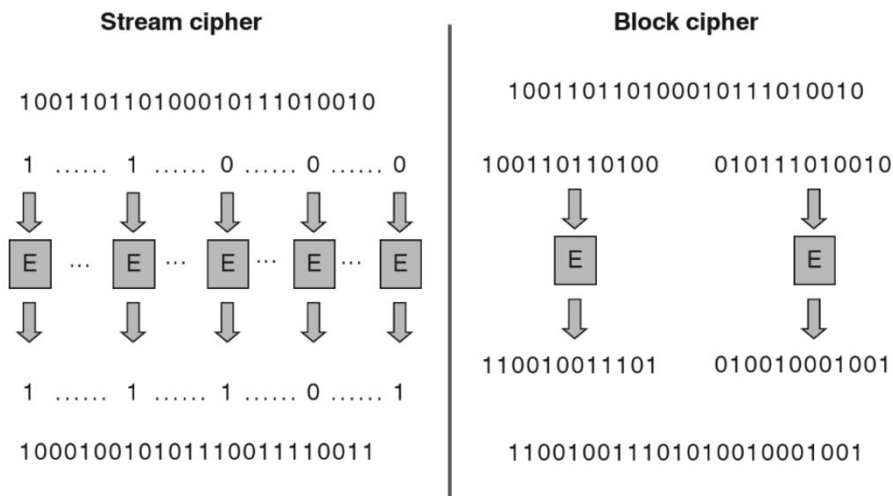


Figure 2.2: Stream ciphers versus block ciphers. (Martin, 2012, p. 107)

Some attacks on RC4 take advantage of the methods with which the session keys are created, shown in Equation 2.1.2.i.

$$\text{Equation 2.1.2.i} \quad s_{\text{session key}} = i_{\text{initialisation vector}} || m_{\text{ain key}} \quad (\text{Klein, 2008, p.1})$$

Key creation as per Equation 2.1.2.i result in predictable behaviours from the session key, and attacks such as the FMS-Attack (Fluhrer, Mantin & Shamir, 2001) take advantage of those behaviours. As yet, no stream cipher that has been standardized and widely adopted to replace RC4. The eSTREAM project was setup specifically for the purpose of standardising a group of new stream ciphers, and was funded and operated through ECRYPT, the European Network of Excellence for Cryptography (Afzal, Kausar & Masood, 2006).

### 2.1.4 Asymmetric Cryptography

Asymmetric encryption, also termed public key encryption, relies on one-way computations for security. Functions such as the computation of prime factors or discrete logarithms are used to provide a one-way trapdoor function that is easy to compute in one direction, but extremely difficult to reverse without all the original information. In this case, each algorithm uses two keys – one public and one private. The public key encrypts the information, but cannot decrypt it. The private key is then used to decrypt the information. Public key systems are often used to securely transmit keys for symmetric encryption, as well as to verify an online identity, such as in a digital signature or

certificate. Digital signatures are an alternative to the physical signature, and give an online option for the verification of an identity. This method requires a way of creating a signature that can be verified by anyone but cannot be forged. The public key/private key system gives an option for this, using a private key that is only known to the user to generate the signature, and a public key that anyone can use to verify it.

RSA - named for the aforementioned Ron Rivest, Adi Shamir and Leonard Adleman - is an example of a public key system, which gives each user a public and private key for verifying and securely transmitting information. The public key is published for use by anyone who wants to be able to communicate securely with the owner of the key. The method of creating these keys relies on a one-way function, so that the private key cannot be computed from the public one. RSA's one-way function is the Integer Factorization Problem or IFP (Yan, 2008). RSA works because the IFP has no known solution that computes in polynomial time or less. One of the requirements of these public key systems is the implementation of a secure key distribution method. These methods require that the user is verified in some manner, to prevent identity theft, as well as making sure that keys can be updated or withdrawn in real time. These distribution methods remain one of the more challenging parts of the implementation of public key systems.

## **2.2 ERROR CORRECTING CODES**

Error correcting codes form a base of study in coding theory. The use of these codes to ensure the correct and accurate communication of information through data transmission was introduced by Hamming (1950). The motivation behind the creation of these codes was the removal of error in data, through the ability to automatically correct any distortions or changes in the transmission. Originally called *systematic codes*, Hamming (1950) posited that binary codewords of a specific length could be used to ensure redundancy in transmission and operation of data. Each codeword was set a binary code of length  $n$ , wherein  $m$  digits were used for information, and the remaining  $k = n - m$  digits provided for the automatic detection and subsequent removal and correction of errors.

Hamming (1950) defined the redundancy levels of codes that were capable of correcting a single error in data as in Eq. 2.2.i. These codes were based on the number of 1 digits in a codeword – the data of the codeword was stored in the first  $n - 1$  bits, and then in the final position a single 1 or 0 bit was added to ensure that the binary word

contained an even number of 1s. Then, if a single bit of data was corrupted, the scheme would detect the error, as there would no longer be an even number of 1 bits.

$$\text{Equation 2.2.i} \quad R = \frac{n}{n-1} = 1 + \frac{1}{n-1} \quad (\text{Hamming, 1950, p.3})$$

The single error detecting code proposed by Hamming (1950) evolved into the parity check, or parity bit. The system only works reliably when  $n$  is constrained and small, so data could be split into many symbols of length  $n - 1$  and a parity bit added for each. This allows the probability of a double-error to be kept consistently low.

The Hamming distance of a two codewords or code symbols is the bits that differ between them in the same position (Shankar, 1997). The calculation of the Hamming distance provides a basis for determining the minimum distance of an error-correcting code, or the minimum Hamming distance between two code symbols within the code. In order to correct up to  $t$  errors, the minimum Hamming distance of a code must be calculated as in Eq. 2.1.ii.

$$\text{Equation 2.2.ii} \quad d_{min} \geq 2t + 1 \quad (\text{Shankar, 1997, p.34})$$

Reed Solomon codes are an alternative error-correcting code to the Hamming code. The Reed Solomon codes operate on bytes, rather than bits, which gives a larger field for operation.

The benefit of error-correcting codes is their ability to eliminate noise from transmissions, and detect and correct errors in data. The use of codewords or symbols with carefully defined Hamming distances enables the efficient correction of errors. Due to their ability to detect changes in the data, error-correcting codes have been proposed as a method of securing data, such as in the creation of digital watermarks (Mehta, Varadharajan, & Nallusamy, 2012). The use of error-correcting codes in digital watermarks has been found to significantly increase their resistance to attacks.

### 2.3 GROUP THEORY IN CRYPTOGRAPHY

Graph theory and group theory comprise many theorems and methods which are of use in fields such as computer science. Groups, rings and fields are especially of use in cryptography, as their unusual topology provides for many different and robust algorithms. A *group* is a tuple, a pair  $(G, *)$ , where  $G$  is a set of objects – for example, the set of all real numbers – and  $*$  is a binary operation performed on  $G$ , which is closed under  $G$  (Loehr, 2014). Groups must satisfy four basic conditions: Closure; associativity;



identity; and inverse. The function  $*$  is closed under  $G$ , meaning that for any  $a, b \in G$  which is used in the function  $a * b$ , the result will also be in  $G$ . The associativity property requires that combining three or more elements of the set with the function will have the same result, regardless of the order of operation. For any  $a, b, c \in G \mid a * (b * c) = (a * b) * c$ . The identity property requires that there be a single element that, when combined with any other element via the function, results in that other, unchanged element.  $\exists e (e, a \in G \mid e * a = a * e = a)$ . The final property is that of the inverse: for every element  $a$ , there must be an element  $a^{-1}$ , which combines with  $a$  to give the identity element.  $\forall a \exists a^{-1} (a, a^{-1} \in G \mid a * a^{-1} = a^{-1} * a = e)$ . Only a pair that satisfies all of the above properties can be considered a group.

### 2.3.1 Rings and Fields

Rings and fields are extensions of groups. They require all the properties of groups, as well as special properties of their own. A *ring* is a triple  $(R, \#, *)$ , a set  $R$  with two binary functions.  $R$  is an *abelian* group under  $\#$ . This means  $(R, \#)$  satisfies all the conditions of a group, as well as being commutative – for any  $a, b \in R \mid a \# b = b \# a$ . Any group that is commutative is known as an abelian group. The second operand  $*$  is required to be closed and associative under  $R$ . The two operations are usually called  $+$  and  $\cdot$ , or addition and multiplication, respectively. A ring that is *commutative* satisfies a further axiom – its multiplication operation is commutative under  $R$ .  $(a, b \in R \mid a \cdot b = b \cdot a)$  (Cohn, 2000). The set of integers, or  $\mathbb{Z}$ , forms a ring under the addition and multiplication operations, and is a commutative ring. Rings are also formed by the set of rational numbers ( $\mathbb{R}$ ), and the set of natural numbers ( $\mathbb{N}$ ).

A *field* is a further extension of a ring. If a ring is commutative, unital, contains no zero divisors, and each non-zero element of the ring is a unit, then that ring is also a field. A field has 4 binary operations – as well as the addition and multiplication they inherit from rings, they have two inverse functions for these (Cohn, 2000). The inverse of addition is defined as subtraction, and the inverse of multiplication is the division function. So a field  $F$  would be  $(F, +, \cdot, -, \div)$ . Fields can be finite, or infinite. For example, the set of all rational numbers ( $\mathbb{Q}$ ) is an infinite field. A particular set  $\mathbb{Z}_p$  is a finite field if  $p$  is prime. Systems such as Elliptic Curve Cryptography are concerned with transformations over finite fields.

### 2.3.2 Matrices and Graphs

Another area of relevance in computing from graphic methods is matrix theory. A matrix is an array of numbers. These are the matrix entries, also simply referred to as entries. Matrices are used in cryptographic methods such as secret sharing schemes to encode shares of information. These schemes rely on matrix operations and representations to scramble, expand and then encode the data. Matrices are also used to organize information about groups, rings and fields, such as a Cayley table, which displays the result of the binary operation on each combination of elements in the set. Implementations of graphs with encryption schemes can utilize matrices to represent vertices and edges. Matrix operations, such as matrix multiplication, are used in systems such as Visual Cryptography. Matrix multiplication is of particular use because it is not commutative – the order in which the matrices are multiplied affects the outcome.

Families of matrices such as Hadamard matrices are used in the generation of error-correcting codes. Hadamard matrices are defined as an “ $n$  by  $n$  matrix  $H$  with entries  $+1$  or  $-1$  such that  $HH^T = nI[1]$ ” (Chan-Hyoung, Hong-Yeop & Kyu Tae, 1998, p. 117). All rows within a Hadamard matrix are mutually orthogonal. Hadamard matrices also give rise to Sylvester and Walsh matrices (Giorgobiani, Kvaratskhelia, & Menteshashvili, 2015). A Hadamard matrix produces Hadamard codes, which provide high levels of error-correcting ability. Given a Hadamard matrix  $H_n$ , of size  $2^n$  by  $2^n$ , a Hadamard code can be created which gives a Hamming distance of  $2^{n-1}$ , and is capable of detecting  $[2^{n-1} - 1]$  errors (Pal, 2007). While transmission of data using these codes requires a higher number of bits, they provide for very good error detection and correction, which is of particular use in noisy networks.

Cryptography makes use of finite, regular graphs, due to the usefulness of their underlying structures. These families, such as Cayley graphs, are a connected and secure structure on which to base algorithms for encryption. A graph  $G$  with a finite set of vertices and edges is defined as a triple  $G = (V, E, \phi)$ , such that  $V$  is the set of vertices,  $E$  the set of edges connecting those vertices and  $\phi$  is the function that maps two vertices into an edge (Agnarsson & Greenlaw, 2007). In this way it can be thought of as an extension of the original group, where edges are a connection formed by the operation of combining some two members of the set  $V$ . A graph’s *degree* is the highest number of edges connecting a vertex to those adjacent to it. For example, a regular graph with degree 2 means that each vertex will be adjacent to exactly two other vertices. Special families of regular, undirected graphs, like Cayley, expander or Ramanujan graphs are of particular interest in encryption schemes. As these graphs are large and undirected they can be constructed to ensure a high level of security in algorithms based around graph

walks, special graph colourings, or those that use the Discrete Logarithm Problem to provide intractable encryption.

## 2.4 GRAPHIC METHODS IN CRYPTOGRAPHY

This section explores the graphic methods applied in cryptography, and relates the necessary background for the research of these methods. Section 2.4.1 discusses cryptography based around graph families. Section 2.4.2 then explores cryptography using systems of multivariate equations.

### 2.4.1 Cryptography Based on Families of Graphs

Graphic based systems rely on group theory and graph theory to create secure algorithms for encryption. Some of the more popular graphic based methods are Elliptic Curve Cryptography (ECC) and Visual Cryptography (VC). However, there are other algorithms that take advantage of the innate properties of group theory and families of graphs. These proposed graphic methods for encryption exploit particular traits of certain types of graphs, such as those using families of graphs of large girth, like Cayley graphs (Ustimenko, 2007). A Cayley graph is defined as a graph  $\mathcal{G}(G, S)$  where  $S$  is a non-empty subgroup of the group  $G$ , such that  $S$  is equal to its own inverse ( $S = S^{-1}$ ), and the set of vertices is equal to  $G$ ,  $V = G$ , and the set of edge elements is as follows:

$$\text{Equation 2.4.1.i} \quad E = \{\{x, y\} : x, y \in G; \exists s \in S : y = xs\}$$

(Davidoff, Sarnak, & Valette, 2003, p.108)

A Cayley graph constructed in the manner described by equation 2.4.1.i is a regular graph, but it is necessary to note that not all regular graphs are also Cayley graphs. Cayley graphs are also undirected. These underlying algebraic structures of the family of Cayley graphs can be exploited for use in encryption. Of particular relevance to the field is the quality of expansion in these graphs – the search for expander families of optimal growth. The growth rate of a graph relates to its diameter, and is generally a function of the number of nodes or vertices in the graph (Krebs & Shaheen, 2011).

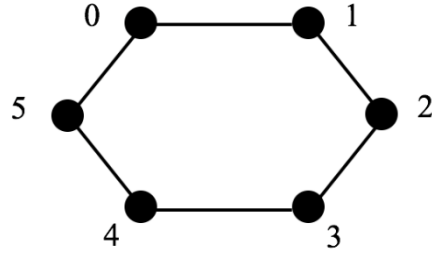


Figure 2.3: A simple Cayley graph, as described by Equation 2.4.1.ii (Davidoff, Sarnak & Valette, 2003, p. 119)

Equation 2.4.1.ii

$$G = \frac{\mathbb{Z}}{6\mathbb{Z}}, S = \{1, -1\}$$

(Davidoff, Sarnak & Valette, 2003, p. 119)

Another family of graphs that are a possible route for cryptographic research is the family of directed graphs of large girth. The fact that there are only three families of undirected graphs of arbitrarily large girth limits their use, however there are infinite numbers of algebraically constructed families of directed graphs of large girth. These can be converted to equivalent Turing machines of basic construction, as a basic finite automaton is equitable to a directed graph, if the memory component is set aside. The arrows on this directed graph can then be labelled with colours as is required according to the automaton's alphabet. These graphs are part of the expander family of graphs (Ustimenko & Romańczuk, 2013). Cayley graphs can be used to describe a linear automata, while other graph families can be used to result in non-linear systems. Encryption over directed graphs uses finite fields to calculate the arithmetic operations.

Encryption systems based around groups of graphs such as Cayley or expander families use sequences of vertices or graph-colourings to create a ciphertext. Others opt for using strongly regular graphs to generate a Hadamard matrix for encoding images (Priyadarsini & Ayyagari, 2013). Some systems use the vertices to represent the plaintext space and the path within the graph becomes the password (Priyadarsini, 2015). Systems such as these based around walks along graph edges can be used in the construction of stream ciphers (Ustimenko, 2014). Some of these graph based systems are also reliant on the intractability of the DLP, and ensure that the groups or rings they are based around are of sufficiently large girth to make the DLP *NP*-complete (Klisowski & Ustimenko, 2010). Expander graphs are also of particular interest in cryptography. These graphs are sparse, finite, and highly connected. Ramanujan graphs are a particular brand of expander graphs that are of use for encryption. Expander graphs were drawn from the study of Cayley graphs (Polak & Ustimenko, 2013).

## 2.4.2 Multivariate Cryptography

Systems have been proposed that utilise group theory and rings to create encryption that relies on the combining of two group elements. Elliptic Curve Cryptography (ECC) transports the classic Discrete Logarithm Problem onto an elliptic curve or graph-based encryption and the reversal of this process is computationally infeasible without the original units involved (Hurley & Hurley, 2011). Public key cryptosystems based around commutative rings also use a variant of the Diffie-Hellman problem to secure their protocols (Kotorowicz, Romanczuk, & Ustimenko, 2011). Multivariate cryptography is the set of cryptosystems which use polynomials and finite commutative rings for encryption, and these are part of the post-quantum cryptography movement. Post-quantum cryptography involves systems that are theoretically resistant to Quantum attacks (Ustimenko, 2014).

Graphic based cryptographic research revolves around increasing efficiency and security. Graphic based methods are based around a “significant demand... for new non-standard cryptographic methods” (Paszkievicz et al., 2001, p. 1) ECC, VC and other graphic methods are currently being researched and expanded as this demand grows. Current research as applies to general graph based methods has focused on different families of graphs – such as expander graphs, which are very highly connected but have few nodes (Polak & Ustimenko, 2013). The implementation options for programming graphs are also a topic of research, with current methods using lists and matrices.

Another development in graphic methods has been the implementation of algebraic geometry into the field of multivariate public key cryptosystems. These are based around a set of multivariate quadratic polynomial equations over a finite field (Ding & Yang, 2009). Further to this, there has been study into parameterized matrices for systems of paraunitary equations for encryption. Multivariate polynomials are a solution to the problems of RSA and an alternative to systems like ECC, using multivariate systems of equations over small fields, such as  $GF(2^m)$  where  $m$  is some small number (Delgoshia & Fekri, 2006). The use of multivariate polynomials is a proposed solution to the issues with key size and set up time, both of which are high in computational complexity and require large amounts of data to communicate. Multivariate systems generally use quadratic polynomial fields. The multivariate systems rely on their own version of the one-way problem, in this case called the MQ problem, based on the computational complexity of solving many different quadratic equations over multiple different fields using many different variables. The complexity of the MQ problem has

led to them being proposed as a possible quantum-resistant encryption method (Liu, Han & Wang, 2011).

## **2.5 ELLIPTIC CURVE CRYPTOGRAPHY**

This section reviews Elliptic Curve Cryptography (ECC) and its applications. Section 2.5.1 gives an introduction to ECC and how it relates to the prior standards in public key systems. Section 2.5.2 explores the Elliptic Curve Discrete Logarithm Problem. Section 2.5.3 then discusses the current trends in research and the applications of ECC.

### **2.5.1 Elliptic Curve Cryptography and RSA**

Elliptic Curve Cryptography (ECC) is a proposed alternative to the public key system RSA, as it provides equivalent security with smaller key sizes and lower overheads (Stallings, 2014). It was intended as a method of transferring the public key discrete logarithm problem into a system which would allow for more efficient computation without loss of security (Miller, 1985). The constant acceleration of computational power has resulted in RSA being considered less than secure in some situations due to its key length, and the high overheads encountered in increasing that key length (Bai, Zhang, Jiang, & Lu, 2012). In fact, in 2003 RSA using a 576 bit key was successfully broken over a three-month time span, further cementing its declining level of security due to its reliance on computational complexity (Ontiveros, Soto, & Carrasco, 2006). An RSA key size of 1024 bits is equivalent to a 163 bits in ECC. The larger the RSA key size the smaller the ratio of the ECC equivalent key, for example a 256 bit ECC key is equivalent to a 3072 bit key in RSA (Pateriya & Vasudevan, 2011). Because ECC based systems are able to provide far smaller key length without sacrificing security, they have become a more attractive option than RSA, which is of much higher computational cost especially in environments of low computing power.

ECC uses transformations over one of two types of field: a finite Galois field  $G(F_p)$ , where  $p$  is a large prime, or a finite field of characteristic 2, also known as a binary field, notated as  $GF(2^m)$  (Bai et al., 2012). A cyclic group, such as those used in ECC, is considered to be appropriate for the implementation of a discrete logarithm based system if it satisfies the following: the entries of a group require minimal representation; the binary operation performed on the group is efficient; and the DLP within the group remains intractable (Galbraith & Menezes, 2005). The elliptic curve consists of several elements: it has a series of rational points, which form the entries of the set within the

group; there is also an element that is a special point at infinity – called point  $O$  – which is also known as the identity element (Ye & Liu, 2011). This set that forms the basis of the field is formed by the solutions to the following equation:

$$\begin{aligned} \text{Equation 2.5.1.i} \quad & (x, y) \in K^2 \\ & y^2 = x^3 + ax + b \end{aligned}$$

where  $a, b \in K$  (Koblitz, 1987, p.1).

A cryptographically strong elliptic curve is one that is non-singular (Kamarulhaili, 2010). In other words, the roots of the polynomial of the curve must be unique. Elliptic curves can be represented in many different ways, including as coordinate systems. Computation of these curves can be made more efficient by the use of different kinds of coordinate systems. Following the optimization of the coordinate system it is possible to mix several different coordinate systems to improve the computational time even more and further optimize the algorithm (Setiadi, Kistijantoro, & Miyaji, 2015). The basis of the ECC algorithm involves the encoding of a message or plaintext onto a point of the chosen curve for encryption (Singh & Debbarma, 2014). The point used is taken from the group of rational points which form the series of the curve. Each point of the set corresponds to a different part of the plaintext message. This can be done for alphabet characters using a code table which corresponds to points and for binary implementations can be used even to encrypt images.

### 2.5.2 The ECC Discrete Logarithm Problem

Public key cryptography systems, such as ECC, rely on the intractability of the discrete logarithm problem (Galbraith & Menezes, 2005). The Discrete Logarithm Problem (DLP) is the one-way property of computing logarithms. The one-way property, or trapdoor function, is that they are easy to compute in one direction, but hard to reverse without the information used in the original computation. This basis, which forms the set of public key cryptography systems, relies on computational complexity for security. The DLP, as defined for any finite cyclic group  $G$ , is as follows:

$$\text{Equation 2.5.2.i} \quad f, g \in G : \exists y (f^y = g)$$

such that  $y$  is the smallest possible positive integer that satisfies equation 2.5.2.i (Polak, Romańczuk, Ustimenko, & Wróblewska, 2013). This problem, originally called the Diffie-Hellman problem and used in traditional public key cryptography systems, was ported to the domain of Elliptic Curves to increase security, and this version is known as the Elliptic Curve DLP, which uses scalars and point multiplication.

ECC is uses scalar multiplication to compute the one-way function that results in

the elliptic curve DLP, using point  $P$  from the set of points, and a scalar multiplier  $k$ .

$$\text{Equation 2.5.2.ii} \quad k.P = Q$$

This operation provides another point on the curve. The scalar multiplication operation is fairly simple to compute, however reversing it – computing  $k$  where only  $Q$  and  $P$  are known – is not currently feasible in less than exponential time, as it is the brute force equivalent of searching through all possible multiples until a common point between  $Q$  and  $P$  is found, and in application  $k$  should be large enough to make this computationally infeasible. As such, the computational complexity of the elliptic curve DLP is where the security of ECC lies (Amara & Siad, 2011). Simplifying the point multiplication operation is one of the optimization goals of ECC algorithm research, as it is the most expensive part of the algorithm (Sutter, Deschamps, & Imana, 2013). This point multiplication can be implemented as a multiplication in the software or hardware, or broken down into other operations, such as modular functions of addition and multiplication, which are lower level computations (Qu & Hu, 2010). Using broken down modular operations makes the scalar multiplication less expensive.

### **2.5.3 Applications and Research in ECC**

Elliptic curve cryptography has been transplanted into protocols for Diffie-Hellman key exchange, and other researchers have looked at the introduction of text-based encryption systems using ECC (Vigila & Muneeswaran, 2009), which have proven to have very high levels of security against brute force attacks. There has been research into different algorithms for utilizing the security of ECC, for example implementing matrix scrambling to improve the overall security against current attacks. Matrix scrambling in ECC uses circular queues to shift the text in random patterns (Amounas & Kinani, 2012). The matrix-scrambling technique adds cycles of encryption, which ensure the plaintext is encrypted differently each time, and as such helps to protect against cryptanalysis. ECC has also been implemented as an authentication setup in smartphones and similar devices using QR (Quick Response) codes to secure their online activity. QR codes are two dimensional matrix barcodes, and due to their prevalence on mobile platforms they are an effective option for generating and securing one-time passcodes (Thiranan et al., 2014). ECC can also be utilized in e-commerce, as the creation of digital signatures is central to each step of the process in SET (Secure Electronic Transactions) protocols (Xia, 2012). Because the signatures are created multiple times, the use of ECC is more efficient than methods such as RSA, as it lessens the load incurred by the processing application. ECC has also been successfully used to encrypt multimedia imagery during compression,



where it has been proven to be efficient in encrypting the imagery without affecting the overall compression algorithm's efficiency. However, the compression and encryption process does result in a degradation of clarity in the final recovered image, in varying levels (Tawalbeh, Mowafi, & Aljoby, 2013).

Elliptic Curve Cryptography is of particular interest in systems that operate on limited resources – such as smart cards and other embedded systems (Targhetta, Owen, Israel, & Gratz, 2015). These systems require efficient implementations, particularly in regards to the more complex ECC operations, such as the scalar multiplication of the curve points which is one of the more expensive to perform. Some recent research has focused on finding a way to improve computation time for this operation, as the reduction of computational time for this part of the algorithm increases the overall efficiency of the implementation (Leca & Rincu, 2014). One of the key ways to decrease the complexity of this operation is the reduction of the Hamming weight of the scalar value. This can be done through a conversion to binary numbers to improve the efficiency of the scalar multiplication (Akhter, 2015). There has also been interest in utilizing ECC algorithms for wireless sensor networks, due to the limited computing power in the individual connected nodes, which prevents the implementation of traditional public key architecture as there cannot be a single trusted public key authority, as well as the difficulty of performing the high cost operations in RSA (Modares, Moravejosharieh, & Salleh, 2011). In situations like sensor networks, ECC provides an advantage because of its lower computational costs, allowing implementation in low-level hardware (Deligiannidis, 2015). Utilizing the set of shifting primes as basis for the curves can also increase the efficiency of the algorithm. This enables the use of multiplication operations without requiring the use of any multiplier function, instead implementing addition and shifts to the same result, which is far more practical for low-cost hardware. Simple embedded systems do not always have a hardware implementation for multiplication, thus making this method of ECC highly attractive as a security option (Marin, Jara, & Skarmeta, 2012).

## **2.6 VISUAL CRYPTOGRAPHY**

This section describes Visual Cryptography (VC), and the ways in which it can be applied to current technologies. Section 2.6.1 gives an overview of the secret sharing schemes VC is based on and the original proposed VC methodology. Section 2.6.2 discusses extended VC schemes, and 2.6.3 explores the issues of pixel expansion and contrast

constraints. Section 2.6.4 discusses the advances made in Random Grid VC, and 2.6.5 then describes current research and applications in VC.

### **2.6.1 Secret Sharing Schemes**

Visual Cryptography (VC) is a popular graphic method for encrypting images, though it is generally a less effective graphic-based system than ECC, due to computational complexity, and overheads, as well as being less applicable to general encryption problems. VC is a set of secret sharing schemes that divide a secret image into  $n$  parts, called shares, of “random binary patterns” (Zhi, Arce & Di Crescenzo, 2006, p. 2441), that only reveal the original image when all shares are superimposed upon one another. However it encounters difficulties due to the high overhead incurred by pixel expansion; which is the number of subpixels required to create a pixel that will encode the share (Hajiabolhassan & Cheraghi, 2010). Much of the research conducted into VC has gone into the issue of minimizing pixel expansion, which is usually directly affected by the number of nodes in the scheme (Blundo, Ciamato, & De Santis, 2006) but it is yet to gain wide application use. Research into visual cryptography schemes has also begun to expand to colour images (Liu, Wu & Lin, 2008), and into integrating visual cryptography into authentication methods (Jaya, Malik, Aggarwal, & Sardana, 2011). Another option proposed for minimizing pixel expansion is step construction, which uses a recursive implementation to create several shares for a single participant (Liu, Wu & Lin, 2010).

Secret sharing schemes are designed to ensure a single secret can be securely shared between a specific group of users. It guarantees that only pre-agreed subsets of those users are able to access the information (Blakley & Kabatiansky, 2011). Each scheme has a dealer, who creates and distributes the shares from the scheme, as well as the user set, who each receive a single share. The shares are each a subset of the original secret. The dealer uses a predefined algorithm to split the secret into shares, which have to be recombined to recreate the secret. Secret sharing schemes operate on a specific definition of perfect secrecy: that a scheme that does not reveal any information about the original secret without all the required subsets is considered perfectly secure. This forms the basis of visual cryptography schemes, which split images into secure shares (Naor & Shamir, 1995). One of the main tenants of the original visual cryptography schemes was that they are able to be decrypted without assistance of a computer, or any specialized cryptographic knowledge (Naor & Shamir, 1995). This makes it an attractive option for instances when more complicated systems are not feasible. The encoded subpixels appear as either black or white to the human visual system, when all necessary

shares are layered over one another (Droste, 1996). The lowering of the contrast – such as in schemes that optimise pixel expansion – make it more difficult to visually decode.

### **2.6.2 Extended Visual Cryptography Schemes**

Further alterations to the original VC schemes have been proposed – called Extended VC schemes or EVCS – which encode the shares into target images, in order to ensure that they appear innocuous (Ateniese, Blundo, Santis, & Stinson, 2001). Using a target image implements a layer of steganography over the encryption, hiding the fact that the image contains a secret share of a message. While this addition of stenography does increase the security of the system, it also heightens the computational requirements of the implementation and the pixel expansion of the scheme overall, requiring each subpixel be encoded to match two separate contrast constraints, one to securely encode the secret image and another to ensure that it matches the target image (Liu & Wu, 2011). EVCS can also enable multiple secrets to be shared between different accepted parties (Klein & Wessler, 2007). With the steganography within an EVCS it is also possible to use a chaotic map to generate one of the pair of shares to improve the security of the scheme. Using a chaotic map to generate half of each pair increases its resistance to cryptanalytic attacks (Mostaghim & Boostani, 2014). The encoded shares can also be designed as circles, which enables multiple secrets to be encoded by the different rotations of the shares (Shyu, Huang, Lee, Wang, & Chen, 2007). Implementing circular shares means that a single pair of shares can encode more than one secret, and requires that the users have knowledge of which rotations of each circle are required to decode each secret image.

Further extension of visual cryptography schemes has resulted in graph-based EVCS (GEVCS). This uses a graph-based substructure, in which there are multiple pairs, each of which is able to recover a secret unique to that pair. This means each share encodes subsets for every pair it is part of and the shares are denoted by a node in the graph. If there exists a vertex connecting two nodes, then there is a secret shared between them. Combining the two shares will reveal the secret denoted by the edge (Lu, Manchala, & Ostrovsky, 2011). If a graph is complete, or fully connected, then every node shares a secret with every other node. This is not always the case, so before combining shares it is necessary to check that the nodes in question are connected by an edge. This means the creation process for each share is much more complex, as there are multiple target images for each source image. Other GEVCS have been proposed, which require that each pair of shares be combined at specific angles to encrypt and decrypt the shares (Feng, Wu,

Tsai, Chang, & Chu, 2008).

Because of how complex the creation of shares in a GEVCS is, the main graph of the scheme can be decomposed into subgraphs. Decomposing the graph allows each share to be created in a smaller scale, which decreases the overall computational complexity of the process. A share matrix is then created for recombining the separate subgraphs. This involves creating a share matrix for each subgraph, padding it to ensure they are all of even length using extra one bits, then concatenating the matrices. Once the concatenation is complete, the result is a complete share matrix for the overall graph.

### **2.6.3 Pixel Expansion and Contrast Constraints**

Graph based extended visual cryptography uses matrix operations to encode the shares of information from an original plaintext image. These operations generate the format for subpixels, and contain contrast blocks for both the source and the target. The operations involve the calculation of the number of subpixels in an image that are to be black – controlling the image’s contrast. The contrast constraint also allows the share to be generated so as to satisfy a particular contrast constraint, or how dark or light a pixel needs to be in order to be considered ‘black’ or ‘white’. This controls the level of clarity in the final image once all shares are combined, and as such, every VC scheme attempts to maximise the contrast as much as is possible (Liu, Wu & Lin, 2010). One of the issues with pixel expansion and contrast is that they cannot both be optimal at once. Each algorithm is required to make a trade-off between pixel expansion and clarity of contrast (Arumugam, Lakshmanan, & Nagar, 2013). There have been many proposed systems to minimize the pixel expansion, and therefore increase the efficiency of the encoding algorithm. Many proposed schemes require polynomial pixel expansion. In most, the expansion is directly affected by the number of shares (Blundo et al., 2006). Those that are attempting to optimize expansion have had success in constraining the pixel expansion, however this comes with the trade-off of lowered contrast. Researchers have managed to constrain pixel expansion to  $O(\log n)$  time in some schemes (Ateniese, Blundo, De Santis, & Stinson, 1996), while systems based around extended graph structures have provided constant expansion (Lu et al., 2011).

### **2.6.4 Random Grid Visual Cryptography Schemes**

A technique known as Random Grid VC (RGVC) has been proposed by researchers as a counter to traditional, deterministic VC - as well as probabilistic VC - as a way to control pixel expansion. Probabilistic VC differs from classical, deterministic VC in that it uses

a binary basis matrix to select whether a given pixel is black or white given equal probability. In the traditional RGVC scheme, each pixel's likelihood of being either black or white in a particular share is decided by a random coin toss style operation, and the pixels are individually considered to be grids (Kafri & Keren, 1988). The trade-off for the lack of pixel expansion in the RGVC scheme is the light transmission. The light transmission is the amount of light that is capable of diffusing through the stacked transparencies of the shares – the overall contrast of the final decoded image. Because the light transmission of shares in an RGVC scheme is automatically  $\frac{1}{2}$ , as approximately 50% of all pixels are black in each share, the overall quality of recovered images is significantly impacted (Hou, Wei, & Lin, 2014). To counter this degradation of the images, generalized RGVC was introduced to create an adjustable light transmission. (Wu & Sun, 2013). Generalized RGVC gives an adjustable probability for the likelihood that a given pixel in a particular share will be white.

Further extensions to RGVC schemes have been proposed, such as common share RGVC. In a common share RGVC scheme involves a scheme that has a set of shares for participants, plus a single key share which is the same for all secrets, and must therefore be kept secure, as would be a user's private key in a public key system (Joseph & Ramesh, 2015). The individual shares are constructed using a random grid algorithm, where the first share is created randomly, and the preliminary second share is created based on that first share and the secret image. This set of steps is then iterated over the preliminary share two to create the final share two and the preliminary share three, over the preliminary share three to create the final share three and the preliminary share four, and so on and so forth. The original randomly generated share becomes the key image for all shares. The result of this algorithm is a VC scheme that is asymmetric, rather than classically symmetric as in probabilistic VC. However, it has also been shown that a regular RGVC scheme can be converted to an equivalent classical VC scheme and vice versa, as there exists a strict equivalence relation between the two types (De Prisco & De Santis, 2014). The relation between these types means that it is possible to use research and findings in both types of scheme.

### **2.6.5 Applications and Research in Visual Cryptography**

Current research has looked at the possible implementation of VC algorithms into fields which require high levels of security in image related data, such as biometrics stores of data for facial recognition, or fingerprint verification (Ross & Othman, 2011). As VC methods involve splitting the secret image into shares, there can then be a private and

public database of biometric images, and only once the shares from the two are combined will the original biometric data be of use. The original image can also be decomposed into shares using target images of other biometric data, benefitting from the extra layer of steganography provided by such EVCS.

The expansion of VC schemes into the arena of colour imagery is a current area of development in research, as the addition of colour images increases the overall complexity and pixel expansion of the schemes. As such, classical VC methods cannot be used for this purpose. Schemes have been proposed that utilize half-toning methods on colour images to simplify the process of encryption. Error diffusion has previously been used to half-tone a grayscale image for VC schemes, and has been further extended to allow for implementation in schemes that use colour images. In colour half-toning, the process is applied to the different channels of colour individually (Kang, Arce, & Lee, 2011). However while the resulting images are recognizable, a side-effect of the half-toning process is a degradation of image quality, for both the shares and the decoded secret image, as it introduces noise and therefore lowers the overall image contrast.

It is also possible to add supplementary material to VC shares, using a method introduced as tagged VC (Wang & Hsu, 2011). In this manner, the shares can be folded to give extra information to the participants, as each contains the secret image, plus a set tag image for each share that is generated in the scheme. As a result, each individual tagged share can then be folded to reveal the tag image. Such a scheme can be extended to allow for multiple folding operations to occur, and for shares to be folded at different angles. The addition of tags to a VC scheme can also allow for an extra layer of security, wherein the tags can contain a particular security message to guard against forged shares, and assist in cheating prevention. These tagged schemes have also been extended to create a system in which the VC scheme is lossless – that is, there is no difference in visual quality between the original and the decoded secret image (Wang, Pei & Li, 2014).

## **2.7 ISSUES AND PROBLEMS**

This section describes the current issues and problems facing cryptography. Section 2.7.1 looks at current problems in ECC, while 2.7.2 discusses issues facing VC. Section 2.7.3 then explores difficulties in graph based cryptography.

### **2.7.1 Issues in Elliptic Curve Cryptography**

The security of ECC relies on computational complexity – the assurance that it is intractable to compute the Elliptic Curve Discrete Logarithm Problem. This reliance means that the security would be severely compromised should the ever-increasing speed of technology provide a method of computing the solution to the Elliptic Curve Discrete Logarithm Problem in less than the current exponential time. On the realization of quantum computers, the Elliptic Curve Discrete Logarithm problem will no longer be computationally infeasible to compute (Krämer, 2015). The weakness surrounding ECC in a post-quantum world is based on Shor’s algorithm (Shor, 1994), operating on a quantum computer, which is capable of solving problems such as discrete logarithms in polynomial time (Ding, Petzoldt, & Wang, 2014). Aside from the possibility of breaking the Discrete Logarithm Problem, ECC also has disadvantages in its implementation. It is highly complex to implement, and the resulting ciphertext message is increased in length from the original plaintext (Chandra et al., 2014).

Advances in fields such as index calculus and number-field sieves have shown possible weaknesses in systems based around the problem of computing discrete logarithms (Joux & Vitse, 2012). Index calculus, a method of computing discrete logarithms using probability and field arithmetic, has been used by mathematicians to exploit characteristics of groups and to then solve the original discrete logarithm problem in sub-exponential time (Miller, 1985). While classic index calculus has not been implemented successfully against general ECC systems, and exponential time square root attacks are more efficient against these general ECC algorithms (Silverman & Suzuki, 1998), the reduction in computing time for solving the discrete logarithm problem in other systems may suggest weakness in the overall computational complexity of DLP-based systems. Further, for some special families of elliptic curves, it is possible to transpose the curve into a field where index calculus is then an efficient option for attacks. One example is elliptic curves over binary fields, where an attack using Weil descents is capable of solving the elliptic curve DLP in sub-exponential time (Petit & Quisquater, 2012). Another family of curves which is vulnerable is the class of supersingular elliptic curves, which can be transformed into an extension field. If the chosen  $k$  is small, the extension field system can then be solved in polynomial time (Menezes, Okamoto, & Vanstone, 1993). Index calculus attacks on discrete logarithms also depend on the type of curve and the field over which the problem is defined. Over small fields, cover attacks using index calculus are best suited, while decomposition attacks work over curves on an extension field. In a variation on the basic index calculus attack, a combination of Weil descent and decomposition index calculus attacks have also been shown to enable a transplanted

elliptic curve in a Jacobian field to be successfully solved, with a 146-bit elliptic curve defined over  $\mathbb{F}_{p^6}$ , an extension field of degree 6, taking approximately one month to break (Joux & Vitse, 2012). The development of these cover and decomposition attacks therefore concretely threaten the security of ECC as a whole.

Implementation of ECC in smart cards could theoretically be weakened by fault attacks, which is a type of side channel attack that actively forces a fault within the algorithm (Jie & King, 2013). These attacks then gather the faulty information from the card to rebuild the secret key for use by the attacker. In particular, there has been the suggestion that ECC could be exploited by sign change attacks, which alter the sign of the point on the curve, however the actual implementation of this attack would be both complex and unlikely to succeed in breaking most applications of ECC. Because of the suggested weakness of ECC to these fault attacks, it is important that the design of the algorithm take side channel attacks into account during the development process, to ensure the system is robust (Ma & Wu, 2014).

### **2.7.2 Issues in Visual Cryptography**

VC schemes encounter difficulties due to pixel expansion, which is the number of subpixels required to encode the correct level of contrast in each share. This expansion greatly affects the required overhead of VC schemes, and as such is the target of much research (Blundo et al., 2006). While there have been schemes proposed that give a constant pixel expansion, such as graph-based extended VC (Lu et al., 2011), many schemes require linear, or even polynomial pixel expansion based on the number of nodes within the scheme, making them infeasible for larger implementations. Within the schemes which ensure pixel expansion remains constant, the overhead for the encoding of the shares is still computationally high for large images with a greater numbers of pixels. These systems which constrain pixel expansion also degrade the contrast of an image, as there are fewer subpixels differentiating dark and light in the image, making it more difficult for the human eye to visually decode. Once multiple colours are introduced to the scheme, pixel expansion becomes even more complex, and overall image contrast is lowered further. A colour VC scheme will also require higher overall time complexity, as each colour within the image must have a different threshold for contrast (Liu et al., 2008).

VC is also open to malicious man-in-the-middle attacks, during the transfer of shares to participants. If the shares are intercepted, the malicious intermediary could keep the original share, and forward a new, false share to the intended participant. The



interception of the share would as such result in the security of the scheme being completely undermined. Attacking a VC scheme in this manner is generally referred to as cheating. While this risk can be decreased by the implementation of an EVCS where each participant is assigned a specific target image, cheating is still possible, by a malicious participant. A malicious participant is an authorized participant in the scheme, who then proceeds to undermine its security through the generation of false shares. Cheating prevention VC schemes have been proposed that use specific basis matrices in the generation of both the secret shares, and a set of verification shares, to counter the ability to generate fake shares (Hu & Tzeng, 2007). These matrices added an extra column to the originals, one column of all 1s in the secret share matrix and one column of all 0s in the verification share matrix. The verification shares can then be stacked to check the veracity of the image. However these basis matrix schemes have since been proven through cryptanalysis not to be cheating immune using two theoretical cheaters working in concert, who are then able to determine the location of these extra columns within the basis matrices (Chen, Horng, & Tsai, 2012). To prevent this type of cheating, it is necessary to introduce multiple extra zero columns into the basis matrices. As a result, cheating prevention VC schemes result in higher overheads and increased pixel expansion when compared regular VC algorithms, which results in a lower level of utility in real-world application. The proposal of adding tags to individual shares to allow for the identification of false or forged shares may offer additional protection against cheating, however it is still vulnerable to attack if an attacker is in possession of a genuine share, and can therefore find and replicate the security tag.

### **2.7.3 Issues in Graph Based Cryptography**

Encryption systems that use graphs for encoding, like those based around VC, can have very high computational overheads, due to the size of the graphs required to achieve the required levels of security. Also, those encryption methods that base themselves around the special colourings of vertices and edges are vulnerable to cubical linearization attacks, which make decryption possible, despite being costly in practice (Ustimenko, 2014). For those graph-based systems that also rely on the DLP, the same vulnerabilities encountered by ECC encryption apply.

Another issue within graph based systems is implementation. Representing a graph within a computer program can be broken down into four possible types: the adjacency list; the adjacency matrix; the incidence list; and the incidence matrix. Each lists either vertices or edges, and they are either enumerated fully - in a matrix - or only

where a connection occurs - in a list (Riaz & Ali, 2011). These implementations affect the use of a particular system, especially with larger connected graphs, with many entries in its matrix or list.

## **2.8 CONCLUSIONS**

Graphic based systems are slowly being incorporated into mainstream use due to their high levels of security. However, the heavy overhead incurred by the computational components of VC systems can limit their usefulness, and they remain vulnerable to attack through the creation of forged shares. Meanwhile the security of ECC algorithms depends on the DLP remaining intractable. The development of specialized cover and decomposition attacks against ECC, and the future advent of quantum computing put the security of the Elliptic Curve DLP at risk. These issues require further examination, as do alternative graphic based systems that incorporate the use of topology for high levels of security. ECC is currently the best developed, researched and applied graphic based cryptographic system and it can therefore be used as a benchmark to compare the performance of any other graphic based system.

Based on the analysis of the security of ECC and VC systems, a comparative analysis of four existing cryptographic methods is proposed in the next chapter, as well as the design, implementation and evaluation of a fifth proposed system. Chapter 3 defines the research methodology which is fundamentally a comparison of competing cryptographic algorithms.

# **Chapter 3**

## **Methodology**

### **3.0 INTRODUCTION**

In Chapter 2, a wide range of literature was reviewed and assessed, and the foundational topics of cryptography, group theory, and graphic methods as applied to encryption were defined. These topics were then analysed, and current research in the areas of encryption and graphic based methods was explored.

In this chapter, the foundation formed from the analysis of literature in Chapter 2 is utilized in the design of the study. Section 3.1 critically evaluates similar studies of relevance to the topic, and explains in depth how the authors of these studies went about the study and the results they gained. Section 3.2 then gives the design of the study, with reference to the standards and benchmarks from the previous research evaluated in section 3.1. In section 3.3 the requirements for the collection and analysis of the resulting data are explained, as well as the data presentation format. Section 3.4 acknowledges the limitations of the research design, and section 3.5 summarises the overall study.

### **3.1 REVIEW OF SIMILAR STUDIES**

In this section, prior similar studies and relevant works are reviewed and evaluated for strengths and weaknesses, as well as the potential application they pose to the design of this study. The key focus is on the way in which the authors went about their research in order to achieve their findings. In this way the best approach to cryptographic testing research may be derived. Nine comparative studies which propose testing criteria for the evaluation of encryption algorithms are introduced and explored in depth in sections 3.1.1 through 3.1.9. Each is evaluated for strengths and weaknesses, and the ways in which the proposed benchmarks in each have been tested is explored. The following section 3.2 then details the design of the research, and the standards for testing developed from the benchmarks proposed in the evaluated studies.

#### **3.1.1 Jeeva, Palanisamy and Kanagaram (2012)**

Jeeva, Palanisamy, and Kanagaram (2012) propose several standards for measuring the security of an algorithm and these are tested on the most widespread symmetric and

asymmetric encryption algorithms. To measure the overall security of an encryption algorithm Jeeva et al. (2012) propose using the key length, strength against attacks such as brute force, known plaintext attacks, et cetera. The ability to alter the encryption parameters at run time, called ‘tunability’ (Jeeva et al., 2012, p.3036) is also suggested as a desirable trait for an algorithm to have, as it increases the overall security.

<b>Factors Analyzed</b>	<i>Symmetric Encryption</i>					<i>Asymmetric Encryption</i>	
	<b>AES</b>	<b>DES</b>	<b>3DES</b>	<b>Blowfish</b>	<b>RC4</b>	<b>RSA</b>	<b>Diffie-Hellman</b>
<b>Encryption Ratio</b>	High	High	Moderate	High	Low	High	High
<b>Speed</b>	Fast	Fast	Fast	Fast	Slow	Fast	Slow
<b>Key Length</b>	128, 192 or 256 bit	56 bit	112 or 168 bits	32 to 448 bits	256 bits	> 1024 bits	Key Exchange Management
<b>Tunability</b>	No	No	No	Yes	No	Yes	Yes
<b>Security Against Attacks</b>	Chosen plaintext, known plaintext.	Brute force	Brute force, chosen plaintext, known plaintext	Dictionary attacks	Bit flipping attacks	Timing attacks	Eavesdropping

Figure 3.1: Results from Jeeva et al., 2012, p. 3036

Jeeva et al. (2012) propose that the efficiency of an encryption algorithm can be measured through computation time. The overall time taken to encrypt and decrypt the information is required to be “fast enough to meet real time requirements” (Jeeva et al., 2012, p.3036). Also proposed is the calculation of the encryption ratio – measured by the length of the data to be encrypted and the key length. Jeeva et al. (2012) suggest this should be constrained as much as possible to improve overall efficiency in the algorithm. The results of their study suggest overall that symmetric encryption gives a faster performance where the encryption ratio is higher. Systems such as RC4, a symmetric stream cipher with a low encryption ratio, are classed as slow, while AES, which receives a high encryption ratio, is rated as fast.

The benefit of studying both the efficiency and the security of encryption algorithms is that the trade-off between the strength of the algorithm and the overall computational complexity. It can be carefully evaluated to deliver a better overall

understanding of the usefulness of the algorithm in real world situations. Encryption systems such as AES, which is shown in Figure 3.1 to have high levels of both efficiency and security, will be more portable to multiple architectures and situations than those such as RC4, which Figure 3.1 shows as rated slow in overall encryption time. Strong but slow encryption algorithms are unlikely to be utilized in any situation that requires real-time processing or fast communication between participants.

### **3.1.2 Afzal, Kausar and Masood (2006)**

Afzal, Kausar, and Masood (2006) proposed and implemented a framework for the evaluation of 34 different stream ciphers. These ciphers were submitted to ECRYPT, the European Network of Excellence for Cryptography, for the eSTREAM project, which looked to create a standard stream cipher algorithm (Afzal et al., 2006). Stream ciphers are highly useful in processing real time data, due to the way they operate on individual pieces of plaintext sequentially, usually at high speed. The study done by Afzal et al. (2006) was prompted by the need for a new standard of stream cipher, and focused on the evaluation of the submissions for such a standard by the international research community. The study looked at the overall design of each of the ciphers. Each main subset of stream cipher is broken down, evaluated for strengths, weaknesses and practical applications. For the purposes of evaluation, the proposed ciphers are split into two categories; those with high-level software implementations and security of at least 128-bits, and those with low-level hardware implementations and security of 80-bits. The hardware and software implementations were also designated as either bit-oriented, operating on a single bit of data at a time, or word oriented, operating on a bit-word.

According to Afzal et al. (2006), there are several main elements commonly used in stream cipher design. Linear feedback shift registers (LFSR) are common due to their structure, which allows for low-cost implementation. Because LFSR operate in a linear fashion, ciphers that use LFSR in their design are required to also include a form of non-linear function. Non-linear feedback shift registers (NLFSR) do not suffer from the linear properties of LFSR, but most NLFSR operate in small cycles, and are weakened as a result. Feedback with carry shift registers (FCSR) are LFSR with one point of difference. Instead of modulo arithmetic using modulus 2, the addition is performed through carrying propagation. To ensure non-linearity, functions such as clock-controlled generators can be implemented in the cipher. If the cipher is irregularly clocked, this can break the linear properties of the algorithm. Non-linearity can also be introduced using combining or filtering functions.

Within the categories set out by Afzal et al. (2006), LFSR was the most commonly implemented function, however only a few ciphers submitted used a non-linear function in conjunction with LFSR. The only cipher that implemented a non-linear function was subsequently discovered to be weak to an algebraic attack. The second most popular design element found in the 34 submitted ciphers was the non-linear combining or filtering functions.

Because Afzal et al. (2006) evaluate the different ciphers based on their design elements, they are able to compare within categories, as well as apply known techniques for cryptanalysis on the ciphers within those categories. This gives a framework for the testing of each cipher, as not all cryptanalytic methods can be applied to all stream ciphers. The results of this comparative study by Afzal et al. (2006) explored only the design features of each stream cipher, and gave little practical data in regards to each cipher's efficiency, or their individual security. Most of the analysis of the algorithms was instead based on the theoretical basis of the design elements.

### **3.1.3 Sharma, Garg and Dwivedi (2014)**

Multiple studies have examined the comparative efficiency and performance of encryption algorithms, without also exploring the security of each algorithm. In the study by Sharma, Garg, and Dwivedi (2014), the authors propose a new symmetric encryption method called NPN, or  $n$ th prime number. The NPN encryption system is then compared with the DES algorithm. The comparison in Sharma et al. (2014) is based on the time taken for encryption and decryption in each algorithm, as well as the overall memory requirements for each. This provides a clear and concise view of the comparative efficiency between the proposed algorithm and the benchmark DES algorithm, however the results of this study are limited by the lack of comparison between the security of the two competing algorithms.

The proposed NPN encryption system in Sharma et al. (2014) utilizes multithreading for efficiency optimization, and is a symmetric cipher. The use of parallel programming within the implementation is offered as a way to optimize its time requirements in encryption and decryption. NPN operates on the class of Strings in Java, and involves finding the  $n$ th prime number for each character of the plaintext, and then adding a particular constant to that recovered prime prior to adding the result to the ciphertext string.

The choice of DES as the single comparative algorithm is somewhat limiting, as DES was superseded in 2001 by AES after being proven to be computationally insecure.

As such, DES is no longer considered a benchmark for security and efficiency in cryptography applications. The comparative results of the study look at a standard DES algorithm and a multithreaded NPN algorithm, and their relative efficiency in nanoseconds. For a larger data size of 156 characters, the standard DES algorithm takes approximately 1.93% (3d.p.) of the time taken by the multithreaded NPN algorithm. The NPN algorithm also uses only a 32-bit key, which suggests the algorithm would be computationally insecure.

Sharma et al. (2014) offer data on the overall efficiency of each algorithm as relates to encryption and decryption time, but do not give information about the comparative security of the algorithms. This lack of analysis in regards to security means the overall analysis is limited in its ability to draw conclusions about the use of each algorithm. The choice of DES as a benchmark also limits the author's ability to demonstrate the security and efficiency of their algorithm as compared with current technological standards.

#### **3.1.4 Kohafi, Turki and Khalid (2003)**

In Kofahi, Turki, and Khalid (2003), the efficiency of the DES, 3DES and Blowfish algorithms are compared, based on memory requirements and processing time. Similar to the previous studies discussed, Kofahi et al. (2003) provide a clear comparison of the efficiency of each algorithm, but the comparative security is not discussed.

Kofahi et al. (2003) utilize the inbuilt Java Cryptography Architecture (JCA) for the implementations of their compared algorithms. All three compared algorithms are symmetric block ciphers, and are designed around Feistel ciphers. Because of the similarities in design, they are highly comparable. Kofahi et al. (2003) timed each of the modular operations of key generation, encryption and decryption individually for each of the three algorithms, without user interaction, so as to give a more complete picture of the efficiency of each. For all algorithms, the time required for key generation was approximately equal. Blowfish was demonstrably more efficient in encryption and decryption than DES and 3DES, taking approximately 13.5 seconds for each operation, where DES took 25.3 for encryption and 26.5 for decryption, and 3DES took 39.2 for encryption and 38.5 for decryption.

The comparison of relative efficiency of the algorithms in Kofahi et al. (2003) does not look at CPU load, or memory requirements, and also fails to address the relative security of each algorithm. As such, it is limited in comparative ability to the time requirements for each stage of the algorithm. However, the testing schema is designed to

compare the three algorithms and gives a thorough overview of the time requirements, which can be used as a basis for further testing.

### **3.1.5 Masadeh, Aljawarneh, Turab and Abuerrub (2010)**

Masadeh, Aljawarneh, Turab, and Abuerrub (2010) provide a framework for evaluating proposed algorithms against industry standards. The authors propose a new encryption method specifically aimed towards the encryption of wireless network traffic, and compare it in practical implementations against AES, DES, 3DES and Blowfish. However, this comparison is limited as the proposed method is asymmetric, while all the comparative algorithms are symmetric.

The proposed sWiFi system (secure Wireless Fidelity) uses a Feistel structure and asymmetric encryption. It utilizes Automata Theory, and contains an alphabet, or codebook, of all words  $W$ . The designed implementation operates on ASCII characters, and has two main functions for key generation  $S(L)$  and  $P(L)$  which are used to create the keys for encryption and decryption in the algorithm. The size of the key used in the system proposed by Masadeh et al. (2010) is not discussed, though the algorithm is disclosed as operating on 64 bit blocks of data.

The study bases evaluation on the time requirements for each of the algorithms. All algorithms were evaluated on three operating systems, Windows Vista, Windows XP and Linux. The algorithms were tested in each environment on three sizes of file: 145MB, 510MB and 900MB. Masadeh et al. (2010) then give the time taken in seconds for encryption in each algorithm. The proposed sWiFi scheme gives a better time performance overall than the standard algorithms it is tested against, for each data size. The time taken for decryption and key generation is not addressed by the study.

The results of the study are based on quantitative empirical data, and all algorithms are tested on three different computer operating systems and with three different sample sizes of plaintext for a fuller range of results. The lack of results relating to relative efficiency in decryption and key generation limits the ability of the authors to draw conclusions about the performance of the algorithms. Masadeh et al. (2010) are also hampered by the lack of evaluation of the security of the compared algorithms, as the proposed sWiFi system does not involve the disclosure of key size or details of its implementation. Comparison to the efficiency and security of other asymmetric ciphers would also give a better view of the relative performance of the proposed sWiFi system.

### **3.1.6 Thakur and Kumar (2011)**



Thakur and Kumar (2011) compared the relative performance of AES, DES and Blowfish, using the execution time over different sizes of data as the measure of efficiency. AES was evaluated in each of four modes: ECB, CBC, OFB and CFB. The simulation results gave Blowfish as the most efficient algorithm in respect to performance.

Thakur and Kumar (2011) used the Java Cryptography Architecture (JCA) in the Java Development Kit 1.7 to implement the three algorithms, through use of Java's Cipher class. DES was evaluated with a key size of 64 bits, while AES and Blowfish used a key size of 128 bits. The authors did not evaluate relative security, as "strength against cryptographic attacks is already known and discussed" (Thakur and Kumar, 2011, p. 10). Instead, performance was evaluated based on the time required to encrypt and decrypt plaintext of multiple sizes. The execution time was measured in seconds, and the tested plaintext sizes were from 3 KB to 203 KB blocks. Each experiment was performed twice, on a system with an AMD Sempron processor running 2GB of RAM. Thakur and Kumar (2011) reason that the repetition of the experiment allows them to establish of the validity of the experimental results.

Overall, Blowfish gave a better performance in encrypting and decrypting the data than DES or AES. AES was tested against DES and Blowfish in each available mode: ECB, CBC, OFB and CFB. This enabled a further comparison of the efficiency of each mode of AES encryption, with OFB mode resulting in the best performance. However, AES resulted in the highest processing time of all the algorithms, regardless of the chosen mode of encryption. The repetition of the tests lent validity, though many more repetitions would likely provide a more conclusive picture.

### **3.1.7 Bhat, Ali and Gupta (2015)**

Bhat, Ali, and Gupta (2015) studied the performance of the AES and DES encryption algorithms. This research looked at the memory requirements of each algorithm and time taken, as benchmarks for efficiency. It also examined the overall avalanche effect of the algorithms as the benchmark for security. The benefit of looking at the avalanche effect in an encryption algorithm is the ability to gauge the scheme's resistance to chosen plaintext attacks.

AES resulted in almost double the change in bits that was provided by DES in the ciphertext given a one bit variation in the plaintext. This increase in the avalanche effect provides a much higher level of security against cryptanalysis. However, the overall memory requirements for AES proved to be four times that of the DES implementation.

The simulation time of AES was also far higher than that of DES, with AES taking approximately ten times the simulation time of the DES implementation. As AES operates using a 128-bit key where DES uses a 64-bit (56-bits for computation) key, this discrepancy in efficiency is expected. The trade-off required for higher levels of security is decreased efficiency. Bhat et al. (2015) implemented the experimental design in Matlab 7, on an Intel Pentium machine with 2 GB of RAM.

The authors do not discuss the repetition of the performed tests, or the specific details of the implementations, which creates difficulties for readers in regards to reproducing the study or critically analysing the results.

The comparison made by Bhat et al. (2015) of avalanche effect and efficiency gives a clear view of the some of the trade-offs made in obtaining the higher level of security in AES. The dramatic increase in the avalanche effect is tempered by the higher performance costs necessary to achieve the required level of security. The lack of information regarding the design of the study gives the reader little to no information about the robustness of the presented results.

### **3.1.8 Prachi, Dewan and Pratibha (2015)**

Prachi, Dewan, and Pratibha (2015) compared the security and efficiency of ECC, RSA, DES and AES. This comparative analysis was based on theoretical knowledge, as opposed to practical simulations. Efficiency was measured by the key size of each scheme and the estimated output size of the data, to evaluate likely memory requirements. The security analysis was based on the key size of the scheme in relation to the number of operations required for a successful brute force attack.

The overall consensus of Prachi et al. (2015) was that ECC had a better level of security and a higher level of efficiency than the other algorithms, based on these theoretical underpinnings. The comparison of key sizes between RSA and ECC displayed a higher level of computational security for relative key size in ECC, with a 160-bit ECC key equivalent in security to a 1024-bit RSA key. ECC is also shown to be theoretically more efficient than RSA, as a smaller key size results in lower computational overheads. However, the comparison with DES is limited to the key size of the algorithm, and while the operation of an AES implementation is explained by Prachi et al. (2015), little practical or theoretical comparison is made by the authors between AES and ECC. The results of the analysis gave the security of algorithms as “Good”, “Excellent” or “Not Good Enough” ratings, but did not give the metrics used to arrive at these conclusions, which removes the ability to reproduce the study.

### **3.1.9 Singhal and Raina (2011)**

Singhal and Raina (2011) compared the efficiency and performance of AES and RC4. They used empirical data, and their metrics based the evaluation on time taken for encryption and decryption, the throughput (processing speed in kB/s), the CPU load for each process and the overall memory used. The wide span of the metrics gives a thorough representation of the efficiency of the two algorithms, and the differences in performance between RC4 as a stream cipher and AES as a block cipher.

Each algorithm was tested with multiple sizes of plaintext data, with file sizes between 100 KB and 50 MB. AES was also tested in three different operating modes, ECB, CBC and CFB, to see the effect the mode has on its overall performance. ECB mode gave the best performance of the AES encryption modes, however it was still significantly slower than RC4. Encryption time was also tested using multiple different key sizes of 128, 196 and 256. The time requirements for encryption in RC4 remained stable despite the change in key size, while the time taken for encryption with AES increased as the key size increased. One explanation for the increase in time is the increase in rounds. 192 bit AES involves 12 rounds, compared to 128 bit AES with 10 rounds, and 256 bit AES performs 14 rounds. This translates into many more operations per data block to encrypt the plaintext, which then accounts for the time difference. Similarly, as the key size used in AES increases, the throughput of the algorithm decreases. The results for efficiency in decryption is mirrored by those of encryption.

In testing the memory, Singhal and Raina (2011) found that RC4 required lower levels of memory to perform encryption on larger files. As the file size increased, AES required dramatically more memory than the RC4 implementation. The CPU load requirements for RC4 over the different file sizes were also lower than that of all modes of AES. Based on these metrics, RC4 takes less time to encrypt and decrypt information overall than AES, regardless of the mode of use, and the key size of the encryption system is shown to have less effect on the overall encryption time of RC4 than it does in AES. Using the standards for evaluation given, the authors are able to provide empirical evidence for the assertion that the performance of RC4 is more efficient than that of AES.

## **3.2 RESEARCH DESIGN**

This section gives the design of the study, which starts with guidance derived from the research reviewed in Section 3.1. Section 3.2.1 summarizes the studies discussed in

Section 3.1, as well as the issues and problems faced in these studies. The research questions are proposed in 3.2.2. The phases of the research and the architecture of the testing systems are described in 3.2.3. Then in 3.2.4, the design of the proposed graphic system, referred to as a coordinate matrix encryption scheme, is explained.

### **3.2.1 Summary of Similar Studies and Review of the Problems and Issues**

The standards for security and efficiency proposed in Jeeva et al. (2012) are the most comprehensive of the examined studies, and as such can be used to design the benchmarks for the comparison of the graphic based encryption algorithms. The study provides clear testing criteria, which measure the efficiency and security for each algorithm. However, more detailed analysis of the security of each algorithm is required, and thus extra testing criteria from other studies will also be required such as the measuring of the avalanche effect of each algorithm, proposed in Bhat et al. (2015). The avalanche effect of an algorithm is an easily implemented and highly effective way of quantifying the security of a system. It is necessary when designing the study to ensure that all standards for both efficiency and security are as comprehensive and exhaustive as possible, so that the resulting data can be compared with confidence that it is representative of the relative performance of each algorithm.

Many of the evaluated studies focused in particular on either security or efficiency in their testing. This bias towards on particular aspect of the algorithms results in an uneven comparison of the trade-offs required. It is necessary to study how a particular algorithm has achieved a high level of efficiency, and what trade-offs may have occurred in security to obtain it, and vice versa. As such, this study will attempt to balance the testing of each aspect, so that the reader might get a more complete picture of the benefits, drawbacks and possible applications of each algorithm tested.

Another point of importance in the comparative analysis of encryption algorithms is the necessity of comparing algorithms of the same classification. Studies such as Masadeh et al. (2010) compare a new asymmetric method with standard symmetric methods, rather than with established asymmetric methods. While some comparison between types is useful, more relevant data results from the comparison of algorithms of the same type. As such, this study will use comparable algorithms of both symmetric and asymmetric type, as well as graphic and classical algorithms and stream and block ciphers, to give a full range of results by comparing industry standards with the proposed symmetric graphic-based, word-oriented, stream cipher system.

One aspect on which multiple studies on the comparison of encryption algorithms

often suffer is the lack of empirical data on which to base assertions. Surveys such as Kofahi et al. (2003) and Chandra et al. (2014) rely on the theoretical basis of the algorithms in question, which limits the results of such a comparison. The research in this study will rely on both theoretical knowledge and actual test results, so as to give a more thorough and representative analysis. The importance of the theoretical comparative performance of the algorithms is uncontested, however providing empirical data from simulations of the algorithms in a study gives a better view of the theoretical results, as well as providing a stronger basis for any and all conclusions from the data. As such, this study will look at both the theoretical basis of and real-world implementations of the comparative algorithms and the results of both, so as to find the algorithm that best balances security and efficiency.

### **3.2.2 Research Questions and Hypotheses**

From the literature reviewed in Chapter 2, a detailed understanding and evaluation of the ideas, foundations and benchmarks for cryptographic algorithms has been presented. The integrity and security of data are the main motivations behind the use of cryptographic algorithms and the implementation of graphic based systems revolves around their unusual structures and high levels of security. The ease and efficiency of the deployment of an encryption algorithm is also an important factor in choosing a cryptographic algorithm. Using these criteria of security and efficiency, it is possible to evaluate multiple cryptographic algorithms for their overall usefulness and application to real world situations. Based on these ideas and research criteria, two main research questions have been formulated for the study of graphic based encryption algorithms in this thesis. The first research question can be derived from the evaluation of literature surrounding ECC, evaluated in 2.5, which suggests the particular structure allows for greater security and efficiency than the traditional RSA. The improvement in security is shown in Pateriya & Vasudevan (2011), who found that a 256 bit ECC key is equivalent to a 3072 bit key in RSA. The second research question comes from the research completed into problems surrounding the implementation of ECC and VC schemes, evaluated in 2.7.1 and 2.7.2. VC schemes suffer particularly due to pixel expansion. Blundo et al. (2006) found that in most VC schemes pixel expansion increased relative to the number of nodes. Liu et al. (2010) noted that all schemes attempt to minimize this expansion to improve the efficiency of the implementation. ECC schemes face difficulties in implementation due to their complexity, and the increased length of the ciphertext (Chandra et al., 2014).

*Research Question 1:*

**What are the security benefits of graphic based systems in comparison to classical block ciphers?**

*Research Question 2:*

**What difficulties are faced in the implementation of graphic based systems?**

*Sub-questions:*

*Sub-question 1:*

Does the implementation of the proposed method provide better levels of security than the comparable algorithms?

*Sub-question 2:*

How is the level of security achieved in the proposed method?

*Sub-question 3:*

What is the reduction in computational overhead in the proposed scheme from comparable algorithms?

From these research questions, the studies evaluated in 3.1 and the literature reviewed in Chapter 2, two specific hypotheses have been developed for further exploration. Hypothesis 1 is derived from the research conducted into comparing traditional RSA and the graphic-based ECC. ECC provides a faster execution with lower overheads, as demonstrated in the study conducted by Prachi et al. (2015), reviewed in subsection 3.1.8. Hypothesis 2 is derived from the studies of the security of VC methods, and the VC definition of perfect secrecy, as originally proposed by Naor & Shamir (1995), wherein an adversary with unlimited computing power is required to guess, for any given pixel, whether that pixel is black or white.

*Hypothesis 1:*

**Graphic based methods provide a better level of security with lower overheads than classical encryption techniques.**

*Hypothesis 2:*

**The proposed encryption system based around graphic methods is computationally secure against cryptanalytic and brute force attacks.**

### 3.2.3 Research Phases & Algorithm Implementations

The testing and analysis performed in the study will require multiple phases of development. Each phase will enable fine-tuning of the algorithms and testing methods, so as to ensure that all collected results are as accurate as possible. Phase 1 will involve the development of the implementations for the comparative algorithms. Phase 2 will see the implementation of the proposed coordinate matrix encryption scheme. Phase 3 will then consist of the testing of the algorithms, based on the benchmarks described in Section 3.3 for efficiency and security. In Phase 4, further refinements will then be made to the proposed coordinate encryption scheme, which will be retested as part of Phase 5. The final Phase 6 will involve the analysis of the data from phase 5 based on the predetermined benchmarks for efficiency and security.

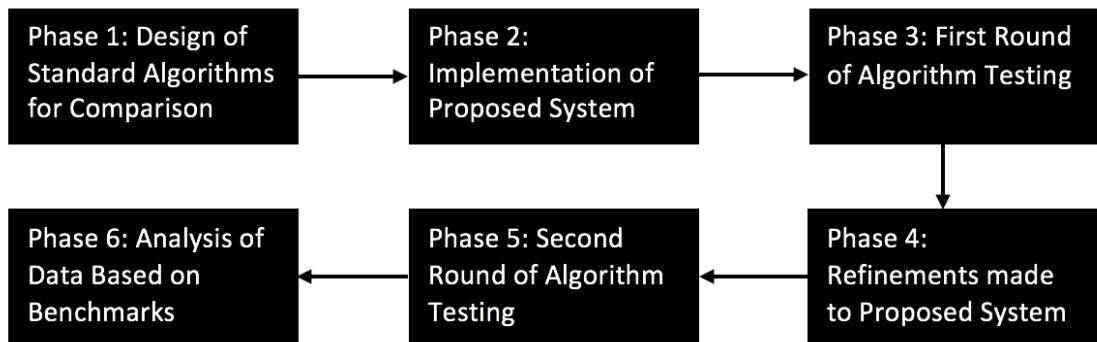


Figure 3.2: Phases of research

All proposed algorithms will be implemented in Java, and two different versions of the proposed algorithm will be developed to allow for a more effective comparison. The testing will occur on the following machine configuration: a laptop with a 3.1 GHz Intel Core i7 processor, and 16 GB RAM.

The standardized algorithms that have been chosen for the purposes of comparison are: Elliptic Curve Cryptography (ECC); 2-out-of-2 Visual Cryptography (VC); the Advanced Encryption Standard (AES); and Rivest Cipher 4 (RC4). ECC is the most widely developed of graphic encryption methods, and should therefore provide a target in performance for the security and efficiency of such systems. A 2-out-of-2 VC scheme operating on binary digits instead of black and white pixels will provide a highly comparable implementation to the proposed coordinate matrix encryption system. Each of the generated shares for the VC scheme will consist of a binary string, rather than a black and white share image. AES is included to allow for the comparison of the differences in efficiency and security between graphic and classical methods of encryption. Finally, RC4, while now considered insecure, is still one of the best-researched stream ciphers, and is not under patent. It is also freely available for

implementation through Java’s cryptographic framework, and as such can be used for comparison between the proposed system detailed in 3.2.4 and classical stream ciphers. The implementations of AES, RC4 and ECC will utilize Java’s inbuilt crypto package and its functions.

### 3.2.4 Coordinate Matrix Encryption Algorithm Design

The proposed algorithm design for the Coordinate Matrix Encryption (CME) scheme will be based around a square coordinate matrix and transformations in a finite Galois field  $GF(2^n)$ . The coordinate matrix design will be based around the concepts used in error-correcting codes, in which sparse matrices and code words are used to eliminate noise from the transmissions, and will utilize security principles from VC. A brief overview of the algorithm is given in this section. Full implementation details can be found in Chapter 4, and source code for the algorithm can be found in Appendix B.

```
Total strings: 8
Number of occupied spaces: 32
Number of blank spaces: 32
Total matrix size: [8,8]
[---][---][101][---][---][---][---][---]
[---][010][011][---][---][---][000][---]
[000][111][---][001][---][101][---][---]
[110][000][001][---][---][---][---][111]
[---][---][---][010][111][100][100][---]
[010][100][001][---][---][100][101][011]
[---][---][101][010][---][110][---][---]
[110][---][011][011][111][000][001][110]
Set up complete, time taken: 19 ms.
Total memory used: 0.43109130859375 MB
```

Figure 3.3: A randomly generated key matrix for a 3-bit coordinate matrix scheme.

An  $n$ -bit coordinate scheme which uses all  $2^n$  possible  $n$ -bit strings will consist of a  $2^n$ -by- $2^n$  encryption matrix, containing  $2^{2n}$  total coordinates. Within the matrix, each possible  $n$ -bit string of binary values will be assigned to multiple random coordinate locations, as per Equation 3.2.4.i.

$$\text{Equation 3.2.4.i} \quad M_{size} = (2^n)^2$$

$$N_{locations \text{ per string}} = \frac{M_{size}}{2(2^n)}$$

The remaining coordinate locations will be assigned as empty, for use in padding the ciphertext output. The process of generating this key matrix is shown in Figure 3.4. Once this coordinate matrix has been created it becomes the encryption key for all users, because the coordinate scheme is symmetric. The total number of possible key matrices



for an  $n$ -bit scheme is described in Eq. 3.2.4.ii. Fig 3.3 shows a 3-bit CME key matrix.

Equation 3.2.4.ii

$$bit\ string\ length = 2^n$$

$$Matrices = (bit\ string\ length + 1)^{bit\ string\ length^2}$$

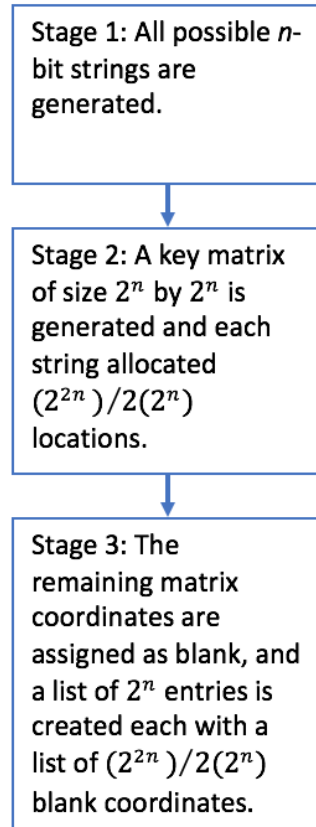


Figure 3.4: Key matrix generation in the Coordinate Matrix Encryption scheme.

As well as the creation of a key matrix, the key matrix is then used to produce a pseudo-random key string. This key string is the first  $x$  coordinate for each of the generated bit strings. The key string is combined with the plaintext as the first part of the encryption process to eliminate any statistical properties prior to the main encryption, via an exclusive-OR operation. The key string is combined with the plaintext again as the last stage of decryption to recover the original plaintext. Because the key string is produced from the key matrix, there is no requirement to communicate an extra key, as all parties with a copy of the key matrix can calculate the key string.

The main encryption process uses a randomized coin toss style procedure, which is similar to the VC method of choosing whether a given pixel is black or white. This coin toss decides if the next section of the ciphertext is to be a blank padding section, or if it is the next section of the plaintext message. If it is a blank padding section, one of the

locations containing an empty entry is picked at random from a blank list, and the binary or integer coordinates (depending on the implementation) of that location are then input as the next part of the ciphertext. Else, if the section is a part of the plaintext message, then a location containing that bit string is randomly chosen from the list of locations for the string. The location is then translated into the corresponding coordinates and concatenated to the ciphertext. The scheme involves the addition of exactly the same number of blank coordinates as enciphered message coordinates. As a result of the addition of padding characters, the resulting ciphertext is exactly four times the length of the plaintext, with two coordinates for every message or padding character, and exactly the same number of padding and message characters. The style of encryption means that the total length of the outputted ciphertext is fixed at exactly four times the length of the plaintext, which may prove to result in undesirable overheads for transmission.

The padding of the ciphertext adds noise and confusion to the output, and the exclusive-OR operation assists in stripping the statistical properties from the plaintext. The addition of multiple locations for each bit string also further diffuses any statistical properties that the combined key string and plaintext may have, and protects against attacks involving known or chosen plaintext. The variable nature of the padding and different coordinate locations for each string results in multiple different ciphertexts for each encryption of a single plaintext, and as such help guard against known plaintext attacks. Because the encryption process operates on individual pieces of the plaintext sequentially, the proposed CME scheme can be classified as a stream cipher in the byte implementation, operating byte-by-byte, fulfilling the definition in Martin (2012) of a stream cipher as a block cipher with a block size of less than 64 bits. The proposed system could theoretically be implemented with a larger block size, for use as a standard symmetric block cipher, however the memory requirements to generate a key matrix in a 64-bit scheme would be significant, and are too costly for the purposes of this research. As such, the proposed CME system will be implemented as a stream cipher, with a block size of 8-bits for the implementation compared with AES, RC4 and ECC, and a block size of 4-bits for the implementation compared with VC.

The use of multiple locations for each bit string and the addition of an equal number of padding coordinates at random locations in the ciphertext should provide resistance to cryptanalysis, and particularly to known and chosen plaintext attacks, as the encryption process therefore results in a non-singular mapping. This non-singular mapping means each plaintext input has many possible ciphertext outputs for any one key matrix. The multiple locations also result in far more of the overall matrix being taken up

by bit strings than would be the case if each string appeared only once. Again, this helps prevent cryptanalytic attacks, as it increases the likely occurrence of the same of padding coordinates appearing more than once, which is helpful in further confusing any analysis of the resulting data. A sample of a 16-bit plaintext and the corresponding 64-bit ciphertext resulting from encryption using a 4-bit coordinate matrix scheme is shown in Figure 3.5.

**Plaintext:**  
**0101000110111011**  
**Ciphertext:**  
**1110011001001011101101001100010001100111101100111010010000000010**

*Figure 3.5: Example plaintext ciphertext pair output from a 4-bit CME scheme.*

The decryption process uses the same key matrix as in the encryption process and looks up each of the coordinates. If a given coordinate is an empty padding variable, it is discarded. If not, the value of the coordinate is combined with the next character of the key string using exclusive-OR, and the resulting value is added to the plaintext output. In this manner, the extra noise generated by the encryption process to ensure security is efficiently removed during decryption. Because each step of the decryption process consists only of simple entry check and exclusive-OR operation, the overall efficiency for decrypting the ciphertext is theoretically higher than that of the encryption process.

The CME algorithm, implements VC methods of security but is classified as a symmetric encryption algorithm, not a secret sharing scheme. Depending on the bit-size of the scheme, it can be classed as either a stream or block cipher. This is due to the way it operates on a set number of bits at each point before moving to the next set, and the blocks of bits are not linked. The 8-bit byte version is a stream cipher, as it moves along the plaintext encrypting it one byte at a time. Figure 2.2 (Martin, 2012, p.107) showed the operation of block versus stream ciphers. In relation to classical encryption methods, the CME cipher also makes use of some of the ideas of error-correcting codes seen in subsection 2.2. The influence of error-correcting codes can be seen in the use of binary codewords of a particular bit size, which are used to encrypt the data. Based on the divisions established by Afzal et al. (2006), the 8-bit CME scheme is a word-oriented stream cipher, as it operates byte-by-byte, rather than bit-by-bit.

### **3.3 DATA REQUIREMENTS**

This section outlines the requirements for the analysis and presentation of the raw data resulting from the research outlined in the previous section. Subsection 3.3.1 summarizes

the type of raw data that will be used for analysis of the results, measuring for efficiency and security. Subsection 3.3.2 describes the methodology of the analysis, and the standards each algorithm will be compared on. Subsection 3.3.3 outlines the ways in which the results of this analysis will be presented, and how the overall comparisons between the algorithms will be displayed.

### **3.3.1 Algorithm Testing**

Each of the algorithms for comparison will be implemented in Java. The VC and binary string version of CME will take as input a pseudorandom binary plaintext string, and return the encrypted binary string before decrypting and returning the corresponding plaintext. The AES, RC4 and byte-oriented CME schemes will take as input a UTF-8 encoded plaintext string, transform it into the corresponding byte array, and encrypt it. After the decryption the array will be translated back into its corresponding UTF-8 encoding. Each of the operations within the implementations will be timed, without any user interaction, to ensure that the times are accurately reflected. The raw data output of each algorithm will be collected. Each algorithm will undergo multiple tests on different plaintexts of different lengths. Testing will be repeated over many iterations and the mean calculated. For the purposes of analysis, transformations on the binary string will be done in a high-level Java software implementation of the algorithms for VC and CME, and on byte arrays in a low-level Java software implementation for the ECC, AES, RC4 and CME comparisons, to ensure the results are an accurate representation of the comparative results of each of the algorithms.

The VC and bit-string CME algorithms will be given as input different pseudorandom binary plaintext strings of data, in size 16, 32, 64, 128, 256 and 512 bits. This will allow the testing of the implementations over a variety of sizes, to observe how well the schemes scale up. Testing will be done for efficiency based on time requirements and memory occupied by the JVM at each stage, while testing for security will be based on the avalanche effect, the theoretical resistance of the scheme to brute force and chosen or known plaintext attacks.

For the testing performed on byte array implementations of AES, RC4 and CME, the input will be a plaintext string encoded in UTF-8. This will be transformed into its corresponding byte array, before encryption and decryption are performed. The resulting array of bytes will then be transformed back into its corresponding UTF-8 encoded plaintext string. The schemes will be tested on multiple sizes of plaintext data, with lengths of 304, 928, 3024, 4408 and 8166 bits. These English language plaintext strings

will contain generic frequency information as is typical of the language, and have been taken from excerpts of *Pride and Prejudice* (Austen, 2006) and *Hamlet* (Shakespeare & Ackroyd, 2006). The use of this particular type of plaintext data will allow for frequency analysis to be performed on the resulting ciphertexts. A custom-built program will be implemented to analyse the ciphertext frequencies. Figure 3.6 gives an example output of the frequency analysis program. The AES, RC4 and byte CME schemes will also be tested for efficiency using the time and memory requirements at each stage, and for security using theoretical resistance to brute force attacks, chosen or known plaintext attacks, and the practical avalanche effect.

```
Frequency analysis of encrypted ciphertext, based on bit scheme length:  
Value : 1111, Occurs : 1, Bit Value :  
Value : 0000, Occurs : 1, Bit Value : 11  
Value : 1100, Occurs : 1, Bit Value : 00  
Value : 1011, Occurs : 1, Bit Value :  
Value : 0111, Occurs : 1, Bit Value : 10
```

*Figure 3.6: An example of frequency analysis on a 2-bit coordinate matrix scheme.*

As ECC is a scheme used mainly in the implementation of the ECC Diffie-Hellman key exchange protocol, rather than for the actual encryption of data, the comparisons between the byte implementation of CME and ECC will occur based on the practical efficiency and theoretical security of key generation. These comparisons will be based on the results of timing the operations, the memory required to perform the operations, and the underlying theoretical basis of the algorithms.

The memory requirements of each of the algorithms will be based on the in-use memory in the Java Virtual Environment at the end of each stage of the implementation.

### **3.3.2 Algorithm Analysis**

This section describes the standards used for the analysis of the results from the algorithm testing phase. The tested algorithms will be evaluated along benchmarks for security and efficiency. Security will be measured via theoretical analysis of the key space and resistance to attacks, as well as practical cryptanalysis and examination of factors such as the avalanche effect. The efficiency of the compared algorithms will be measured through the time in milliseconds taken for each stage of the process, and the required memory usage for each of these stages.

To measure the efficiency of the algorithms, the setup of the key, the encryption, and the decryption of the binary string will all be timed as part of the implementation. This will enable comparison between each of the implemented methods, as well as the average time of each algorithm over many tests at each level of encryption. The resulting

times will be compared in milliseconds. The amount of memory required by the Java Virtual Machine (JVM) runtime environment during the execution of the implementation, for setting up the scheme, for encrypting the plaintext, and for decrypting the ciphertext will all be measured to evaluate the hardware and space requirements that are necessary for each algorithm.

The security of the algorithms will be evaluated by several factors. Mathematical evaluation of the key space, the number of operations required to try all possible keys, frequency analysis of the ciphertext (where applicable), and theoretical weaknesses of each algorithm to known and chosen plaintext attacks will also be explored. The avalanche effect of the ciphertext resulting from a change to a single bit or byte of the plaintext undergoing encryption will also be measured for each algorithm. The program implemented to measure the overall avalanche effect in the byte implementations of AES and CME will look at both the number of changes to the overall bytes of the ciphertext – how many of the same bytes occur in the two different ciphertexts – as well as the changes in order – how many of the same bytes occur in the same places in the two ciphertexts. The avalanche effect in the bit string implementations of CME and VC will be measured by the percentage of positions in which the bits are unchanged. These tests will provide a view of the overall avalanche effect in each algorithm.

### **3.3.3 Data Presentation**

The data resulting from the testing and analysis phases of the research will be presented in Chapter 4: Research Findings. The analysis of the efficiency of each algorithm will be discussed individually, as well as the comparative overall efficiency for each algorithm. These results will be presented visually as well as textually, collated into tables for more effective understanding. The analysis of the security of each algorithm will be presented textually, as will the comparative analysis of the algorithms. The presentation of data on key space relative to scheme size will also be presented visually in comparative tables, in which data and results will be grouped first by the particular test, then by the algorithm used, and finally by the message or data block size that was presented for encryption. The grouping of data will provide an explanation of results as well as any and all assertions made based on those results. The results will then be critically examined further and the implications of the study will be explored in Chapter 5: Research Discussion.

The source code for each algorithm's implementation and for the custom-built testing programs will be provided for review in Appendix B, while the plaintext data used for testing each implementation will be provided in Appendix C. A selection of raw

testing data and results will be provided in Appendix D. The addition of the testing data will allow the reader to better understand the overall results, as well as gain further insight into the way in which the data was collected for the study. The addition of the source code also provides the opportunity for the reproduction of the study by the reader.

### **3.4 LIMITATIONS**

It is necessary to identify the limitations of the research, so as to correctly define the scope of the results, and the possible applications of the data presented. These limitations should also be kept in mind when presenting and analysing the results of the research.

The research plan proposed in Section 3.3 is affected by the design of the algorithm implementations for each compared scheme. Due to the use of high-level software implementations for the most effective comparison of VC and string-oriented CME, the efficiency of the algorithms may be less than it would in an equivalent low-level software or hardware implementation. This trade-off is necessary to ensure that each of the algorithms is able to be fairly compared with its counterparts, but it is important to note nonetheless that this particular implementation will most likely result in slightly longer processing times for these VC and CME schemes than might occur in real world applications that implement encryption schemes at a low-level in software or hardware instead. The results from the VC and string -oriented CME will therefore not be as easily generalized to those real-world applications as the results from the other algorithms may be.

The programming for each implemented scheme will also impact on the results. The implementations of AES and RC4 will utilize the inbuilt functions of the `javax.crypto` package, as will the implementation of ECDH. These inbuilt options for encryption simplify the overall implementation. A version of the CME scheme which operates on byte arrays will be created for a more balanced comparison with these algorithms. In comparison to VC, the implementation of the schemes for VC and the high level CME will use custom built code based around binary strings to function. This may result in the implementations for ECC, AES, RC4 and the byte-oriented version of CME having an advantage in terms of efficiency, over those of VC and the high-level string-oriented CME scheme, and as such the results will not be comparable between these classes.

Another limitation of this study is that one of the chosen comparative algorithms, ECC, is an asymmetric cryptographic system, while the others are symmetric. The rising level of research and implementation of ECC as an option for encryption makes it an

important component of any study relating to the exploration of graphic methods in cryptography, but the results of the study will need to take the difference in structure into account, and discuss the affect this difference will have on the data. The difference in structure may impact the overall results for the ECC algorithm. ECC is also used for key exchange in symmetric encryption systems, through the ECC Diffie-Hellman protocol. As such, the comparisons between ECC and CME will be based around the efficiency of performance in the set up process, and the security as it relates to the ease of key generation, rather than the result of encryption, due to ECC's use in generating keys for use in other systems.

It should be noted that the implementation of VC in this study is altered to perform encryption on binary strings. Typical VC schemes operate on black and white share images, with each pixel transformed into a subset of black and white subpixels. The superimposition of these subpixels upon one another then recreates the original pixel. For the purposes of this research, the pixels are replaced by bits, and each bit is transformed into a set of bits referred to as a subpixel array during the share creation process. The secret image and the share images are all therefore replaced by binary strings of bits, and the share strings can then be combined to recreate the original secret string. The translation of these VC schemes into binary encryption algorithms is a trivial one, but the change in domain for the algorithm may possibly affect its results. This alteration should therefore be kept in mind for the results of the VC algorithm implementation, and in the following discussion of those results.

In relation to the use of RC4 for comparative purposes, it must be noted that RC4 has been proven insecure, and is no longer recommended for use (Rivest & Schuldt, 2014). However, it is still the best researched and most widely studied stream cipher, and as it is freely available with no restrictions or patents, it has been chosen as the classical stream cipher. It is necessary to include a stream cipher comparison in the research, as CME is classed as a word-oriented, symmetric stream cipher.

Further research opportunities are presented by these limitations, such as a study of the equivalent hardware implementations of the tested algorithms or of the porting of the proposed CME system into a low-level hardware implementation for alternative domains. The limitations also affect the ability to compare results outside the designated pairs. For example, the results from testing the binary string version of VC will not be comparable with those of the byte level implementation of AES. The use of a high-level implementation for the VC algorithm will give results that are unlikely to generalize well to lower level implementations. These research possibilities and the effect the limitations



had on the raw testing data will be discussed in greater detail in Chapter 6.

### **3.5 CONCLUSION**

In this chapter, the overall design and methodology of the research has been explained, and the benchmarks for the analysis of each comparative algorithm have been set based on those set by previous studies in the area of encryption algorithms. The research questions, hypotheses and motivations have been discussed, and the proposed coordinate matrix system has been laid out. Limitations of the study design have been discussed, to ensure the reader is aware of the impact these may have on the resulting data in the following chapters.

In the next chapter, the results of the study will be given, and the data reviewed based on the framework outlined in this chapter. The gathered data will be presented individually for each algorithm, followed by the comparative results.

# **Chapter 4**

## **Research Findings**

### **4.0 INTRODUCTION**

In Chapter 3, a framework for testing comparable algorithms with regards to efficiency and security was proposed. This gave standards to evaluate the chosen ECC, VC, AES and RC4 schemes against, as well as the newly proposed Coordinate Matrix Encryption scheme.

In this chapter, the results from the testing framework from Chapter 3 are given, first individually for each algorithm, and then the overall comparative results. Section 4.1 gives the results for the CME algorithm, followed by the results for AES in section 4.2. Section 4.3 details the results for ECC, section 4.4 gives the results for VC, followed by the results for RC4 in section 4.5. Section 4.6 provides the overall comparison between the different schemes, broken down into efficiency and security components, and finally, section 4.7 gives the conclusion and a summary of the results.

### **4.1 COORDINATE MATRIX ENCRYPTION**

This section describes the results of testing the Coordinate Matrix Encryption scheme (CME), proposed in Chapter 3. Section 4.1.1 details the specifics of the implementation used for the algorithm which operates on binary strings, while 4.1.2 gives the details of the implementation which is based around byte arrays. Section 4.1.3 describes the results from the tests derived to evaluate the algorithms' efficiency. Finally, section 4.1.4 gives the results for the tests for the algorithms' security.

#### **4.1.1 Implementation Details for CME on Binary Strings**

The implementation for the CME scheme was completed in Java. The scheme utilized 2-dimensional arrays (referred to hereafter as 'key matrices') of a Coordinate Entry class to achieve the required high-level software implementation. This implementation was discussed briefly in section 3.2.4, and the specifics of the design are elaborated here.

The first step of the implementation was the generation of all possible bit strings

for the selected bit length. Once these were enumerated, each string was assigned to an instance of the Coordinate Entry class. These contained several elements, including a Boolean isEmpty value, a list of all matrix locations for that string, and the bit string value.

The key matrix was generated using a pseudo random number generator (PRNG) to choose multiple different locations for each of the bit strings. The PRNG gave two pseudo-random numbers, used as  $x$  and  $y$  coordinates in the matrix, which were then checked by the algorithm. If the location was empty, the bit string in question was inserted into that location in the matrix. If not, another location was generated by the PRNG. Once the matrix had been generated, all blank locations were enumerated in a separate list of blank entries. The total number of blank entries was assigned to be equal to the number of all possible bit strings, and then each blank entry was assigned multiple coordinates in the same method as the bit string entries. Once all blank entries had been assigned the requisite number of empty coordinates, the key string was calculated. This was a single binary string containing the first  $x$  coordinate for each of the bit strings. Because these coordinates were randomly generated as part of the key matrix setup, and only the pair of  $[x,y]$  coordinates was required to be unique, the key string gives the equivalent of a PRNG bit key, of length  $2^n$ , where  $n$  is the number of all possible bit strings.

The encryption in the CME scheme took the plaintext and converted it into a binary string if it was not already in binary format. The first operation of the encryption process was an exclusive-OR, in which the binary encoding of the plaintext was combined with the binary key string. The key string and plaintext were combined using an exclusive-OR operation bit by bit, with the key string wrapping around to the start if the plaintext length exceeds its own. The encryption process moved along the resulting string in sections of  $n$ -bits at a time. A coin toss operation utilizing a PRNG decided if the next part of the ciphertext was to be an encrypted piece of the message, or a blank padding coordinate. If the coin toss decided to add a padding character, another PRNG picked a single blank coordinate pair location from the location list of a randomly chosen blank entry. If the coin toss chose to encrypt the next bit string of the message, the PRNG chose a random location from the list of coordinates containing that particular bit string. The algorithm then added the two randomly chosen  $[x, y]$  coordinates in binary representation to the string of ciphertext. The encryption process continues until all message characters have been enciphered, and an equal number of padding coordinates have been placed in the string.

Decryption in the CME scheme is based on entry lookup. For each of the

coordinates in the ciphertext, the algorithm checks the value in the given location in the key matrix. If the coordinates are empty, that part of the ciphertext is discarded. If the coordinates contain a bit string value, that value is added to the plaintext. Once all coordinates have been checked, the recovered string is then combined with the key string using exclusive-OR. Upon completion of the lookup and application of the key string, the original plaintext has been recovered. Because the process of decryption is based on lookup operations, the overall time complexity for decryption in the CME implementation is theoretically lower than that of encryption.

#### **4.1.2 Implementation Details for CME based on Byte Arrays**

The CME scheme described in 4.1.1 allows for the testing of different sizes of bit schemes, however, as it is a high-level implementation which operates on binary strings, the overall efficiency of the implementation will automatically be significantly lower than those inbuilt Java functions for encryption systems such as AES, which perform operations on arrays of bytes. In order to provide an accurate picture of the comparative efficiency of the schemes, a version of CME using a fixed 8-bit scheme size was developed which operates on arrays of bytes, instead of processing binary strings. This lower level implementation of the CME scheme is therefore able to operate in a fashion comparable with that of the inbuilt Java functions for encryption using AES and key generation using ECC.

The overall CME Byte Array scheme operates in a similar implementation to that described in 4.1.1. The set of all possible binary strings is stored in a matrix using a modified version of the Coordinate Entry class, which stores each bit string as its corresponding byte value. The data to be encrypted does not require padding, as each UTF-8 character is translated into its corresponding byte value between  $[-128, 127]$ . The ciphertext is then composed of an array of integer values from  $[0, 255]$ , and every pair of integer values gives the  $x$  and  $y$  locations of an entry in the key matrix. Unlike the high-level implementation, the exclusive-OR operation on the plaintext is done byte-by-byte while the coordinate encryption process occurs, rather than separately beforehand. The encryption process moves along the array and either inputs a set of two integer coordinates for the relevant byte value, or a set of coordinates for a randomly selected empty padding location. This integer array is then transmitted as the ciphertext.

The decryption process reverses the encryption by checking the location based on every two entries in the integer array that composes the ciphertext, with the entries assigned as  $x$  and  $y$  respectively. If a byte value exists at the location, it is combined with

the current integer value of the key string, and the result of this exclusive-OR operation is converted into a byte and added to the byte array which composes the plaintext. Once all coordinates have been checked, the plaintext byte array is translated back into the UTF-8 characters encoded by each byte value, and displayed.

The full source code for both implementations of the CME schemes can be found in Appendix B.

#### 4.1.3 Efficiency

The efficiency of the CME scheme was measured using several different methods, as per the research design laid out in Chapter 3. The time for set up and key generation, the time to encrypt and to decrypt the data, and the overall memory used in the running of the implementation were measured. Each piece of data was encrypted and decrypted 1000 times in the byte implementation, and 500 times in the bit-string implementation. The average time and memory for each of these iterations was recorded, and averages for each data size were then calculated. The testing data used is available in Appendix C. The UTF-8 encoded plaintext used for testing the byte implementation of CME was taken from *Hamlet* (Shakespeare & Ackroyd, 2006) and *Pride and Prejudice* (Austen, 2006). Source code for the testing programs is available in Appendix B.

The time requirements for the byte implementation of CME increased linearly with the data size, as is to be expected of a stream cipher which operates on a piece of data byte-by-byte. Table 4.1 gives the mean encryption and decryption times for each of the tested data sizes.

<b>Data Size</b>	<b>Average Encryption (ms)</b>	<b>Average Decryption (ms)</b>
<b>304</b>	0.031	0.012
<b>928</b>	0.059	0.023
<b>3024</b>	0.136	0.065
<b>4408</b>	0.173	0.050
<b>8144</b>	0.262	0.093

*Table 4.1: Mean encryption/decryption times for byte CME (3d.p.)*

The time requirements for the bit-string based implementation were higher, due to the nature of the operations performed. As the implementation used string variables and performed various permutation and substitution operations on them, the time taken was noticeably longer. The mean time requirements as calculated over 500 iterations are shown in Table 4.2.

<b>Data Size (bits)</b>	<b>Average Encryption (ms)</b>	<b>Average Decryption (ms)</b>
<b>16</b>	0.020	0.026
<b>32</b>	0.066	0.036
<b>64</b>	0.104	0.060
<b>128</b>	0.214	0.074
<b>256</b>	0.396	0.136
<b>512</b>	1.130	0.328

*Table 4.2: Mean encryption/decryption times for 4-bit string CME (3d.p.)*

The memory requirements for the byte implementation of CME were tested by measurement of the total memory occupied within the Java Virtual Machine environment during each task: set up of the scheme and key matrix; encryption; decryption. The set up and key generation was tested 100 times, with the memory requirement in megabytes and the time taken in milliseconds recorded. These mean of these results was then calculated. Table 4.3 shows the results for the scheme setup.

<b>Memory used (MB):</b>	1.217
<b>Time taken (ms):</b>	80.130

*Table 4.3: Mean setup time and memory for byte CME (3d.p.)*

The memory requirements for encryption and decryption in the byte version of CME were measured for each of the data string sizes, over the course of 1000 encryption and decryption iterations with a single key. Table 4.4 shows the mean results of these tests. Similarly to the time requirements, the increase is linear with the size of the encrypted data.

<b>Data Size</b>	<b>Average Encryption (MB)</b>	<b>Average Decryption (MB)</b>
<b>304</b>	1.244	1.244
<b>928</b>	1.245	1.246
<b>3024</b>	1.251	1.251
<b>4408</b>	1.255	1.255
<b>8144</b>	1.263	1.264

*Table 4.4: Mean encryption/decryption memory for byte CME (3d.p.)*

The memory requirements for the bit-string version of CME were tested similarly. Setup

time and occupied memory for the JVM were recorded for 100 iterations, and the mean calculated. Table 4.5 gives these results.

<b>Memory used (MB):</b>	0.448
<b>Time taken (ms):</b>	21.440

*Table 4.5: Mean setup time and memory for 4-bit string CME (3d.p.)*

As the bit string implementation operated on a four bit scheme size with a 16-by-16 key matrix, the overall requirements for creating and storing the key were less than that of the eight bit scheme used in the byte implementation, which utilized a 256-by-256 key matrix.

The memory requirements for encryption and decryption in the 4-bit string implementation were measured over 500 iterations with a single key for each of the different data sizes of the pseudo-random bit strings. The mean of these results was then calculated for each data size. Table 4.6 gives these results.

<b>Data Size (bits)</b>	<b>Average Encryption (MB)</b>	<b>Average Decryption (MB)</b>
<b>16</b>	0.475	0.475
<b>32</b>	0.476	0.478
<b>64</b>	0.476	0.476
<b>128</b>	0.478	0.478
<b>256</b>	0.480	0.480
<b>512</b>	0.484	0.484

*Table 4.6: Mean encryption/decryption memory required for 4-bit string CME (3d.p.)*

#### **4.1.4 Security**

The CME scheme is quantifiably resistant to brute force attacks for implementations equal to or greater than a given size of bit scheme. This is because of the relative size of the matrix, and the number of possible values within each location of the matrix. On average, a brute force attack requires that an adversary attempt  $\frac{1}{2}$  of all possible keys. The number of possible matrix keys for a given bit scheme is shown in Equation 4.1.4.i.

$$\text{Equation 4.1.4.i} \quad \text{bit string length} = 2^n$$

$$M_{\text{atrices}} = (\text{bit string length} + 1)^{\text{bit string length}^2}$$

Thus, if an adversary attempted to brute force a 4-bit CME scheme, the number of matrices that they would be required to try, on average would be:

$$\text{Equation 4.1.4.ii} \quad M_{\text{atrix attempts}} \approx \frac{1}{2}(17^{256})$$

$$M_{matrix\ attempts} \approx \frac{1}{2}(9.883798 \times 10^{314})$$

$$M_{matrix\ attempts} \approx 4.941899 \times 10^{314}$$

The result of Equation 4.1.4.ii is so large it is impractical to compute with a regular calculator, as it exceeds the maximum allowable value for an IEEE float. As such, the variable point integer toolbox in MatLab was necessary to compute the result. For a more effective implementation, using an 8-bit byte-oriented scheme, the possible matrix keys are  $257^{65536}$ . The average number of key matrix attempts which a brute force attack would require to break an 8-bit scheme is given in Equation 4.1.4.iii.

$$\text{Equation 4.1.4.iii} \quad M_{matrix\ attempts} \approx \frac{1}{2}(257^{65536})$$

$$M_{matrix\ attempts} \approx \frac{1}{2}(2.3832557 \times 10^{157937})$$

$$M_{matrix\ attempts} \approx 1.19162785 \times 10^{157937}$$

A side effect of the number of possible matrices is that it is statistically likely that, were it possible to try all matrix keys for a given scheme, the adversary would likely encounter more than one key matrix that resulted in an intelligible plaintext. This is further exacerbated by the exclusive-OR operation which utilizes the first generated  $x$  location for each of the bit-strings/bytes, as an adversary would also be required to try the different possible key strings for each key matrix. The adversary would then be required to work out which key was the correct one. The number of possible key strings within each key matrix for a given bit scheme size is given in Equation 4.1.4.iv.

$$\text{Equation 4.1.4.iv} \quad k_{eystrings} = locations^{bit\ strings}$$

$$k_{eystrings} = ((2^{2^n}/2(2^n))^{2^n}$$

For a bit scheme of size 4, the number of key strings an adversary would be required to try for each attempted key matrix is shown in Equation 4.1.4.v. The adversary would need to try all key strings for the incorrect key matrices, and approximately half of the key strings for the correct key matrix. The average number of key string attempts for the correct key matrix is given for a 4-bit scheme in Equation 4.1.4.vi.

$$\text{Equation 4.1.4.v} \quad k_{eystrings} = ((2^{2^n}/2(2^n))^{2^n}$$

$$k_{eystrings} = ((2^{2^4})/2(2^4))^{2^4}$$

$$k_{eystrings} = 8^{16}$$

$$k_{eystrings} = 2.814749767 \times 10^{14}$$



$$\text{Equation 4.1.4.vi} \quad BF_{\text{keystrings}} \approx \frac{1}{2}(2.814749767 \times 10^{14})$$

$$BF_{\text{keystrings}} \approx 1.407374884 \times 10^{14}$$

Given the number of possible key strings, the approximate total average number of operations to attempt to brute force a 4-bit CME scheme is given in Equation 4.1.4.vii.

$$\text{Equation 4.1.4.viii} \quad BF_{\text{total}} \approx (S_{\text{strings}} \cdot (\frac{1}{2} \cdot (m_{\text{matrices}}) - 1)) + (\frac{1}{2} \cdot S_{\text{strings}})$$

$$BF_{\text{total}} \approx (8^{16} \cdot (\frac{1}{2} \cdot (17^{256}) - 1)) + (\frac{1}{2} \cdot (8^{16}))$$

$$BF_{\text{total}} \approx (2.814749767 \times 10^{14}) \cdot (4.941899 \times 10^{314} - 1) + (1.407374884 \times 10^{14})$$

$$BF_{\text{total}} \approx 1.391020944 \times 10^{329}$$

One of the key features of the algorithm is its diffusion of any statistical properties of the plaintext, by using multiple locations for each bit string in the matrix and by the exclusive-OR operation performed prior to the coordinate encryption. The matrix is set up so that exactly half is blank locations, and the other half is bit strings. During the encryption of the plaintext, whether the algorithm decides to insert one of the blank locations or one of the locations for the next bit string to encrypt is decided by a coin toss procedure which uses a PRNG. Because of this, the likelihood of any particular blank padding location occurring is approximately equal to the likelihood of any particular bit string location occurring. The calculation of the number of locations per string is shown in Equation 4.1.1.ix.

$$\text{Equation 4.1.4.ix} \quad M_{\text{size}} = (2^n)^2$$

$$N_{\text{locations per string}} = \frac{M_{\text{size}}}{2(2^n)}$$

As any given coordinate is equally likely to be an empty padding location as it is a bit string, even given unlimited computing power, an adversary would be required to guess, with a ½ chance of guessing correctly for each string, whether a coordinate contained a part of the message, or was simply padding material. Because of the addition of the exclusive-OR operation with the key string prior to turning the message into encrypted coordinates, the frequency information of the original text is not reflected in the result. The likelihood of correctly guessing for all coordinates of any given string is given in Equation 4.1.3.x, the binomial probability formula from the Bernoulli Trials.

$$\text{Equation 4.1.4.x} \quad P(k \text{ successes in } n \text{ trials}) = \binom{n}{k} p^k q^{n-k}$$

where  $n$  is the number of trials,  $k$  is the number of successes,  $(n-k)$  is the number of failures,  $p$  is the probability of success in one trial, and  $q = 1-p$  which is the probability

of failure in one trial. Equation 4.1.4.xi gives the probability of correctly guessing whether a coordinate is full or blank for all coordinates of a 20 coordinate ciphertext string in a single attempt.

$$\begin{aligned} \text{Equation 4.1.4.xi} \quad P(20) &= \binom{20}{20} \cdot (0.5)^{20} \cdot (0.5)^0 \\ P(20) &= 9.536743164 \times 10^{-7} \end{aligned}$$

The smallest plaintext string used to test the byte version of CME was 304 bits. This resulted in a ciphertext of 1216 bits, or 608 coordinates. Equation 4.1.4.xii gives the probability of successfully guessing for each of the 608 coordinates whether they are padding or full.

$$\begin{aligned} \text{Equation 4.1.4.xii} \quad P(608) &= \binom{608}{608} \cdot (0.5)^{608} \cdot (0.5)^0 \\ P(608) &= 9.41374947 \times 10^{-184} \end{aligned}$$

The CME scheme avoids frequency analysis through the combination of the exclusive-OR operation prior to encryption, and the multiple locations assigned to each bit string. The creation of the list of blank entries, which mirrors exactly that of the list of occupied coordinates, gives another layer of confusion to the statistics of the resulting data. The scheme was analysed through a specially developed frequency analysis program, which took the ciphertext input, as well as the key matrix, and calculated the frequency of occurrences for each blank and full coordinate. The frequency analysis program then output the statistics for how many of the singular occurrences were blank coordinates, how many were full coordinates; how many of the double occurrences were blank or full coordinates; and so on. The program was run on the CME byte implementation, using the plaintext strings of differing sizes, and each string was used to generate one thousand ciphertexts from a single key. The results of the frequency analysis for each of these ciphertexts were then tabulated, and the averages mapped.

Of the occurrences, coordinates that occurred once were equally likely to be blank or full, as were secondary occurrences. Third and fourth occurrences in full versus blank coordinates were within 10% of one another. Table 4.7 shows the average frequency information for the largest of the testing data, an English language string of 8816 bits, as well as the number of times a ciphertext gave that frequency of occurrences within the thousand tests. The overall frequencies are fairly flat, and as they reflect the statistics of an already encrypted piece of data, give little to no information on the underlying plaintext. In addition, the greater the number of times the particular frequency occurred, the flatter the distribution became. As such, it seems likely that more extensive testing

would result in even flatter distributions.

<b>Frequency</b>	<b>Blank</b>	<b>Full</b>	<b>Times Occurred</b>
<b>1</b>	50.047%	49.953%	1000
<b>2</b>	48.589%	51.411%	1000
<b>3</b>	46.491%	53.508%	308
<b>4</b>	40.000%	60.000%	5

*Table 4.7: Frequency analysis of ciphertext from an 8816-bit string. (3d.p.)*

The addition of empty padding coordinates helps protect the scheme against known and chosen plaintext attacks, as knowing what the plaintext message is does not give any further information about which coordinates in the ciphertext are empty padding, and which are locations for plaintext strings. The exclusive-OR operation performed on each bit string or byte assists in stripping statistical properties from the data, and protects the scheme from frequency analysis. An adversary with unlimited computing power would still need to venture a guess whether each coordinate was empty or not.

In regards to chosen plaintext attacks, the CME byte scheme gave a similar distribution when a 4048-bit string of a single repeated character was used. Of the thousand variable ciphertexts resulting in the encryption process with the same key, only 46 ciphertexts contained coordinates that were repeated more than twice. Table 4.8 shows the average results for the analysis of the chosen plaintext attack. Because there are so few repeated coordinates, and blank coordinates also occur in the repetitions, a chosen plaintext attack would likely give little information about the possible key string or matrix. Also, any information about the repeated characters within the key string would not be subject to known frequency information, as the key string is generated randomly.

<b>Frequency</b>	<b>Blank</b>	<b>Full</b>	<b>Times Occurred</b>
<b>1</b>	50.105%	49.895%	1000
<b>2</b>	44.127%	55.873%	1000
<b>3</b>	31.915%	68.085%	46

*Table 4.8: Frequency analysis of ciphertext from a 4048-bit chosen plaintext string. (3d.p.)*

The results from the encryption of a single string of repeated characters can then be compared to those results from analysis of ciphertext from a 4408-bit English language plaintext string, shown in Table 4.9. The overall distributions are similar, and minor

variations may be accounted for by the differing key strings and key matrices.

<b>Frequency</b>	<b>Blank</b>	<b>Full</b>	<b>Times Occurred</b>
<b>1</b>	50.015%	49.985%	1000
<b>2</b>	49.109%	50.891%	1000
<b>3</b>	53.704%	46.296%	54

*Table 4.9: Frequency analysis of ciphertext from a 4408-bit string. (3d.p.)*

The final measure of security was the avalanche effect of the algorithm. Because the scheme results in a different ciphertext almost each time the same plaintext is encrypted by the same key, the CME scheme provides a good avalanche effect, with under 0.5% of bytes unchanged from the previous ciphertext. The avalanche effect was measured in the byte implementation by the total percentage of the same bytes occurring, and the percentage of the same bytes occurring in the same position. For the sake of comparison, these variables were measured over 1000 iterations of encryption/decryption on data that varied by a single bit, and over 1000 iterations on the same piece of data. Table 4.10 gives the results of the changed and unchanged data averaged over these tests.

<b>Data Size</b>	<b>Unchanged from Previous</b>		<b>1-bit Altered from Previous</b>	
	<b>Same Bytes</b>	<b>Same Position</b>	<b>Same Bytes</b>	<b>Same Position</b>
<b>304</b>	44.653%	0.441%	44.839%	0.414%
<b>928</b>	84.064%	0.419%	84.026%	0.388%
<b>3024</b>	99.722%	0.390%	99.713%	0.422%
<b>4408</b>	99.973%	0.400%	99.984%	0.404%
<b>8144</b>	100.000%	0.396%	100.000%	0.395%

*Table 4.10: Avalanche effect in byte CME. (3d.p.)*

## **4.2 ADVANCED ENCRYPTION STANDARD**

This section discusses the details of the low level software implementation of AES, as well as the results for efficiency and security based on the testing criteria. Section 4.2.1 gives a detailed discussion of the implementation used. Section 4.2.2 discusses the AES efficiency, based on the testing data. Finally, section 4.2.3 gives an overview of the AES security, using the theoretical underpinnings of the algorithm, as well as the practical data from testing results.

### 4.2.1 Implementation Details

The implementation for AES encryption utilized the inbuilt standards of the `javax.crypto` and `java.security` packages available as part of the JSE 7. The scheme used the inbuilt `SecureRandom` function to generate a random 128-bit key and the Initialisation Vector. The algorithm took as input a UTF-8 encoded plaintext string, which was then converted into a byte array and encrypted using Java's Cipher function, utilizing the parameters for AES Encryption, CBC mode, and PKCS5 Padding. This function returned a byte array as the ciphertext. Decryption was then completed using the same parameters with the Cipher function, and the resulting plaintext byte array was converted back into its UTF-8 encoded string.

The mode of encryption was chosen as CBC to provide better security against methods of statistical analysis, as studies suggest that use of ECB mode can result in repeated blocks of ciphertext, and it is the least secure mode of operation for AES implementations (Thakur & Kumar, 2011). CBC mode combines each block of ciphertext with the next consecutive block of plaintext, and as such prevents the repetition of blocks. The full source code used for the AES implementation of encryption and decryption methods can be found in Appendix B.

### 4.2.2 Efficiency

The efficiency of AES was measured by the time taken to setup the scheme and key, encrypt the plaintext, and decrypt the ciphertext. The memory occupied in the JVM at each of these stages was also recorded. The averages were then calculated. This process was done for each of the different data sizes. The testing data used is available in Appendix C. The source code for testing programs can be found in Appendix B.

The time taken for encryption and decryption was measured for each of five data sizes, with each piece of data encrypted and decrypted 1000 times. The time taken for each iteration was tabulated. The means are shown in Table 4.11.

Data Size	Average Encryption (ms)	Average Decryption (ms)
304	0.199	0.170
928	0.142	0.182
3024	0.173	0.179
4408	0.185	0.196
8144	0.148	0.250

Table 4.11: Mean encryption/decryption times for 128-bit AES (3d.p.)

The time taken and memory used for setting up the implementation were also measured. The scheme setup was completed 100 times, and the results for occupied memory in megabytes and time taken in milliseconds were recorded. Table 4.12 gives these results.

<b>Memory used (MB):</b>	2.364
<b>Time taken (ms):</b>	409.000

*Table 4.12: Mean setup time and memory required for 128-bit AES (3d.p.)*

The memory occupied by the JVM for encryption and decryption were measured over 1000 iterations on each of the different strings of data. Table 4.13 gives the results of these tests.

<b>Data Size</b>	<b>Average Encryption (MB)</b>	<b>Average Decryption (MB)</b>
<b>304</b>	1.415	1.393
<b>928</b>	1.416	1.399
<b>3024</b>	1.417	1.414
<b>4408</b>	1.417	1.422
<b>8144</b>	1.419	1.442

*Table 4.13: Mean memory required for encryption/decryption in 128-bit AES (3d.p.)*

### 4.2.3 Security

Security in AES was tested using the schema defined in Chapter 3. The operations required to brute force attack the implementation, vulnerability to known and chosen plaintext, frequency analysis, and the avalanche effect were examined. This allowed the analysis of the overall security level of the scheme. The testing data used is available in Appendix C.

A brute force attack on AES relies on attempting the different possible keys. 128-bit AES has a total key space of  $2^{128}$ . A brute force attack in general requires that one half of all possible keys are attempted. The average number of attempts for a brute force attack on AES 128 are detailed in Equation 4.2.3.i.

$$\begin{aligned}
 \text{Equation 4.2.3.i} \quad \text{operations} &\approx \frac{1}{2} 2^{128} \\
 \text{operations} &\approx \frac{1}{2} (3.4028237 \times 10^{38}) \\
 \text{operations} &\approx 1.7014118 \times 10^{38}
 \end{aligned}$$

The number of operations for 128-bit AES as shown in Equation 4.2.3.i is considered to

be impracticably large for an attacker to attempt given current technological standards.

AES was designed to be resistant to frequency analysis. The multiple round keys that are added in each of the 12 rounds of 128-bit AES as well as the substitutions and permutations are employed to destroy the statistical properties of the plaintext. Frequency analysis was performed on the ciphertext resulting from the AES encryption scheme on each of the five plaintext strings. Table 4.14 gives the results of frequency analysis on the ciphertext of a 8144-bit plaintext string, over the course of 1000 encryptions and decryptions, each with a different key – as AES results in a singular mapping, the same ciphertext for the same plaintext encrypted with the same key.

<b>Frequency</b>	<b>Average # of Bytes</b>	<b>Times Occurred</b>
<b>1</b>	18.626	1000
<b>2</b>	37.606	1000
<b>3</b>	50.462	1000
<b>4</b>	50.010	1000
<b>5</b>	39.938	1000
<b>6</b>	26.581	1000
<b>7</b>	15.194	1000
<b>8</b>	7.554	1000
<b>9</b>	3.403	978
<b>10</b>	1.780	749
<b>11</b>	1.271	388
<b>12</b>	1.068	146
<b>13</b>	1.019	54
<b>14</b>	1	18
<b>15</b>	1	4
<b>16</b>	1	1

*Table 4.14: Frequency analysis of ciphertext from a 8144-bit string in 128-bit AES (3d.p.)*

The distribution of frequencies occurs over a curve, with higher numbers of bytes at the 3<sup>rd</sup> and 4<sup>th</sup> frequencies than at the singular and double frequencies. The lowest occurrences are at the highest three frequencies.

AES was designed to be resistant to all forms of chosen and known plaintext analysis. As a measure of this resistance, frequency analysis was also performed on the ciphertext resulting from a 4048-bit string consisting of a single repeated character over 1000 iterations of encryption and decryption, each with a different key. The number of

occurrences of each frequency were then measured and averaged. Table 4.15 shows the averaged results of the chosen plaintext attack analysis.

<b>Frequency</b>	<b>Average # of Bytes</b>	<b>Times Occurred</b>
<b>1</b>	69.115	1000
<b>2</b>	69.620	1000
<b>3</b>	46.234	1000
<b>4</b>	23.054	1000
<b>5</b>	9.207	1000
<b>6</b>	3.070	963
<b>7</b>	1.491	561
<b>8</b>	1.098	184
<b>9</b>	1.026	38
<b>10</b>	1	13

*Table 4.15: Frequency analysis of ciphertext from a 4048-bit single character string in 128-bit AES (3d.p.)*

The results of the frequency analysis on the ciphertext of the chosen plaintext string can be contrasted with the results of testing on the ciphertext from a 4408-bit string of plaintext data. Table 4.16 gives the frequency analysis results of testing on the ciphertext of a 4408-bit string.

<b>Frequency</b>	<b>Average # of Bytes</b>	<b>Times Occurred</b>
<b>1</b>	62.636	1000
<b>2</b>	69.346	1000
<b>3</b>	49.979	1000
<b>4</b>	27.471	1000
<b>5</b>	11.751	1000
<b>6</b>	4.234	990
<b>7</b>	1.794	757
<b>8</b>	1.170	317
<b>9</b>	1.049	81
<b>10</b>	1	16
<b>11</b>	1	1

*Table 4.16: Frequency analysis of ciphertext from a 4408-bit string in 128-bit AES (3d.p.)*

The final security measurement was that of the avalanche effect of the algorithm. Each



of the plaintext data sizes was altered by one bit for each encryption/decryption iteration, and the percentage of the same bytes, as well as the percentage of bytes occurring in the same position were measured over the course of 500 iterations. The results were then tabulated and the mean calculated. Table 4.17 gives the results of the avalanche testing.

Data Size	1-bit Altered from Previous	
	Same Bytes	Same Position
<b>304</b>	37.767%	24.779%
<b>928</b>	62.777%	38.905%
<b>3024</b>	87.935%	45.857%
<b>4408</b>	94.276%	48.227%
<b>8144</b>	99.100%	48.593%

Table 4.17: Avalanche effect in 128-bit AES (3d.p.)

### 4.3 ELLIPTIC CURVE CRYPTOGRAPHY

This section gives the details and basis for the implementation of the ECC scheme. Section 4.3.1 gives a detailed overview of the way the implementation was created, followed by a discussion of its efficiency in section 4.3.2, based on the test results from the research design defined in Chapter 3. Section 4.3.3 then discusses the scheme's security, based both on the theoretical analysis of the basis of the algorithm, and the practical data from testing.

#### 4.3.1 Implementation Details

The implementation used for performing the ECC key generation and ECC Diffie-Hellman key exchange is based on that given by Martinez and Encinas (2013). The authors describe the use of the inbuilt functions of JSE 7 which allow for the simple implementation of ECC protocols over predefined curves. The curve used for the purposes of this research was the Java curve *secp192r1*, which is given in the standards for ANSI X9.62 as X9.62 prime192v1, and the NIST FIPS 186-2 standard as NIST P-192. This curve is defined over a prime field  $\mathbb{F}_p$ , and has an equivalent security level of 192 bits. The equation for the finite prime field of this elliptic curve, the field polynomial is given in Equation 4.3.1.i, as specified in NIST (2000).

$$\text{Equation 4.3.1.i} \quad p = 2^{192} - 2^{64} - 1$$

Martinez and Encinas (2013) provide examples of source code for implementing these

curves in the java.security package included in JSE 7, and this code was modified for use in the comparative analysis. The full source code used for the ECC protocols can be found in Appendix B.

### 4.3.2 Efficiency

The efficiency of the ECC implementation was tested using the schema laid out in Chapter 3. Time for the ECDH scheme setup and the memory occupied by the JVM during setup were measured over multiple iterations of the algorithm. These results were then tabulated.

The setup time was measured in milliseconds over 100 iterations, and gave the average time requirement for the ECDH protocol to complete, with the generation of individual private-public key pairs, and the calculation of a common key. Concurrently, the total memory requirement was calculated using the occupied memory in the JVM during this process. Table 4.18 gives the average results of this testing.

<b>Memory used (MB):</b>	1.192
<b>Time taken (ms):</b>	359.500

*Table 4.18: Memory and time requirements for execution of ECDH protocol (3d.p.)*

### 4.3.3 Security

The implemented ECDH scheme results in two separate sets of keys: the original public-private key pair generated for each user from the curve; and the shared symmetric key generated using the two public-private key pairs during the Diffie-Hellman key exchange. The generated symmetric key is 128-bits, which gives the symmetric key the same level of security against brute force attacks as 128-bit AES, as per the average number of operations calculated in Equation 4.3.3.i.

$$\begin{aligned}
 \text{Equation 4.3.3.i} \quad \text{operations} &\approx \frac{1}{2} 2^{128} \\
 \text{operations} &\approx \frac{1}{2} (3.4028237 \times 10^{38}) \\
 \text{operations} &\approx 1.7014118 \times 10^{38}
 \end{aligned}$$

The public-private key pairs are secured by the difficulty of computing discrete logarithms on elliptic curves. The curve used in the implementation gives a relative security level of 192-bits. According to Stallings (2014, p. 296) the fastest current method for solving the elliptic curve discrete logarithm problem is the Pollard rho method, which gives 192-bit ECC the equivalent security of 1024-bit RSA, and of an 80-bit symmetric encryption algorithm. The average number of operations to brute force the 192-bit ECC is given in Equation 4.3.3.ii, and the number of operations using the Pollard rho method

is given in Equation 4.3.3.iii.

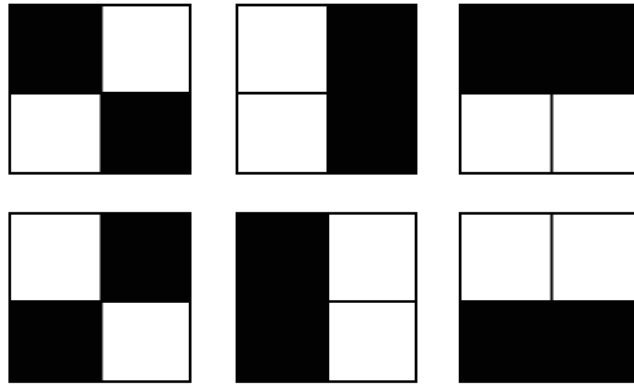
$$\begin{aligned} \text{Equation 4.3.3.ii} \quad BF_{operations} &\approx \frac{1}{2}(2^{192}) \\ BF_{operations} &\approx \frac{1}{2}(6.2771017 \times 10^{57}) \\ BF_{operations} &\approx 3.1385509 \times 10^{57} \\ \text{Equation 4.3.3.iii} \quad PR_{operations} &\approx 2^{80} \\ PR_{operations} &\approx 1.2089258 \times 10^{24} \end{aligned}$$

## 4.4 VISUAL CRYPTOGRAPHY

This section discusses the details for the implementation of the binary version of the 2-out-of-2 VC scheme. Section 4.4.1 details how the implementation was created, and the equivalence to classical VC schemes. Section 4.4.2 then discusses the efficiency of the scheme, based on the results gained from testing. Finally, section 4.4.3 gives the analysis of the scheme's security, both in theory and based on the results of the testing data.

### 4.4.1 Implementation Details

The implementation for the VC algorithm was based on the 2-out-of-2 scheme originally proposed by Naor and Shamir (1995) with the particulars of share generation based on random grid VC as proposed by Kafri and Keren (1988). The implementation designed for this research took a binary string of plaintext as its input, and then generated two shares from the string. In an image-based VC scheme, each single pixel is split into several subpixels, and each subpixel is either black or white. The contrast constraint is the feature of these schemes that makes the pixel either black or white once the subpixels in each share are recombined. For the purposes of this research, the contrast constraint was set to  $\frac{1}{2}$ , so that for a given bit to be considered 'white', represented by a 0, the recombined set of subpixels must be  $\frac{1}{2}$  white, or  $\frac{1}{2}$  0 bits. The subpixels were assigned six possible states using 4-bit strings of binary digits as representation of a two-by-two matrix of visual pixels. These possible states are shown in Fig 4.1. A black subpixel was represented by a 1 bit, and a white subpixel by a 0 bit. For example, the top leftmost subpixel state in Fig 4.1 would be represented by "1001", as the top row has a black then white subpixel, and the bottom row a white then black subpixel. Strings were organized left to right by column then top to bottom by row.



*Figure 4.1: A visual representation of the six possible subpixel states for the implemented VC scheme.*

The six possible subpixel states were all assigned to two separate string arrays, with the second array containing the states in a different order: the opposite subpixel state was assigned to the same location as that of the first array, so that if the subpixel states from the same location in the first and second arrays were combined, the result would be a fully black pixel – represented in the code by a full string of 1s, “1111”. The pairs of opposite subpixel states are shown in Fig 4.4.1.i by column – the top leftmost state is the opposite of the bottom leftmost state, and so on. When transposed one on top of the other, these opposite states would result in all subpixels being black.

The encryption process for generating shares advances along the plaintext one bit at a time. Share generation in the implementation uses a PRNG to pick a subpixel state from the first array. This state is added into the string for the first share. Then the current plaintext bit is checked. If the current bit is a 1, the opposite subpixel state is assigned to share two. If the current bit is a 0, the same subpixel state is assigned to share two. The share generation continues in this manner until subpixels have been generated for every bit of the plaintext.

The decryption process in the implementation revolves around the recombining of the shares. The algorithm advances along the shares four bits at a time, and checks if the current four bits of the shares are equal or opposite. If they are equal, a 0 bit is added to the decrypted plaintext. If they are opposite, a 1 bit is added. Once all subpixel states have been checked, the decrypted plaintext is returned.

The complete source code for the VC algorithm implementation of binary encryption and decryption is available in Appendix B.

#### 4.4.2 Efficiency

The efficiency of the VC implementation was measured using the standards set out in Chapter 3. The time taken in milliseconds to encrypt and decrypt the data, the time taken to setup the implementation, the memory occupied by the JVM in setup, encryption and decryption were all measured and recorded.

The time taken for encryption and decryption in the VC implementation was measured for multiple data sizes of pseudo-random bit strings, and each was encrypted and decrypted over 500 iterations. The mean was then calculated. Table 4.19 gives the results of this testing.

<b>Data Size (bits)</b>	<b>Average Encryption (ms)</b>	<b>Average Decryption (ms)</b>
<b>16</b>	0.056	0.010
<b>32</b>	0.080	0.014
<b>64</b>	0.196	0.052
<b>128</b>	0.368	0.088
<b>256</b>	0.868	0.214
<b>512</b>	2.822	0.492

*Table 4.19: Mean encryption/decryption times in bit-string VC. (3d.p.)*

The memory requirements for encryption and decryption were also measured over 500 iterations of the algorithm. Table 4.20 shows the mean occupied memory in megabytes in the JVM at the end of each task.

<b>Data Size (bits)</b>	<b>Average Encryption (MB)</b>	<b>Average Decryption (MB)</b>
<b>16</b>	0.451	1.729
<b>32</b>	0.452	1.729
<b>64</b>	0.450	1.730
<b>128</b>	0.452	1.732
<b>256</b>	0.455	1.735
<b>512</b>	0.461	1.741

*Table 4.20: Mean encryption/decryption memory requirements in bit-string VC. (3d.p.)*

The time taken and memory required for the setup of the VC scheme was measured over 100 iterations of the algorithm. Because the setup of the scheme required only the assignment of two string arrays of predefined values, the mean setup time was zero milliseconds for all trials. Table 4.21 gives the results of the testing.

<b>Memory used (MB):</b>	0.456
<b>Time taken (ms):</b>	0.000

Table 4.21: Mean setup time and memory requirements in bit-string VC. (3d.p.)

#### 4.4.3 Security

The theoretical security of VC schemes rests on the methods by which shares are generated. Because the first share is assigned random subpixel states, and the second share's subpixel states are assigned based on those of the first share, the states used in the shares do not reflect the encrypted secret image unless recombined. VC schemes operate on a specific definition of security – that even an adversary with unlimited computing power, given a single share of the image, would be reduced to guessing for any given subpixel state: whether the secret image pixel was black or white. This brute force approach of guessing at possible pixel states may be feasible in small shares, where a computer can run through all possible alternatives, but quickly becomes too computationally expensive as the length of shares increases. Because the scheme rests on the shares for security, longer shares give a stronger encryption strength. The likelihood of guessing the correct state for all pixels of the secret image is the equivalent of a binomial probability, as per the Bernoulli trials, the equation for which was given in Equation 4.4.3.i. Given a secret 'image' of length 608-bits, the probability of correctly guessing for all pixels whether they are black or white is given by Equation 4.4.3.ii.

$$\text{Equation 4.4.3.i} \quad P(k \text{ successes in } n \text{ trials}) = \binom{n}{k} p^k q^{n-k}$$

$$\begin{aligned} \text{Equation 4.4.3.ii} \quad P(608) &= \binom{608}{608} \cdot (0.5)^{608} \cdot (0.5)^0 \\ P(608) &= 9.41374947 \times 10^{-184} \end{aligned}$$

A brute force attack on a VC implementation requires prior knowledge of the setup. The attacker must know the pixel expansion of the scheme, in order to identify the subpixel arrays, so that the overall size or length of the secret image or plaintext may be determined. A brute force attack on the scheme would then involve trying all possible plaintexts or images of that length/size. The number of operations required for this task is as follows in Equation 4.4.3.iii.

$$\text{Equation 4.4.3.iii} \quad \text{operations} = 2^{l/s}$$

In Equation 4.4.3.i,  $l$  is the length of the share, and  $s$  is the pixel expansion of the scheme. The number of operations required for a brute force attack is, on average, those required to try half of all possibilities. The average number of operations for the attacker to recover plaintext from the VC scheme is described in Equation 4.4.3.iv.

$$\text{Equation 4.4.3.iv} \quad \frac{1}{2}(2^{l/s})$$

For a share length of 4096 bits, or a secret ‘image’ string of 1024 bits, the average number of attempts to successfully brute force attack the scheme is given in Equation 4.4.3.v.

$$\begin{aligned} \text{Equation 4.4.3.v} \quad A_{ttempts} &\approx \frac{1}{2}(2^{4096/4}) \\ A_{ttempts} &\approx \frac{1}{2}(2^{1024}) \\ A_{ttempts} &\approx \frac{1}{2}(1.79769313 \times 10^{308}) \\ A_{ttempts} &\approx 8.98846567 \times 10^{307} \end{aligned}$$

VC schemes offer non-singular mapping of plaintext to ciphertext, because the first share is assigned random subpixel states, and the second share is assigned based on the first share. Each time the same piece of plaintext is encrypted, it is likely to result in two new shares. This increases its security against outside attacks. However, as the scheme does not have separate ciphertexts and keys, and the shares are both key and ciphertext, a chosen plaintext attack would allow for the unknown share to be generated by an attacker, based on the share that was communicated. This issue is known as cheating, and involves an adversary who is an authorized participant in the scheme. In this manner, the implemented VC scheme has a theoretical weakness to known and chosen plaintext attacks.

The practical security of the VC scheme was measured by the avalanche effect of the algorithm. The percentage of individual bits that were altered based on the alteration of a single bit of the pseudo-random plaintext bit string were measured over all the data sizes, during the course of 500 iterations. Table 4.22 gives the results of this testing.

<b>Data Size</b>	<b>% of Bits Changed</b>
<b>16</b>	49.275
<b>32</b>	50.169
<b>64</b>	50.005
<b>128</b>	49.934
<b>256</b>	49.981
<b>512</b>	50.072

*Table 4.22: Avalanche effect in bit-string VC. (3d.p.)*

## **4.5 RC4**

This section discusses the results of the tests conducted on the RC4 algorithm. Section 4.5.1 gives an in-depth review of the implementation used in the research. Section 4.5.2 then explores the results of the testing for efficiency, and finally, section 4.5.3 details the security results.

### **4.5.1 Implementation Details**

The implementation used in the research conducted into RC4 was based around Java's in-built cryptographic functions. The JSE 7 Crypto and security libraries were utilized, along with the Cipher function. The program used the SecureRandom function to generate an initialization vector, which was then implemented with the KeyGenerator to produce a 128-bit key. The Cipher object was set to RC4 mode, and encryption and decryption occurred on arrays of bytes. The encryption function took a single plaintext string as input, and then returned the encrypted byte array as the ciphertext output. The decryption function took the ciphertext byte array and returned the decrypted byte array, which was then converted back into the original plaintext string using UTF-8 encoding.

The testing programs for frequency analysis and avalanche effect were the same as those used in the evaluation of the AES algorithm, as both schemes produce ciphertext byte arrays. The source code for the RC4 implementations can be found in Appendix B.

### **4.5.2 Efficiency**

The efficiency of the RC4 implementation was measured by the time taken to encrypt and decrypt the data, the average memory required at each stage, and the memory and time taken during the algorithm set up. The time taken was measured in milliseconds using the inbuilt CurrentTimeMillis function, which gives the current system time, while the memory was measured using the current memory occupied by the JVM at each stage during runtime.

The result of the testing for the time taken to encrypt and decrypt the different data sizes is shown in Table 4.23. The times were measured over 1000 iterations on each data size. The encryption and decryption times do not appear to scale linearly with the size of the data.



<b>Data Size</b>	<b>Average Encryption (ms)</b>	<b>Average Decryption (ms)</b>
<b>304</b>	0.014	0.022
<b>928</b>	0.027	0.025
<b>3024</b>	0.023	0.016
<b>4408</b>	0.020	0.045
<b>8144</b>	0.024	0.032

*Table 4.23: Encryption and decryption times in RC4 (3d.p.)*

The amount of memory and the time taken to set up the implementation were also measured over the course of 100 iterations. The results are shown in Table 4.24. The time taken was measured in milliseconds, while the memory requirement was evaluated in megabytes.

<b>Memory used (MB):</b>	2.340
<b>Time taken (ms):</b>	258.500

*Table 4.24: Set up requirements for RC4 (3d.p.)*

The memory requirements for encrypting and decrypting the different data sizes were measured over the course of 500 iterations on each data size. The results of this testing are shown in Table 4.25.

<b>Data Size</b>	<b>Average Encryption (MB)</b>	<b>Average Decryption (MB)</b>
<b>304</b>	1.362	1.361
<b>928</b>	1.363	1.362
<b>3024</b>	1.364	1.366
<b>4408</b>	1.364	1.364
<b>8144</b>	1.366	1.369

*Table 4.25: Memory requirements for RC4 (3d.p.)*

### **4.5.3 Security**

The security of the RC4 algorithm was measured through theoretical and practical analysis. The avalanche effect, frequency analysis of the output, vulnerability to known and chosen plaintext, and the size of the key space were examined to provide an evaluation of the scheme's comparative security.

The number of operations required on average to brute force attack the key space of 128-bit RC4 is equivalent to that of 128-bit AES, given in Equation 4.5.3.i. On average, half of all possible 128-bit keys must be attempted before the system is compromised.

$$\begin{aligned}
 \text{Equation 4.5.3.i} \quad \text{operations} &\approx \frac{1}{2} 2^{128} \\
 \text{operations} &\approx \frac{1}{2} (3.4028237 \times 10^{38}) \\
 \text{operations} &\approx 1.7014118 \times 10^{38}
 \end{aligned}$$

The result of Equation 4.5.3.i is considered to be impracticably large, and as such RC4 is considered secure against brute force attacks in this manner.

RC4 was also tested for vulnerability to frequency analysis, and gave similar distributions to that of 128-bit AES. Table 4.26 shows the frequency distribution of the ciphertext outputs of a single 8144-bit plaintext string. Each encryption and decryption of the string used a different, randomly generated key, as RC4 provides a singular mapping – the same plaintext input with the same key will always provide the same ciphertext output. The tests were conducted over 1000 iterations, and then the mean calculated.

<b>Frequency</b>	<b>Average # of Bytes</b>	<b>Times Occurred</b>
<b>1</b>	18.875	1000
<b>2</b>	37.987	1000
<b>3</b>	50.649	1000
<b>4</b>	50.299	1000
<b>5</b>	38.588	1000
<b>6</b>	26.575	1000
<b>7</b>	14.693	1000
<b>8</b>	7.395	1000
<b>9</b>	3.207	995
<b>10</b>	1.501	872
<b>11</b>	0.889	488
<b>12</b>	0.792	212
<b>13</b>	0.736	72
<b>14</b>	0.682	22
<b>15</b>	0.429	7
<b>16</b>	1	4

*Table 4.26: Frequency analysis of an RC4 encrypted 8144-bit string (3d.p.)*

The vulnerability of RC4 to chosen plaintext attacks was examined using a single 4048-bit chosen plaintext string consisting of a single repeated character. Table 4.27 shows the output of frequency analysis on the resulting ciphertext.

<b>Frequency</b>	<b>Average # of Bytes</b>	<b>Times Occurred</b>
<b>1</b>	70.164	1000
<b>2</b>	69.383	1000
<b>3</b>	45.674	1000
<b>4</b>	22.212	1000
<b>5</b>	8.968	1000
<b>6</b>	2.939	989
<b>7</b>	1.242	683
<b>8</b>	0.889	216
<b>9</b>	0.917	48
<b>10</b>	1	5

*Table 4.27: Frequency analysis of ciphertext from a 4048-bit chosen plaintext string (3d.p.)*

The results in Table 4.27 can then be contrasted with the frequency analysis of ciphertext from a 4408-bit plaintext string containing typical English language frequency information. The frequency analysis of the ciphertext output from this 4408-bit string is given in Table 4.28.

<b>Frequency</b>	<b>Average # of Bytes</b>	<b>Times Occurred</b>
<b>1</b>	64.449	1000
<b>2</b>	68.749	1000
<b>3</b>	49.360	1000
<b>4</b>	26.736	1000
<b>5</b>	11.407	1000
<b>6</b>	4.002	997
<b>7</b>	1.436	848
<b>8</b>	0.948	362
<b>9</b>	0.855	83
<b>10</b>	0.714	14
<b>11</b>	0.750	4
<b>12</b>	1	1

*Table 4.28: Frequency analysis of a 4408-bit string encrypted with RC4. (3d.p.)*

The avalanche effect of the algorithm was measured by changing a single bit of the plaintext data and comparing the change in outputted ciphertext in both the total number of bytes altered and the number of positions altered. This was repeated over 500

iterations for each data size and the means calculated. Table 4.29 gives the results of this testing.

Data Size	1-bit Altered from Previous	
	Same Bytes	Same Position
<b>304</b>	97.668%	97.368%
<b>928</b>	99.472%	99.145%
<b>3024</b>	99.940%	99.735%
<b>4408</b>	99.979%	99.819%
<b>8144</b>	99.997%	99.902%

*Table 4.29: Avalanche effect in RC4 (3d.p.)*

## 4.6 COMPARATIVE RESULTS

This section gives the comparative results of the different implementations of the tested schemes. Section 4.6.1 compares the relative efficiency and security of the high-level software implementations for the 2-out-of-2 VC scheme and the 4-bit CME scheme. This is followed in section 4.6.2 by the comparison of efficiency and security in the low level software implementations of AES and 8-bit byte-oriented CME. Finally, 4.6.3 discusses the comparative results from the byte-oriented CME and the low-level ECC implementation.

### 4.6.1 2-out-of-2 VC versus 4-bit CME

The comparison of the 2-out-of-2 VC scheme and the high level software CME 4-bit scheme was based on the encryption of binary strings. Each scheme was tested on the same pseudo random bit strings of lengths 16, 32, 64, 128, 256 and 512. Each string was encrypted and decrypted 500 times, and the average encryption and decryption times, as well as the resulting ciphertext length were averaged for each bit length.

The measurements for efficiency, using encryption time, decryption time and overall time for set up, were gauged in milliseconds. For encryption and decryption time, the CME scheme out-performed that of the 2-out-of-2 VC scheme, as shown in Table 4.30. As the VC scheme did not require the generation of keys, and the only setup operation it utilized was to assign the string arrays of possible subpixel values, its overall time for setup was significantly better than that of the CME scheme, with the VC implementation taking 0ms where CME took 21.44ms on average. The details of the setup

tests are given in Table 4.31. This difference in setup time was expected, as the CME 4-bit scheme setup involves the generation of a random 16 by 16 key matrix, as well as the generation of all possible bit strings of length 4. The difference in encryption and decryption time was unexpected, as the VC scheme involved fewer operations overall, and used only one pseudorandom generator. However, the CME scheme appeared to give a better performance in encrypting and decrypting data over all strings, and this difference became more pronounced when testing longer bit strings.

<b>Bit String Length</b>	<b>Encryption (ms)</b>		<b>Decryption (ms)</b>	
	<b>VC</b>	<b>4-bit CME</b>	<b>VC</b>	<b>4-bit CME</b>
<b>16</b>	0.056	0.020	0.010	0.020
<b>32</b>	0.080	0.066	0.014	0.036
<b>64</b>	0.196	0.104	0.052	0.060
<b>128</b>	0.368	0.214	0.088	0.074
<b>256</b>	0.868	0.369	0.214	0.136
<b>512</b>	2.822	1.130	0.492	0.328

*Table 4.30: Mean encryption and decryption times for differing bit string lengths in the VC and CME schemes. (3d.p.)*

	<b>4-bit CME</b>	<b>VC</b>
<b>Memory used (MB):</b>	0.448	0.456
<b>Time taken (ms):</b>	21.440	0.000

*Table 4.31: Mean setup for the VC and CME schemes. (3d.p.)*

The practical security of both algorithms was tested by the overall avalanche effect of each when a single bit of the pseudo-random plaintext data was flipped. Both schemes hover at the 50% mark for the total amount of bits that are unaltered from the previous ciphertext. Table 4.32 gives the results for the two schemes side by side for each of the six tested plaintext sizes.

<b>Data Size</b>	<b>% of Bits Unchanged</b>	
	<b>VC</b>	<b>CME 4-bit</b>
<b>16</b>	49.275	50.319
<b>32</b>	50.169	50.619
<b>64</b>	50.005	50.499
<b>128</b>	49.934	50.286
<b>256</b>	49.981	50.337

<b>512</b>	50.072	50.120
------------	--------	--------

Table 4.32: Avalanche effect for differing bit string lengths in the VC and CME schemes. (3d.p.)

The requirements for brute force attacks on each algorithm were also calculated. The number of average operations required to brute force a VC scheme is given in Equation 4.6.1.i, where  $l$  is the share length and  $s$  is the pixel expansion of the scheme.

$$\text{Equation 4.6.1.i} \quad \frac{1}{2}(2^{l/s})$$

In the utilized implementation, the pixel expansion is equal to four, so  $s$  can be replaced by 4 in Equation 4.6.1.i. The overall length of the shares then determines the level of security, as the shares within the VC scheme are both ciphertext and key at once. So a brute force attack on the implementation attempts all possible plaintexts. In contrast, the operations required for the brute force of a 4-bit CME scheme are shown in Equation 4.6.1.ii.

$$\text{Equation 4.6.1.ii} \quad M_{\text{matrix attempts}} = \frac{1}{2}(17^{256})$$

The results of Equation 4.5.1.ii are too large to practicably compute for most calculators. To give a result, the MatLab variable point integer toolbox was utilized. The obtained results are shown in Equation 4.6.1.iii.

$$\begin{aligned} \text{Equation 4.6.1.iii} \quad M_{\text{matrix attempts}} &= \frac{1}{2}(9.883798 \times 10^{314}) \\ M_{\text{matrix attempts}} &= 4.941899 \times 10^{314} \end{aligned}$$

Comparatively, even a VC scheme that consisted of shares of 4096-bits generated from a 1024-bit secret ‘image’ plaintext would not match this level, as based on Equation 4.6.1.i the operations required for a brute force attack on a 4096-bit are shown in Equation 4.6.1.iv.

$$\begin{aligned} \text{Equation 4.6.1.iv} \quad A_{\text{attempts}} &= \frac{1}{2}(2^{4096/4}) \\ A_{\text{attempts}} &= \frac{1}{2}(2^{1024}) \\ A_{\text{attempts}} &= \frac{1}{2}(1.79769313 \times 10^{308}) \\ A_{\text{attempts}} &= 8.98846567 \times 10^{307} \end{aligned}$$

As the result of Equation 4.6.1.iv is seven orders of magnitude smaller than that of Equation 4.6.1.iii, it can be determined that the 4-bit CME scheme is quantifiably more resistant to brute force attack than the VC scheme for shares of up to 4096 bits.

The likelihood of an adversary successfully guessing the secret ‘image’ of the VC scheme for a 608-bit secret image is approximately equal to probability of an adversary successfully guessing for 608-bit coordinate ciphertext whether each coordinate is empty

or full. This probability is given in Equation 4.6.1.v, as per the Bernoulli trials formula.

$$\begin{aligned} \text{Equation 4.6.1.v} \quad P(608) &= \binom{608}{608} \cdot (0.5)^{608} \cdot (0.5)^0 \\ P(608) &= 9.41374947 \times 10^{-184} \end{aligned}$$

#### 4.6.2 AES versus 8-bit CME Byte Scheme

The implementations of the byte-level CME scheme and Java’s inbuilt AES were compared as per the schema outlined in Chapter 3. The implementation efficiency was measured by the time taken in milliseconds at each stage, and the total memory required to complete each stage in megabytes. The security was then measured based on theoretical resistance to brute force attacks, chosen plaintext attacks, the results of practical frequency analysis, and the overall avalanche effect of the scheme. Each category was tested on each of the five different plaintext data strings over many iterations of the algorithm. The UTF-8 encoded plaintext for testing was taken from *Hamlet* (Shakespeare & Ackroyd, 2006) and *Pride and Prejudice* (Austen, 2006).

The time and memory requirements for the setup of the scheme were measured over the course of 100 iterations of the algorithm. The memory in megabytes occupied by the JVM at the end of the setup stage and the total time taken to complete the setup in milliseconds were recorded over the tests. Table 4.33 gives the results of the setup efficiency testing.

	<b>AES</b>	<b>CME</b>
<b>Time taken (ms):</b>	409.000	80.130
<b>Memory used (MB):</b>	2.364	1.217

*Table 4.33: Mean setup requirements for the AES and byte-level CME schemes. (3d.p.)*

The encryption and decryption time for each of the implementations were measured over the course of 1000 iterations of the algorithm on each of the five strings. Table 4.34 gives the results of mean encryption and decryption time over the testing for each different tested plaintext. The time requirements for the byte-level CME increase linearly with the size of the plaintext.

Data Size	Encryption (ms)		Decryption (ms)	
	AES	Byte CME	AES	Byte CME
<b>304</b>	0.199	0.031	0.170	0.012
<b>928</b>	0.142	0.059	0.182	0.023
<b>3024</b>	0.173	0.136	0.179	0.065
<b>4408</b>	0.185	0.173	0.196	0.050
<b>8144</b>	0.148	0.262	0.250	0.093

Table 4.34: Mean encryption/decryption time for the AES and byte-level CME schemes. (3d.p.)

The memory requirements at each stage for encryption and decryption were measured by the amount of memory in use by the JVM at the end of the stage. These values were recorded over 500 iterations of the algorithm for each of the plaintext strings. Table 4.35 gives the results of this testing.

Data Size	Encryption (MB)		Decryption (MB)	
	AES	Byte CME	AES	Byte CME
<b>304</b>	1.393	1.244	1.395	1.244
<b>928</b>	1.397	1.245	1.400	1.246
<b>3024</b>	1.396	1.251	1.416	1.251
<b>4408</b>	1.399	1.255	1.424	1.255
<b>8144</b>	1.395	1.263	1.444	1.264

Table 4.35: Mean encryption/decryption memory for the AES and byte-level CME schemes. (3d.p.)

The security for the two implementations was measured first by theoretical resistance to brute force attacks. 128-bit AES relies on the strength of its key to prevent brute force attacks. Equation 4.6.2.i gives the mean number of attempts an adversary would be required to make on average to correctly identify the key.

$$\begin{aligned}
 \text{Equation 4.6.2.i} \quad \text{operations} &\approx \frac{1}{2} 2^{128} \\
 \text{operations} &\approx \frac{1}{2} (3.4028237 \times 10^{38}) \\
 \text{operations} &\approx 1.7014118 \times 10^{38}
 \end{aligned}$$

This result can be compared with that given in Equation 4.6.2.ii, the average number of attempted key matrices required to brute force the 8-bit byte-level implementation of CME.

$$\text{Equation 4.6.2.ii} \quad M_{\text{matrix attempts}} \approx \frac{1}{2} (257^{65536})$$



$$M_{matrix\ attempts} \approx \frac{1}{2}(2.3832557 \times 10^{157937})$$

$$M_{matrix\ attempts} \approx 1.19162785 \times 10^{157937}$$

The result of Equation 4.6.2.ii is 157,899 orders of magnitude larger than that of Equation 4.6.2.i, suggesting that the CME implementation gives a higher level of resistance against brute force attacks.

The vulnerability of each algorithm to methods of frequency analysis and chosen plaintext was explored through a custom-built program that analysed the frequency of the ciphertext bytes over many iterations of the algorithms. Over the course of 1000 iterations, an 8814-bit string of plaintext data was encrypted into 1000 ciphertexts, and the frequency with which each byte occurred was measured. For the CME implementation, the bytes were measured in their coordinate tuples, and the percentage of those coordinates which were full or empty was calculated. Table 4.36 gives the results from the analysis of 128-bit AES, while Table 4.37 gives the results from the byte-level CME.

<b>Frequency</b>	<b>Average # of Bytes</b>	<b>Times Occurred</b>
<b>1</b>	18.626	1000
<b>2</b>	37.606	1000
<b>3</b>	50.462	1000
<b>4</b>	50.010	1000
<b>5</b>	39.938	1000
<b>6</b>	26.581	1000
<b>7</b>	15.194	1000
<b>8</b>	7.554	1000
<b>9</b>	3.403	978
<b>10</b>	1.780	749
<b>11</b>	1.271	388
<b>12</b>	1.068	146
<b>13</b>	1.019	54
<b>14</b>	1	18
<b>15</b>	1	4
<b>16</b>	1	1

*Table 4.36: Frequency analysis for 128-bit AES on ciphertext from an 8814-bit string. (3d.p.)*

<b>Frequency</b>	<b>Blank</b>	<b>Full</b>	<b>Times Occurred</b>
<b>1</b>	50.047%	49.953%	1000
<b>2</b>	48.589%	51.411%	1000
<b>3</b>	46.491%	53.509%	308
<b>4</b>	40.000%	60.000%	5

*Table 4.37: Frequency analysis for byte-level CME on ciphertext from an 8814-bit string. (3d.p.)*

The security of both schemes was further evaluated by the analysis of the frequencies of ciphertext resulting from the encryption of a string consisting of a single repeated character. Table 4.38 gives the results for 128-bit AES, while Table 4.39 gives the results for the byte-level CME.

<b>Frequency</b>	<b>Average # of Bytes</b>	<b>Times Occurred</b>
<b>1</b>	69.115	1000
<b>2</b>	69.620	1000
<b>3</b>	46.234	1000
<b>4</b>	23.054	1000
<b>5</b>	9.207	1000
<b>6</b>	3.070	963
<b>7</b>	1.491	561
<b>8</b>	1.098	184
<b>9</b>	1.026	38
<b>10</b>	1	13

*Table 4.38: Frequency analysis for 128-bit AES on ciphertext from a 4048-bit chosen plaintext string. (3d.p.)*

<b>Frequency</b>	<b>Blank</b>	<b>Full</b>	<b>Times Occurred</b>
<b>1</b>	50.105%	49.895%	1000
<b>2</b>	44.127%	55.873%	1000
<b>3</b>	31.915%	68.085%	46

*Table 4.39: Frequency analysis for byte-level CME on ciphertext from a 4048-bit chosen plaintext string. (3d.p.)*

The final security measure was the overall avalanche effect of the algorithm. This was tested on the total percentage of bytes that occurred in the previous and current ciphertexts, changed and unchanged, as well as the total percentage of unchanged bytes

occurring in the same position. Each tested plaintext differed from the previous plaintext by exactly one bit, and the algorithms were tested over 500 iterations of encryption and decryption. Table 4.40 gives the comparative results for each of the different data strings.

Data Size	Same Bytes (%)		Same Position (%)	
	AES	CME	AES	CME
<b>304</b>	37.767	44.839	24.779	0.414
<b>928</b>	62.777	84.026	38.905	0.388
<b>3024</b>	87.935	99.713	45.857	0.422
<b>4408</b>	94.276	99.984	48.227	0.404
<b>8144</b>	99.100	100	48.593	0.395

Table 4.40: Avalanche effect in 128-bit AES and byte-level CME schemes. (3d.p.)

#### 4.6.3 ECC versus 8-bit CME Byte Scheme

The comparison of ECC and the byte-level CME scheme was performed according to the schema laid out in Chapter 3. The efficiency of the algorithms was calculated based on average setup time and the memory required by the JVM to execute the setup. The evaluation of security in each algorithm was based on the theoretical resistance of the schemes to brute force attacks.

The efficiency in setup of ECDH and byte-level CME was measured over 100 iterations of the algorithm, based on the occupied memory in megabytes and the time taken in milliseconds. Table 4.41 gives the comparative results, which have been averaged over all trials.

	ECC	CME
<b>Time taken (ms):</b>	359.500	80.130
<b>Memory used (MB):</b>	1.192	1.217

Table 4.41: Setup requirements for byte-level CME and ECDH protocols. (3d.p.)

The security of CME and ECDH can be compared based on the resistance to brute force attacks. Equation 4.6.3.i gives the average number of key attempts required for a brute force attack on byte-level CME, while Equation 4.6.3.ii gives the average number of operations key attempts required for the symmetric key resulting from the ECDH protocol. Equation 4.6.3.iii gives the average number of operations to brute force attack the 192-bit ECC generated private key.

$$\begin{aligned}
\text{Equation 4.6.3.i} \quad M_{\text{matrix attempts}} &\approx \frac{1}{2}(257^{65536}) \\
M_{\text{matrix attempts}} &\approx \frac{1}{2}(2.3832557 \times 10^{157937}) \\
M_{\text{matrix attempts}} &\approx 1.19162785 \times 10^{157937} \\
\text{Equation 4.6.3.ii} \quad \text{operations} &\approx \frac{1}{2}2^{128} \\
\text{operations} &\approx \frac{1}{2}(3.4028237 \times 10^{38}) \\
\text{operations} &\approx 1.7014118 \times 10^{38} \\
\text{Equation 4.6.3.iii} \quad BF_{\text{operations}} &\approx \frac{1}{2}(2^{192}) \\
BF_{\text{operations}} &\approx \frac{1}{2}(6.2771017 \times 10^{57}) \\
BF_{\text{operations}} &\approx 3.1385509 \times 10^{57}
\end{aligned}$$

Based on the results in Equations 4.6.3.1.i through iii, the byte-level CME scheme may be suggested to be quantifiably more resistant to brute force attacks than the 192-bit ECDH scheme.

#### 4.6.4 RC4 versus 8-bit CME

The comparison between RC4 and 8-bit CME was completed using the Java implementations described in sections 4.5.1 and 4.1.2 respectively. The comparison was completed using the time and memory requirements for encryption, decryption and setup; the avalanche effect of the algorithm; frequency analysis of the output; resistance to chosen and known plaintext; and the number of operations required on average to complete a brute force attack on the key space. Complete source code for both algorithms can be found in Appendix B. The testing data used for the study was taken from *Pride & Prejudice* (Austen, 2006) and *Hamlet* (Ackroyd & Shakespeare, 2006). The testing data is available in Appendix C.

The set up requirements of the algorithms were tested over the course of 100 iterations, using both the time elapsed and the memory occupied by the JVM. These results were then tabulated and the means calculated. Table 4.42 gives the results.

	<b>RC4</b>	<b>CME</b>
<b>Time taken (ms):</b>	258.500	80.130
<b>Memory used (MB):</b>	2.340	1.217

Table 4.42: Comparative set up requirements for RC4 and 8-bit CME (3d.p.)

The time taken to encrypt and decrypt the different data sizes for each algorithm was

recorded over the course of 1000 iterations. The mean encryption and decryption time in milliseconds was then calculated in milliseconds for each data size and each algorithm. The RC4 implementation was significantly faster than the CME scheme in encryption over all data sizes, and in decryption over all data of 3024-bits and above. Table 4.43 gives the results of the time requirement evaluations.

Data Size	Encryption (ms)		Decryption (ms)	
	RC4	Byte CME	RC4	Byte CME
<b>304</b>	0.014	0.031	0.022	0.012
<b>928</b>	0.027	0.059	0.025	0.023
<b>3024</b>	0.023	0.136	0.016	0.065
<b>4408</b>	0.020	0.173	0.045	0.050
<b>8144</b>	0.024	0.262	0.032	0.093

*Table 4.43: RC4 versus 8-bit CME encryption and decryption time requirements (3d.p.)*

The final efficiency test for the two algorithms was the comparison of memory requirements during encryption and decryption. Each data size was encrypted and decrypted over 500 iterations, and the mean for each data size was calculated for each algorithm. Table 4.44 gives the results of this testing. Over all data sizes, the memory required for execution of the CME algorithm was lower than that of RC4.

Data Size	Encryption (MB)		Decryption (MB)	
	RC4	Byte CME	RC4	Byte CME
<b>304</b>	1.362	1.244	1.361	1.244
<b>928</b>	1.363	1.245	1.362	1.246
<b>3024</b>	1.364	1.251	1.366	1.251
<b>4408</b>	1.364	1.255	1.364	1.255
<b>8144</b>	1.366	1.263	1.369	1.264

*Table 4.44: RC4 versus CME memory requirements (3d.p.)*

The theoretical security of the two algorithms against brute force attack was measured by the number of operations that would be required on average to successfully recover the key. The mean number of operations to recover the 128-bit RC4 key is given in Equation 4.6.4.i. The mean number of operations required to recover the key matrix from the 8-bit CME scheme is given in Equation 4.6.4.ii.

$$\begin{aligned} \text{Equation 4.6.4.i} \quad \text{operations} &\approx \frac{1}{2} 2^{128} \\ \text{operations} &\approx \frac{1}{2} (3.4028237 \times 10^{38}) \end{aligned}$$

$$\text{operations} \approx 1.7014118 \times 10^{38}$$

Equation 4.6.4.ii

$$M_{\text{matrix attempts}} \approx \frac{1}{2}(257^{65536})$$

$$M_{\text{matrix attempts}} \approx \frac{1}{2}(2.3832557 \times 10^{157937})$$

$$M_{\text{matrix attempts}} \approx 1.19162785 \times 10^{157937}$$

The result of Equation 4.6.4.ii is 157,899 orders of magnitude greater than that of Equation 4.6.4.i. As such, it can be posited that the 8-bit CME scheme offers greater resistance to brute force attacks than the 128-bit RC4.

The security of the two algorithms was measured using frequency analysis the ciphertext of strings of plaintext that reflected English language frequencies and a plaintext string of a single repeated character. The frequency analysis of the English language plaintext strings gave an indication of what, if any, frequency data from the original plaintext might be revealed by analysis of the ciphertext. Over 1000 iterations, a 8814-bit string of English language plaintext was encrypted into 1000 different ciphertexts – using a different randomly generated key for each iteration in the RC4 algorithm – and the frequency distribution of the bytes in each ciphertext was calculated. The mean distribution was then tabulated. Table 4.45 gives the results of the frequency analysis on RC4, while Table 4.46 gives the results of the analysis on 8-bit CME. The distribution of frequencies in the 8-bit CME scheme is significantly flatter than those of the RC4 algorithm.

<b>Frequency</b>	<b>Average # of Bytes</b>	<b>Times Occurred</b>
<b>1</b>	18.875	1000
<b>2</b>	37.987	1000
<b>3</b>	50.649	1000
<b>4</b>	50.299	1000
<b>5</b>	38.588	1000
<b>6</b>	26.575	1000
<b>7</b>	14.693	1000
<b>8</b>	7.395	1000
<b>9</b>	3.207	995
<b>10</b>	1.501	872
<b>11</b>	0.889	488

<b>12</b>	0.792	212
<b>13</b>	0.736	72
<b>14</b>	0.682	22
<b>15</b>	0.429	7
<b>16</b>	1	4

*Table 4.45: Frequency analysis of ciphertext from an 8814-bit string in RC4 (3d.p.)*

<b>Frequency</b>	<b>Blank</b>	<b>Full</b>	<b>Times Occurred</b>
<b>1</b>	50.047%	49.953%	1000
<b>2</b>	48.589%	51.411%	1000
<b>3</b>	46.491%	53.509%	308
<b>4</b>	40%	60%	5

*Table 4.46: Frequency analysis of ciphertext from an 8814-bit string in 8-bit CME. (3d.p.)*

The resistance of the algorithms to chosen plaintext attacks was tested using a 4048-bit plaintext string consisting of a single repeated ‘a’ character. Frequency analysis was performed on the resulting ciphertext of this string over the course of 1000 iterations – using a different key for each iteration of the RC4 algorithm – and the resulting mean distribution was calculated. Table 4.47 shows the frequency analysis of RC4, while Table 4.48 shows the frequency analysis of 8-bit CME.

<b>Frequency</b>	<b>Average # of Bytes</b>	<b>Times Occurred</b>
<b>1</b>	70.164	1000
<b>2</b>	69.383	1000
<b>3</b>	45.674	1000
<b>4</b>	22.212	1000
<b>5</b>	8.968	1000
<b>6</b>	2.939	989
<b>7</b>	1.242	683
<b>8</b>	0.889	216
<b>9</b>	0.917	48
<b>10</b>	1	5

*Table 4.47: Frequency analysis of ciphertext from a 4048-bit chosen plaintext in RC4 (3d.p.)*

<b>Frequency</b>	<b>Blank</b>	<b>Full</b>	<b>Times Occurred</b>
<b>1</b>	50.105%	49.895%	1000
<b>2</b>	44.127%	55.873%	1000

<b>3</b>	31.915%	68.085%	46
----------	---------	---------	----

*Table 4.48: Frequency analysis of ciphertext from a 4048-bit chosen plaintext in 8-bit CME (3d.p.)*

The final measure of security in the algorithms was the avalanche effect they produced. Each plaintext was changed by exactly 1 bit for each new iteration, and the avalanche effect was measured over the course of 500 iterations by the total percentage of bytes that had remained unchanged from the previous plaintext, and the total number of bytes that remained in the same position in both ciphertexts. Table 4.49 gives the results of this avalanche testing. The avalanche effect of the CME algorithm was drastically higher than that of the RC4 algorithm.

<b>Data Size</b>	<b>Same Bytes (%)</b>		<b>Same Position (%)</b>	
	<b>RC4</b>	<b>CME</b>	<b>RC4</b>	<b>CME</b>
<b>304</b>	97.668	44.839	97.368	0.414
<b>928</b>	99.472	84.026	99.145	0.388
<b>3024</b>	99.940	99.713	99.735	0.422
<b>4408</b>	99.979	99.984	99.819	0.404
<b>8144</b>	99.997	100	99.902	0.395

*Table 4.49: Comparative avalanche effect in RC4 and 8-bit CME (3d.p.)*

## **4.7 CONCLUSION**

In this chapter, the test results of the study specified in Chapter 3 were enumerated for each individual algorithm, and the comparative results for each of the tested schemes were detailed. The data collected over many iterations of each of the schemes was averaged and the mean results of the testing were presented. Measures of efficiency and security were given across each of the algorithms, and used to compare the performance of each.

In Chapter 5, the results will be discussed and evaluated, and conclusions about the original hypotheses and research questions will be drawn.



# **Chapter 5**

## **Discussion and Analysis of Findings**

### **5.0 INTRODUCTION**

In Chapter 4, the results of the study were enumerated and explained, and the four algorithms evaluated based on their results in each of the testing criteria. The data was presented, and the sets of algorithms were analysed in pairs for a fairer comparison. The results provided a clearly detailed, achievable level of security in the graphic-based CME system.

In this chapter, the results are explored in depth. Section 1 uses the results of the study to answer the research questions posed in Chapter 3, and to revisit the hypotheses with the information gained from testing. Section 2 then discusses the implications of the results, and their application to real-world problems. Finally, section 3 gives the conclusions, which suggest that there are real world applications and security benefits to graphic-based ciphers.

### **5.1 RESEARCH QUESTIONS AND HYPOTHESES**

In this section, the research questions and hypotheses posed in Chapter 3 are answered with the data from the tests. Section 5.1.1 reiterates the research questions and discusses the answers gained from the study. Section 5.1.2 then explores the hypotheses using the new data available from the testing.

#### **5.1.1 Research Question 1: What are the security benefits of graphic based systems in comparison to classical block ciphers?**

The alternative key structures in graphic based systems can provide a comparatively higher level of security in similarly sized implementations. The proposed Coordinate Matrix Encryption (CME) scheme offered a high level of security over all tests. In comparison with the 128-bit AES implementation, the resistance of the CME scheme to brute force was 157,899 orders of magnitude higher in the number of key attempts required on average as shown in Equations 4.6.2.i & ii, giving it a far higher theoretical resistance to brute force attacks. The avalanche effect of the CME scheme also outpaced

that of the implemented AES, with the CME scheme resulting in less than 1% of the bytes in the ciphertext remaining in the same position after a 1-bit variation in the plaintext (Table 4.40). In comparison, AES resulted in approximately 25-49% of the ciphertext bytes remaining in the same position after a single bit change in the plaintext. Frequency distributions over the ciphertext for both algorithms resulted in little to no information about the plaintext being communicated (Tables 4.36 to 4.39). The statistical properties of the plaintext in both algorithms are diffused by the addition of a pseudo-random key, assisting in creating confusion in the ciphertext. The highest frequency occurrence of the same byte in the ciphertext resulting from the encryption of a 8814-bit plaintext string in AES 128 was 16 – a single byte occurring in the ciphertext 16 times (Table 4.36). In comparison, the same string encrypted with the 8-bit CME scheme resulted in a highest frequency occurrence of three (Table 4.37).

In comparison to the stream cipher RC4, CME gave again a quantifiably higher level of security, with a brute force attack on the 8-bit CME being 157,899 orders of magnitude more expensive than on the 128-bit RC4 as shown in Equations 4.6.4.i & ii. The avalanche effect of the CME cipher was also drastically higher than that of RC4, which produced very little variation from one ciphertext to the next (Table 4.49). As the frequency distribution of RC4 was closely aligned with that of AES, the distribution of the 8-bit CME was flatter than RC4, with much lower occurrences of high frequencies. The highest frequency occurring in the ciphertext of an 8814-bit string in CME was three, while RC4 had a top frequency of 16 (Table 4.45).

These results suggest a higher level of security is given by the graphic-based alternate cipher, than the Feistel-based AES or the traditional stream cipher RC4. The alternative structure of the key, and the use of bit string codewords as an alphabet provides for a much higher number of possible keys than are achievable by a simple binary key string, which has only two possible options for any one bit of the key.

### **5.1.2 Research Question 2: What difficulties are faced in the implementation of graphic based systems?**

The implementation of alternative graphic-based systems requires computational overheads in the creation of the key structure, because these systems rely on complex key structures for security. The creation and storage of such key structures results in a computational overhead not necessarily incurred by classical key structures such as bit keys. The CME method proposed utilizes a large 2-dimensional key matrix, and the

majority of computational overhead in the algorithm lies in this key structure. Because the security of the algorithm rests particularly on the design of the key, the computational complexity of the key is relatively high. The 4-bit scheme produces a 16 by 16 key matrix, roughly equivalent to a 256-bit pseudorandom binary key, however each individual full coordinate in the matrix contains a 4 bit codeword. As a result, the key matrix requires a certain amount of memory to be occupied to store it. This result is in keeping with those ciphers based on special graphs of large girth, such as the Cayley graphs utilized by Ustimenko (2007), which can encounter large computational overheads due to the graph size. The unusual key structure of the CME design is not quantified in the manner of binary keys, where a key size can be simply determined, as in AES, and the particulars of the structure require that the algorithm find alternative means to store it.

### 5.1.3 Sub-Questions

**Sub-question 1:** *Does the implementation of the proposed method provide better levels of security than the comparable algorithms?*

**Answer:** The implemented CME schemes gave a higher level of resistance to brute force attacks than all tested comparable algorithms. The particular structure of the key matrix resulted in many orders of magnitude more operations required to brute force the CME schemes than required for the AES, RC4, VC and ECC schemes. The frequency analysis resulted in an even distribution of full and empty coordinates across the frequencies (Table 4.37), and the binomial probability of an adversary being successfully able to guess for any given ciphertext whether each coordinate was full or empty was low enough for the likelihood to be exceedingly improbable (Equation 4.6.1.v). The avalanche effect of the algorithm was on par with that of the VC scheme, with an approximate 50% change in the ciphertext given a 1-bit change in the plaintext (Table 4.32). When compared with the AES and RC4 schemes, the CME algorithm resulted in fewer than 1% of bytes in the ciphertext remaining in the same position given a 1-bit change in the plaintext. AES resulted in approximately 25-49% of bytes remaining in the same position (Table 4.40), while the RC4 algorithm resulted in approximately 97-99.9% of bytes remaining in the same position after altering a single bit of the plaintext (Table 4.49).

These results suggest the implemented CME scheme could provide higher levels of security than given by the other algorithms. This result is in keeping with the basing of its security on VC models, as well as in keeping with the high levels of security proposed by studies such as Priyadarsini and Ayyagari (2013), who posited a security

level of  $n!$  could be achieved using Hadamard matrices of size  $n$  by  $n$  to create Shrikhande graphs for encoding images. In the method proposed by Priyadarsini and Ayyagari (2013) the adjacency matrix for the Shrikhande graph operates as the private key. The level of security achieved by the scheme shows the usefulness of matrices as encryption keys, though the proposed CME scheme differs from that proposed by Priyadarsini and Ayyagari (2013) as it does not generate a graph for encryption, and the CME key matrix utilizes codewords and blank padding spaces.

**Sub-question 2: *How is the level of security achieved in the proposed method?***

**Answer:** The strength of the CME scheme relies heavily on its particular underlying key structure. The use of a partially-occupied matrix with a codeword alphabet of bit strings allows for a higher level of computational complexity. The random nature of the padding coordinates, and the use of multiple locations for each codeword create a non-singular mapping, so that each plaintext has many possible ciphertexts. This non-singular mapping gives a high-level of security against known and chosen plaintext attacks, as well as frequency analysis. The key matrix is protected from brute force attacks by both its size and the use of a codeword alphabet. Each entry in the matrix has multiple possible entries, rather than being either a 1 or 0. This drastically increases the number of possible key matrices, and therefore the number of operations an adversary would be required to compute in order to break the scheme successfully, as shown in Equations 4.1.4.i through 4.1.4.iii.

**Sub-question 3: *What is the reduction in computational overhead in the proposed scheme from comparable algorithms?***

**Answer:** The comparison of efficiency in the different schemes resulted in CME offering a faster setup time than AES, RC4 and ECC, though the 2 out of 2 VC scheme gave the fastest setup time (Table 4.3.1). The CME scheme offered reduced memory requirements in setup compared to that of the 128-bit AES and RC4 implementations (Tables 4.33, 4.42), though the ECC scheme had lower memory requirements still (Table 4.41). In the comparison of the 8-bit CME scheme and the 128-bit AES implementation, the encryption and decryption time increased linearly in CME, where the encryption time for AES was constant (Table 4.34). Similarly, the RC4 algorithm was significantly faster than the CME scheme in encryption on all data sizes, and in decryption of the larger data

sizes, both of which remained constant in RC4 (Table 4.43). However, while the CME scheme required greater time for encryption for the longer plaintext sizes than AES, decryption in CME was consistently faster than its AES counterpart on all testing data (Table 4.34). The CME scheme also offered a lower memory requirement in encryption and decryption than the AES and RC4 implementations (Tables 4.35, 4.44). In comparison to the VC scheme, the 4-bit CME algorithm offered faster encryption on all plaintext strings, and faster decryption on the longer plaintext (Table 4.30). The memory requirements for setup in VC and CME were similar (Table 4.31). These results present CME as a scheme that is of comparable efficiency in many areas to the tested algorithms, though further optimisations would be necessary to ensure it is truly competitive.

#### **5.1.4 Hypothesis 1: Graphic-based methods provide a better level of security with lower overheads than classical encryption techniques**

**Result:** Indeterminate

**Explanation:** While the overall results for security show that the 8-bit CME scheme gives a higher level of security than the 128-bit AES and RC4 implementations, the efficiency in regards to encryption time increased linearly in the CME scheme (Table 4.1). As the byte implementation of CME could be considered a stream cipher that operated on data byte by byte, the time complexity of the algorithm is linear, and grows with the size of the plaintext. The 128-bit AES system operates on set block sizes of data greater than that of the CME scheme, and as such the AES scheme offers a slower rate of growth in time complexity than that of the CME scheme (Table 4.34). The RC4 algorithm also gave better performance with regards to time in the encryption of data than CME, though required higher levels of memory (Tables 4.43, 4.44). This suggests that the overall efficiency in encryption of data, and therefore the overheads incurred by encryption may be higher in the CME scheme than in the AES and RC4 implementations. The results for the efficiency of AES are supported by the prior studies, such as Jeeva et al. (2012), whose test results gave AES a high overall efficiency rating. Similarly, the results of testing RC4, which produced a faster encryption and decryption time on all data than AES are supported by prior studies such as Singhal and Raina (2011), who tested the comparative efficiency of RC4 and AES, and found that AES was slower and required more memory over all the tests. That the higher level of theoretical security provided by CME resulted in lower efficiency is in keeping with the results of Bhat et al. (2015), who found that

AES was significantly less efficient than DES, as a result of the increased key size and security requirements. The results of the study in the comparison of the efficiency of ECC and AES are also in keeping with prior studies, as Prachi et al. (2015) found that ECC was more efficient than AES, which is reflected in the efficiency results of the implemented ECDH scheme, which was faster in setup than the implemented AES algorithm. The comparison between ECC, AES and RC4 do suggest that the efficiency of the alternative graphic based methods provides for a high level of efficiency that is comparable with that of more traditional encryption methods. Further experimentation is required to fully explore and test the hypothesis, and other traditional ciphers such as Blowfish could be implemented to compare with graphic methods.

#### **5.1.5 Hypothesis 2: The proposed encryption system based around graphic methods is computationally secure against attacks**

**Result:** Accepted

**Explanation:** The results of the testing and analysis show that the CME scheme is computationally secure against brute force attacks. The number of possible key matrices in schemes of 4-bits or more are many orders of magnitude higher than the number of possible keys in current systems such as AES and RC4, shown in Equations 4.1.4.ii, 4.6.2.i, and 4.6.4.i. As 128-bit AES and RC4 schemes are considered to be computationally secure against brute force attacks, and the 4-bit CME scheme is 276 orders of magnitude greater, then the 4-bit CME scheme, and any CME scheme greater than 4-bits, must also be computationally resistant to brute force attacks. In regards to chosen plaintext and known plaintext attacks, the diffusion of statistical properties of the plaintext through the use of a key string, multiple coordinate locations for each codeword, and the addition of blank padding coordinates result in the CME scheme being theoretically secure against known and chosen plaintext attacks. The scheme has also been shown secure against frequency analysis, based on test data gathered in relation to the occurrences of full and blank coordinates within the ciphertext (Tables 4.7 to 4.9). The avalanche effect of the scheme also assists in security against chosen plaintext attacks (Table 4.10). Finally, the non-singular mapping resulting from the design of the algorithm ensures a high level of security against both known and chosen plaintext attacks, as the plaintext input of the algorithm gives many different ciphertext outputs for a single key matrix. As a result, the knowledge of any or all of the plaintext would provide little to no

information about the particular coordinates within the ciphertext, as the adversary would still need to guess for each coordinate whether it was empty or full.

## **5.2 DISCUSSION**

This section discusses the implications of the results. Section 5.2.1 details the specific programs created for testing and how their particular functions were determined. Section 5.2.2 looks at how the study has given information about the security benefits, efficiency trade-offs, and applications of alternative graphic based ciphers. Section 5.2.2 then explores the refinements and alterations that were made to the proposed system during the process of testing and optimizing the scheme.

### **5.2.1 Testing Algorithms**

The programs created for the testing of each of the comparable algorithms were designed to be as thorough as possible. The avalanche effect program utilized in the study was designed in two sections. The first section was used to test the resulting ciphertext from the bit-string CME and 2 out of 2 VC schemes. It took two binary ciphertext strings as input, where the original plaintext used to generate them differed by exactly one bit, and calculated the percentage of bits that occurred in the same position in both ciphertexts. This gave a good overview of the avalanche effect in each of these binary algorithms. The second section of the avalanche analysis program was designed to test the byte version CME and the 128-bit RC4 and AES. As all three algorithms gave arrays of either bytes or integers as the ciphertext output, the program looked at both the total percentage of bytes that occurred in both ciphertexts, and the total percentage of bytes that occurred in the same position in both ciphertexts. The first test, the percentage of bytes occurring in both ciphertexts, grows as the length of the ciphertext grows, as there are only 256 bytes total. As the length of the plaintext increases, all algorithms trended towards 100% for this value, with CME reaching this value more rapidly, as the ciphertext output for the CME scheme is always four times the length of the plaintext, while AES and RC4 only give an increase in length from ciphertext to plaintext if padding is required. The second test, the total percentage of bytes that occur in the same position, gave a better view of the effect a single bit change in plaintext had on the ciphertext. The results from the CME algorithm hovered consistently at less than 1% of the total bytes occurring in the same position, while AES gave a result of 25-49% of bytes occurring in the same position. RC4 resulted in 97-99.9% of bytes occurring in the same position.

The frequency analysis program developed to examine the occurrences of particular bytes within the ciphertext output of the schemes was also developed in two modules. The first module was utilized in the testing of 128-bit AES and RC4, and checked only the number of occurrences for each individual byte. The results were then given as the number of bytes that occurred with a given frequency. The second module was developed specifically to test the byte version of the CME scheme. This module checked the number of occurrences of each set of coordinates, and then also checked – based on the provided key matrix – whether the given coordinates were blank padding, or contained a message character. The results were then given as the number of empty and full characters which occurred with a given frequency. This enabled the examination of trends within the algorithm, and gave practical results for the theoretical analysis of the probabilities with which blank or full coordinate pairs occur.

### **5.2.2 Benefits and Applications of Graphic Based Ciphers**

The use of alternative graphic methods to build ciphers for encryption offers a high level of security against attacks, both brute force and cryptanalysis. While the particular structure of the keys can incur higher computational overheads than traditional methods, the use of these alternative structures for secure communication requires further study. Use of systems like CME could offer higher levels of security in situations in which a slight decrease in efficiency was an acceptable trade-off for increased security. With the advent of quantum computers, and the constant increase in available computational processing power, traditional ciphers require higher and higher levels of security, which in most cases is resolved by a longer key length.

Quantum computing has been posited as a danger to current encryption technologies. According to the algorithm introduced by Grover (1996), the computational complexity of a system of  $O(N)$  in classical computing can be translated into one of  $O(N^{1/2})$  in a quantum computer. This gives 128-bit AES a security level of  $2^{64}$ , equivalent to the now-defunct DES. The 8-bit CME scheme, which offers  $257^{65536}$  different key matrices, would therefore be reduced to the equivalent security level of  $257^{32768}$ . This would still theoretically offer a sufficient level of security. The currently recommended AES 256-bit would be reduced to a security level of  $2^{128}$ , and would as such be many orders of magnitude less than the security of the 8-bit CME scheme. The alternative structure of the key for the CME scheme therefore offers a high level of resistance to brute force attacks, even given the advent of quantum computers.

The use of CME or other alternative graphic based systems as a stream cipher



could offer a theoretical alternative to the now-defunct RC4, which has been proven vulnerable to several specific cryptanalytic attacks. While other stream ciphers, such as Spritz (Rivest & Schuldt, 2014), have been proposed to modify RC4 to prevent these attacks, as yet very little literature is available to provide support for their security. Because there is currently little standardisation in stream ciphers to replace RC4, the opportunity exists for an optimized version of CME to assist in this particular space in applications such as TLS that utilize stream ciphers for encryption on a day-to-day basis.

The structure of the CME scheme key matrix also allows for expansion into higher levels of security. The tested algorithm used a single key string of the first  $x$  coordinate for each codeword. As each codeword has  $\binom{2^{2n}}{2(2^n)}$  locations, each consisting of an  $x,y$  coordinate pair, the scheme allows for the addition of multiple extra key strings, up to a final total of  $2\binom{2^{2n}}{2(2^n)}$ , to be used in the encryption of the data. The addition of these extra key strings in stronger schemes can be done without increasing the size of the key matrix, as the key matrix already contains these generated key strings, and their inclusion would require only one extra exclusive-OR operation per key string into the encryption and decryption loops. As such, the number of key strings can be increased without requiring an increase in the number of communicated keys.

The security of the proposed CME scheme rests in part on the addition of blank padding characters. Because each ciphertext contains an equal number of blank and full coordinates, and both occur even at the highest frequencies within the ciphertext, an adversary would be required to guess, for each coordinate within the ciphertext, whether it contained an encrypted message codeword, or was blank padding. Because of the addition of blank padding characters, the plaintext character locations do not necessarily match their location in the ciphertext. As such, even were an adversary to successfully guess a particular coordinate was full, he would only be able to guess at where in the plaintext that particular coordinate fell.

The use of alternative or unusual key structures enables graphic-based ciphers to use simplified algorithms, as the security rests on the structure, rather than the number of rounds of substitution or permutation. This simplicity of design could be of note in schemes that utilize stream ciphers for operation, as the CME scheme can operate byte by byte, and decryption is, as shown in the results, fast and efficient when compared with industry standards. This high-efficiency in decryption also suggests the proposed system could be utilized in high-security situations where fast access to encrypted data is of greater concern than the speed for the encryption and storage of the data.

### **5.2.3 Difficulties and Optimizations in Implementation**

The proposed graphic CME scheme required several iterations of refining to reach an acceptable result. The testing of the algorithm in regards to frequency analysis, the binomial probability of successfully locating the full coordinates in a given ciphertext, and the speed of the algorithm all provided important data for the refining of the scheme.

The original algorithm had failed to account for the overall trends in frequency within a non-random plaintext string. The addition of the key string to counter this frequency information was therefore a highly important improvement, as early versions without the key string proved vulnerable to frequency analysis given large blocks of ciphertext, where blank coordinates did not occur at higher frequencies. The altering of the structure for the blank coordinates then further mitigated this effect. The total number of blank coordinate entries was calculated to equal the total number of codewords. Each blank coordinate entry was then assigned a list of the same length as each codeword's coordinate list, and these blank entry lists were then populated with randomly chosen empty locations. The algorithm then went through the same process to choose a particular blank coordinate that was required to choose a given codeword location. A blank entry was picked at random, and then a location within that blank entry's list was also picked at random. This resulted, in conjunction with the addition of the key string, in a flatter frequency distribution, and both blank and full coordinates occurring at each frequency, in equal distributions.

The original version of the algorithm also failed to produce a fixed-length ciphertext output. The number of added padding coordinates was determined randomly, and as such, the length of the ciphertext was variable. While this assisted in confusing the plaintext, the possibility existed, however small, that the algorithm would fail to add any padding coordinates, and the ciphertext would consist only of message coordinates. This possibility was dealt with by fixing the length of the ciphertext, and ensuring that equal numbers of padding and codeword coordinates were included in all ciphertext outputs. This also gave a fixed binomial probability for any ciphertext, that for any one coordinate the probability of being either blank or full was 1:1.

Early versions of the CME algorithm also suffered in efficiency due to their programming. Part of the process for refining the algorithm was the editing of the code to make it more efficient. Several extra loops and operations were discovered within the code that could be removed without affecting the algorithm, and replaced with simpler and lower cost operations. This optimization process allowed for the creation of an

algorithm that operated with linear time complexity  $O(n)$ .

### **5.3 CONCLUSION**

This chapter has discussed the results and implications of the study reported in Chapter 4. The research questions and hypotheses were answered and explained, and the findings explored in depth in relation to the previous studies and literature examined in Chapters 2 and 3.

The next and final chapter will draw conclusions from the research. It will discuss the limitations of the study in design and execution, and give recommendations for future research based on the results.

# Chapter 6

## Conclusion

### 6.0 INTRODUCTION

The previous chapter discussed the results of the study, and the implications of these findings. It explained the difficulties encountered in the development of the implementations for each of the algorithms, and looked at the benefits and drawbacks of utilizing graphic-based methods for encryption. The previous chapter also examined the potential uses of alternative key structures like those proposed in the CME scheme.

This chapter enumerates the conclusions of the research. Section one discusses in depth the limitations of the study. Section two then offers recommendations for future research based on the study conducted in this thesis. Then section three gives the final summary and conclusions of the research.

### 6.1 LIMITATIONS OF RESEARCH

During the course of the study, all efforts were made to ensure the results were as even and unbiased as possible. However, several factors must be taken into consideration when examining the results. The programming of the different algorithms implementations is discussed in section 6.1.1, while the comparability of ECC with AES, RC4 and CME is discussed in 6.1.2. The impact of altering the VC scheme to operate on binary characters is then discussed in 6.1.3.

#### 6.1.1 Programming Limitations

The nature of the conducted study required that the system have custom implementations for the different algorithms. It was necessary to insert code to time the functions, and to measure the currently occupied memory. It was also necessary to insert calls to custom programs for measuring the ciphertext frequencies and the overall avalanche effect. These custom implementations may therefore have impacted upon the results of the study. While every effort was made to ensure the implementations were efficient and accurate, it is possible that another programmer writing their own implementations would achieve different results.

The implementations created for the purpose of the research were also affected by

the algorithms they were being compared with. The implementations for VC and the bit-string version of CME were created as high level implementations that operated on strings of input. This meant these implementations were automatically less efficient in memory and time than the implementations of AES, RC4 and byte-oriented CME, which operated on arrays of bytes and integers. This is because the byte implementations use primitive variables, which results in smaller memory requirements and more efficient computation overall. Strings in Java are a more complex variable type, and any change made to a String in Java requires the creation and allocation of an entirely new String. This leads to wasted memory and computation. It also results in the VC and bit-string version of CME test results being less likely to generalize well enough to give data on the efficiency of low-level implementations of the same algorithms.

The implementation of the CME scheme was also affected by its choice of pseudorandom number generator. Due to the constrained problem domain of this thesis, only one pseudorandom number generator was implemented, the inbuilt Java function. Random number generation is a widely researched field in cryptography and security, and the security of an algorithm which utilizes the generation of random or pseudorandom numbers relies on the security of these generators. As such, it is necessary to acknowledge that future research into CME schemes requires exploration of the possible generators, and their effect on the overall scheme. The use of Java's inbuilt random number generator may impede the ability of these results to be more generally compared with stream ciphers that utilize purpose-built secure random number generators.

It must be noted that a more efficient comparison between the algorithms would have utilized schemes with 256 bit or larger keys. However, due to export controls surrounding the dissemination of strong cryptographic algorithms, standard implementations for AES and RC4 did not offer key sizes larger than that of 128. The inbuilt Java functions used to develop the code for these algorithms set a maximum size of 128 for key generation. The use of 128-bit algorithms in comparison to the implemented CME scheme results in less generalizable data, as the scheme implemented would be closer in operation to 256-bit AES.

### **6.1.2 Comparing Asymmetric and Symmetric Systems**

The inclusion of ECC in the study was necessary due to its prevalence as a graphic encryption system. However, it is important to note that ECC is an asymmetric encryption algorithm, while the others used in the study are symmetric. This automatically impacts on the results of comparison with ECC. Jeeva et al. (2012) found the original Diffie-

Hellman protocol slow to execute, and while the ECC version of this protocol is quicker due to its smaller key size, the execution of asymmetric systems can be computationally slower than that of symmetric systems. In the results given in Chapter 4, the execution of the ECDH algorithm was on par with that of the AES setup time. AES 128 took 409 ms (mean) and ECDH with a 192-bit key took 359.5 ms (mean). It must also be taken into account that the most efficient algorithm for breaking the 192-bit curve of the implemented ECDH protocol results in the protocol offering equivalent security to a symmetric 80-bit key, lower than that of 128-bit AES (Stallings, 2014). In order to achieve the equivalent security of 128-bit AES, a much larger elliptic curve would be required. As such, the comparison of the ECDH system is limited by the differences in the overall architecture, and the way its particular function impacts on its efficiency. The resulting comparison between the ECDH scheme and the symmetric cipher implementations therefore is limited in impact and scope, due to the different architectures and security levels. This results in a lack of generalizability to higher security ECC schemes, which are more likely implemented in modern technologies.

### **6.1.3 Binary Implementation of Visual Cryptography**

Classical VC schemes operate on images and their pixels. A secret image is split into shares which contain arrays of subpixels. The comparison between the VC and CME schemes operated on strings of binary plaintext, and so it was necessary to design an implementation which operated on strings instead of images. The equivalent VC implementation for encryption of strings was based on the 2-out-of-2 scheme originally proposed by Naor and Shamir (1995), and utilized share creation from Kafri and Keren (1988). The translation of the VC scheme into one which operates in a different domain was a trivial operation, but may have impacted on the results, as the schemes are designed for efficiency and security in the encryption of images, not text. As such, the results of the utilized VC scheme may not necessarily reflect the operational efficiency of a more standard, classical VC scheme which operates on images.

## **6.2 FUTURE RESEARCH**

The use of alternative key structures and graphic-based ciphers for encryption requires further study. The results given in Chapter 4 show a high security level with comparable efficiency to current cryptographic standards. This security level requires further examination in relation to cryptanalysis. The advent of quantum computing will likely

result in the weakening of current symmetric encryption systems, as well as the destruction of public-key infrastructures. The quantum algorithm proposed by Grover (1996) gives a key of size  $2^n$  the equivalent security of a symmetric key size of  $2^{n/2}$ . A scheme with computational complexity  $O(N)$  then has complexity  $O\left(N^{\frac{1}{2}}\right)$ . This would render 128-bit AES insecure, as the computational complexity would be reduced to  $2^{64}$ , equal to the 64-bit DES algorithm, depreciated in 2001. The results of Chapter 4 suggest that alternative key structures could provide a pathway for the development of secure post-quantum cryptography. The number of possible key matrices for an 8-bit scheme in CME remains very high, even when subjected to Grover's algorithm, with a revised security level of  $2^{37}$  (2768). The security level of CME opens avenues for further research into the resistance of CME and other alternative systems to quantum-based attacks. Future study could examine the particular matrix structure of the CME key scheme, and how this could be implemented in post-quantum cryptography.

Further study could also be done in comparison with other stream ciphers that are currently part of the ESTREAM portfolio such as HC-128 (Wu, 2008), and block ciphers such as Blowfish (Schneier, 1993). The expanding of comparable algorithms would allow for the system to be more accurately placed in the current cryptographic landscape. The research conducted in this study was limited in scope due to algorithm availability, and as such there are further opportunities to look at the comparison of alternative key systems such as CME with other well-developed and industry adopted algorithms. Given that studies such as Thakur and Kumar (2011) suggested that Blowfish gave even better overall performance than AES, it would be expected that Blowfish would result in a faster encryption time than CME, but further exploration of the comparative security would provide a detailed look at the trade-offs between efficiency and security. A comparison between HC-128 and a byte-level version of CME would give researchers the opportunity to examine the ways in which the two algorithms differed in efficiency and security, to further the current comparison between CME and RC4. Because the 8-bit or byte-level CME scheme is a word-oriented stream cipher, the comparison with other proposed stream ciphers would give further understanding to how CME could be placed as an alternative within the current encryption infrastructure. Further investigation could also examine the avenues for incorporating CME into current stream cipher-based technologies such as TLS.

Future research could also explore implementing CME algorithms with multiple key strings. The addition of differing key strings could provide for higher levels of

security and flatter distributions across the board. The key matrix setup of CME allows for the use of up to  $2\binom{2^{2n}}{2^{2n}}$  key strings in the encryption process, without communicating any extra keys. This ability to further adapt and customize the algorithm provides for research opportunities into the overall effect on security and efficiency when increasing the number of key strings used.

Ciphers that provide non-singular mappings of plaintext to ciphertext require further investigation. The addition of this chaos element adds further layers of obscurity and obfuscation to the encryption system. The non-singular mapping provides for security against known and chosen plaintext attacks, as shown in the security analysis of both CME and VC. Further research should explore other options for encryption systems that provide these non-singular mappings, particularly within the realm of graphic-based systems.

Further study could be done to explore ways in which the CME scheme may be optimized for greater efficiency. Improvements to the algorithm design and the schema could allow for a low-level implementation which provided the level of security shown in this thesis, while also allowing for a higher level of efficiency. The improvement of CME with regards to efficiency and time complexity would allow for implementations which could be utilized in domains with limited computing power, such as smart cards, and for integration into technologies such as TLS, which require high-performance stream ciphers to function.

The indeterminate result of Hypothesis 1 opens avenues for further research into comparisons between graphic-based and traditional cryptographic methods. The results of this study were unable to conclusively prove that graphic-based methods for encryption offered higher levels of security with lower overheads than traditional methods. Further research into this hypothesis would include comparing CME and other graphic-based systems against a variety of different traditional ciphers, on larger plaintext sizes and documents. Further research should also compare the performance of traditional and graphic-based systems in different domains, such as image encryption or incorporation into email clients.

The use of 128-bit AES and RC4 in this study was due to cryptographic export constraints. Further study could explore the comparative results of 256-bit AES with the 8-bit CME scheme, and with recently developed 256-bit stream ciphers. The results of the efficiency tests between AES and CME suggest that the CME scheme should offer competitive levels of efficiency with the 256-bit AES scheme. The use of alternative random number generators could also be explored, to determine the effect different



generators have on the overall efficiency of the algorithm.

The implementation options for graphic-based systems such as CME should be further researched. Low-level implementations for hardware such as smart cards could prove of use in banking situations, as the results of Chapter 4 suggest that CME offers a high-security scheme with a relatively low memory requirement in comparison to other schemes. As architecture such as smart cards usually operates on a limited memory capacity, schemes that can offer reduced memory requirements are of value in this field.

The possibility of implementing the CME scheme in technologies such as TLS, which is designed around the use of stream ciphers, and previously utilized RC4, deserves further exploration. The improvements offered by CME in terms of memory requirements, avalanche effect, and brute-force resistance make it of relevance to securing online interactions. The development of a transport layer level scheme for a high-efficiency implementation could provide an option for addressing the gap left by the depreciation of RC4.

### **6.3 CONCLUSION**

The research conducted in this study has explored the possibilities offered by alternative key structures and graphic-based methods for the development of encryption algorithms. These structures present a significant research opportunity for secure communication, and require further study and exploration. The results suggest that systems based around alternative key structures and graphic-methods could offer high levels of security while remaining competitively efficient in execution. Further research into optimization and application is required to fully explore these possibilities.

## References

- Afzal, M., Kausar, F., & Masood, A. (2006, 13-14 Nov. 2006). *Comparative Analysis of the Structures of eSTREAM Submitted Stream Ciphers*. Paper presented at the ICET '06, International Conference on Emerging Technologies, 2006. doi:10.1109/ICET.2006.335958
- Agnarsson, G., & Greenlaw, R. (2007). *Graph Theory: Modelling, Applications, and Algorithms*. New Jersey: Pearson Education Ltd.
- Akhter, F. (2015, 26-27 Nov. 2015). *A novel Elliptic Curve Cryptography scheme using random sequence*. Paper presented at the 2015 International Conference on Computer and Information Engineering (ICCIE). doi:10.1109/CCIE.2015.7399314
- Amara, M., & Siad, A. (2011, 9-11 May 2011). *Elliptic Curve Cryptography and its applications*. Paper presented at the 7th International Workshop on Systems, Signal Processing and their Applications (WOSSPA), 2011. doi:10.1109/WOSSPA.2011.5931464
- Amounas, F., & Kinani, E. H. E. (2012, 20-21 April 2012). *An elliptic curve cryptography based on matrix scrambling method*. Paper presented at the National Days of Network Security and Systems (JNS2), 2012. doi:10.1109/JNS2.2012.6249236
- Anderson, R. (2008). *Security Engineering: A Guide to Building Dependable Distributed Systems* (2nd ed.): John Wiley & Sons.
- Arumugam, S., Lakshmanan, R., & Nagar, A. K. (2013). Graph access structures with optimal pixel expansion three. *Information and Computation*, 230, 67-75. doi:10.1016/J.IC.2013.07.002
- Ateniese, G., Blundo, C., De Santis, A., & Stinson, D. R. (1996). Visual Cryptography for General Access Structures. *Information and Computation*, 129(2), 86-106. doi: 10.1006/INCO.1996.0076
- Ateniese, G., Blundo, C., Santis, A. D., & Stinson, D. R. (2001). Extended capabilities for visual cryptography. *Theoretical Computer Science*, 250(1-2), 143-161. doi:10.1016/S0304-3975(99)00127-9
- Austen, J. (2006). *Pride and Prejudice*. London, Great Britain: Headline Review.
- Bai, Q.-h., Zhang, W.-b., Jiang, P., & Lu, X. (2012, 11-13 Aug. 2012). *Research on Design Principles of Elliptic Curve Public Key Cryptography and Its Implementation*. Paper presented at the International Conference on Computer

- Science & Service System (CSSS), 2012. doi:10.1109/CSSS.2012.310
- Bhat, B., Ali, A. W., & Gupta, A. (2015, 15-16 May 2015). *DES and AES performance evaluation*. Paper presented at the International Conference on Computing, Communication & Automation (ICCCA), 2015. doi:10.1109/CCAA.2015.7148500
- Blakley, G. R., & Kabatiansky, G. (2011). Secret Sharing Schemes. In H. A. van Tilborg & S. Jajodia (Eds.), *Encyclopedia of Cryptography and Security* (pp. 1095-1097): Springer US. doi:10.1007/978-1-4419-5906-5\_389
- Blundo, C., Ciamato, S., & De Santis, A. (2006). Visual cryptography schemes with optimal pixel expansion. *Theoretical Computer Science*, 369(1–3), 169-182. doi:10.1016/J.TCS.2006.08.008
- Chan-Hyoung, P., Hong-Yeop, S., & Kyu Tae, P. (1998). *Existence and classification of Hadamard matrices*. Paper presented at the 1998 Fourth International Conference on Signal Processing Proceedings, 1998. ICSP '98. doi:10.1109/ICOSP.1998.770165
- Chandra, S., Paira, S., Alam, S. S., & Sanyal, G. (2014, 17-18 Nov. 2014). *A comparative survey of Symmetric and Asymmetric Key Cryptography*. Paper presented at the 2014 International Conference on Electronics, Communication and Computational Engineering (ICECCE). doi:10.1109/ICSEMR.2014.7043664
- Chen, Y. C., Horng, G., & Tsai, D. S. (2012). Comment on "Cheating Prevention in Visual Cryptography". *IEEE Transactions on Image Processing*, 21(7), 3319-3323. doi:10.1109/TIP.2012.2190082
- Cohn, P. M. (2000). *Introduction to ring theory*: Springer Science & Business Media.
- Davidoff, G., Sarnak, P., & Valette, A. (2003). *Elementary number theory, group theory and Ramanujan graphs* (Vol. 55): Cambridge University Press.
- De Prisco, R., & De Santis, A. (2014). On the Relation of Random Grid and Deterministic Visual Cryptography. *IEEE Transactions on Information Forensics and Security*, 9(4), 653-665. doi:10.1109/TIFS.2014.2305574
- Delgosha, F., & Fekri, F. (2006). Public-key cryptography using paraunitary matrices. *IEEE Transactions on Signal Processing*, 54(9), 3489-3504. doi:10.1109/TSP.2006.877670
- Deligiannidis, L. (2015, 27-29 May 2015). *Elliptic curve cryptography in Java*. Paper presented at the IEEE International Conference on Intelligence and Security Informatics (ISI), 2015. doi:10.1109/ISI.2015.7165975
- Ding, J., Petzoldt, A., & Wang, L.-C. (2014). The Cubic Simple Matrix Encryption

Scheme. *Post-Quantum Cryptography*, 76.

- Ding, J., & Yang, B.-Y. (2009). Multivariate public key cryptography *Post-Quantum Cryptography* (pp. 193-241): Springer. doi:10.1007/978-3-540-88702-7\_6
- Droste, S. (1996). New Results on Visual Cryptography. In N. Kobitz (Ed.), *Advances in Cryptology — CRYPTO '96* (Vol. 1109, pp. 401-415): Springer Berlin Heidelberg. doi:10.1007/3-540-68697-5\_30
- Feng, J.-B., Wu, H.-C., Tsai, C.-S., Chang, Y.-F., & Chu, Y.-P. (2008). Visual secret sharing for multiple secrets. *Pattern Recognition*, 41(12), 3572-3581. doi:10.1016/J.PATCOG.2008.05.031
- Fluhrer, S., Mantin, I., & Shamir, A. (2001). *Weaknesses in the key scheduling algorithm of RC4*. Paper presented at the International Workshop on Selected Areas in Cryptography.
- Galbraith, S., & Menezes, A. (2005). Algebraic curves and cryptography. *Finite Fields and Their Applications*, 11(3), 544-577. doi:10.1016/J.FFA.2005.05.001
- Giorgobiani, G., Kvaratskhelia, V., & Menteshashvili, M. (2015, Sept. 28 2015-Oct. 2 2015). *Some properties of Hadamard matrices*. Paper presented at the Computer Science and Information Technologies (CSIT), 2015. doi:10.1109/CSITechnol.2015.7358251
- Grover, L. K. (1996). *A fast quantum mechanical algorithm for database search*. Paper presented at the Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, Philadelphia, Pennsylvania, USA. doi:10.1145/237814.237866
- Hajiabolhassan, H., & Cheraghi, A. (2010). Bounds for visual cryptography schemes. *Discrete Applied Mathematics*, 158(6), 659-665. doi:10.1016/j.dam.2009.12.005
- Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2), 147-160.
- Hosnieh, R., Martin von, L., & Christoph, M. (2013). Cryptography in Electronic Mail *Theory and Practice of Cryptography Solutions for Secure Information Systems* (pp. 406-427). Hershey, PA, USA: IGI Global. doi:10.4018/978-1-4666-4030-6.ch016
- Hou, Y. C., Wei, S. C., & Lin, C. Y. (2014). Random-Grid-Based Visual Cryptography Schemes. *IEEE Transactions on Circuits and Systems for Video Technology*, 24(5), 733-744. doi:10.1109/TCSVT.2013.2280097
- Hu, C. M., & Tzeng, W. G. (2007). Cheating Prevention in Visual Cryptography. *IEEE Transactions on Image Processing*, 16(1), 36-45. doi:10.1109/TIP.2006.884916

- Hurley, B., & Hurley, T. (2011). Group ring cryptography. *International Journal of Pure and Applied Mathematics*, 69(1), 67-86.
- Jaya, Malik, S., Aggarwal, A., & Sardana, A. (2011, 11-14 Dec. 2011). *Novel authentication system using visual cryptography*. Paper presented at the 2011 World Congress on Information and Communication Technologies (WICT). doi:10.1109/WICT.2011.6141416
- Jeeva, A., Palanisamy, D. V., & Kanagaram, K. (2012). Comparative analysis of performance efficiency and security measures of some encryption algorithms. *International Journal of Engineering Research and Applications (IJERA) ISSN*, 2248-9622.
- Jie, L., & King, B. (2013, 4-7 Aug. 2013). *Smart card fault attacks on elliptic curve cryptography*. Paper presented at the IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS), 2013. doi:10.1109/MWSCAS.2013.6674882
- Joseph, S. K., & Ramesh, R. (2015, 16-19 Dec. 2015). *Random grid based visual cryptography using a common share*. Paper presented at the 2015 International Conference on Computing and Network Communications (CoCoNet). doi:10.1109/CoCoNet.2015.7411259
- Joux, A., & Vitse, V. (2012). Cover and decomposition index calculus on elliptic curves made practical *Advances in Cryptology–EUROCRYPT 2012* (pp. 9-26): Springer. doi:10.1007/978-3-642-29011-4\_3
- Kafri, O., & Keren, E. (1988). Method and apparatus of encryption of optical images: Google Patents.
- Kamarulhaili, H. (2010, 7-10 Aug. 2010). *Generating Elliptic Curves Modulo  $p$  for Cryptography Using Mathematica Software*. Paper presented at the Seventh International Conference on Computer Graphics, Imaging and Visualization (CGIV), 2010. doi:10.1109/CGIV.2010.22
- Kang, I., Arce, G. R., & Lee, H. K. (2011). Color Extended Visual Cryptography Using Error Diffusion. *IEEE Transactions on Image Processing*, 20(1), 132-145. doi:10.1109/TIP.2010.2056376
- Klein, A. (2008). Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography*, 48(3), 269-286. doi:10.1007/s10623-008-9206-6
- Klein, A., & Wessler, M. (2007). Extended visual cryptography schemes. *Information and Computation*, 205(5), 716-732. doi:10.1016/j.ic.2006.12.005
- Klisowski, M., & Ustimenko, V. (2010, 18-20 Oct. 2010). *On the implementation of*

- public keys algorithms based on algebraic graphs over finite commutative rings.* Paper presented at the Proceedings of the 2010 International Multiconference on Computer Science and Information Technology (IMCSIT). doi:10.1109/IMCSIT.2010.5679687
- Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of computation*, 48(177), 203-209.
- Kofahi, N. A., Turki, A.-S., & Khalid, A.-Z. (2003, 30-30 Dec. 2003). *Performance evaluation of three encryption/decryption algorithms.* Paper presented at the IEEE 46th Midwest Symposium on Circuits and Systems, 2003.
- Kotorowicz, J., Romanczuk, U., & Ustimenko, V. (2011, 18-21 Sept. 2011). *On the implementation of stream ciphers based on a new family of algebraic graphs.* Paper presented at the Federated Conference on Computer Science and Information Systems (FedCSIS), 2011.
- Krämer, J. (2015). *Why Cryptography Should Not Rely on Physical Attack Complexity.* Singapore: Springer.
- Krebs, M., & Shaheen, A. (2011). *Expander Families and Cayley Graphs : A Beginner's Guide.* Cary: Oxford University Press.
- Leca, C. L., & Rincu, C. I. (2014, 29-31 May 2014). *Combining point operations for efficient elliptic curve cryptography scalar multiplication.* Paper presented at the 10th International Conference on Communications (COMM), 2014. doi:10.1109/ICComm.2014.6866676
- Liu, F., & Wu, C. (2011). Embedded Extended Visual Cryptography Schemes. *IEEE Transactions on Information Forensics and Security*, 6(2), 307-322. doi:10.1109/TIFS.2011.2116782
- Liu, F., Wu, C., & Lin, X. (2010a). A new definition of the contrast of visual cryptography scheme. *Information Processing Letters*, 110(7), 241-246. doi:10.1016/j.ipl.2010.01.003
- Liu, F., Wu, C., & Lin, X. (2010b). Step Construction of Visual Cryptography Schemes. *IEEE Transactions on Information Forensics and Security*, 5(1), 27-38. doi:10.1109/TIFS.2009.2037660
- Liu, F., Wu, C. K., & Lin, X. J. (2008). Colour visual cryptography schemes. *Information Security, IET*, 2(4), 151-165. doi:10.1049/iet-ifs:20080066
- Liu, M., Han, L., & Wang, X. (2011). On the equivalent keys in multivariate cryptosystems. *Tsinghua Science and Technology*, 16(3), 225-232. doi:10.1016/S1007-0214(11)70033-5

- Loehr, N. (2014). *Advanced Linear Algebra*. Bosa Roca: CRC Press.
- Lu, S., Manchala, D., & Ostrovsky, R. (2011). Visual cryptography on graphs. *J. Comb. Optim.*, 21(1), 47-66. doi:10.1007/s10878-009-9241-x
- Ma, K., & Wu, K. (2014). Error Detection and Recovery for ECC: A New Approach Against Side-Channel Attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(4), 627-637. doi:10.1109/TCAD.2013.2293058
- Marin, L., Jara, A., & Skarmeta, A. F. (2012, 4-6 July 2012). *Shifting Primes: Optimizing Elliptic Curve Cryptography for Smart Things*. Paper presented at the Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012. doi:10.1109/IMIS.2012.199
- Martin, K. M. (2012). *Everyday Cryptography: Fundamental Principles & Applications*. New York: Oxford University Press.
- Martinez, V. G., & Encinas, L. H. (2013). Implementing ECC with Java Standard Edition 7. *International Journal of Computer Science and Artificial Intelligence*, 3(4), 134.
- Masadeh, S. R., Aljawarneh, S., Turab, N., & Abuerrub, A. M. (2010, 16-18 Aug. 2010). *A comparison of data encryption algorithms with the proposed algorithm: Wireless security*. Paper presented at the Sixth International Conference on Networked Computing and Advanced Information Management (NCM), 2010.
- Mehta, S., Varadharajan, V., & Nallusamy, R. (2012). Tampering resistant self recoverable watermarking method using error correction codes. *International Journal of Information and Computer Security*, 5(1), 28-47. doi:doi:10.1504/IJICS.2012.051089
- Menezes, A. J., Okamoto, T., & Vanstone, S. A. (1993). Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39(5), 1639-1646.
- Miller, V. S. (1985). *Use of elliptic curves in cryptography*. Paper presented at the Advances in Cryptology—CRYPTO'85 Proceedings.
- Modares, H., Moravejosharieh, A., & Salleh, R. (2011, 12-14 Dec. 2011). *Wireless Network Security Using Elliptic Curve Cryptography*. Paper presented at the First International Conference on Informatics and Computational Intelligence (ICI), 2011. doi:10.1109/ICI.2011.63
- Mostaghim, M., & Boostani, R. (2014, 3-4 Sept. 2014). *CVC: Chaotic visual cryptography to enhance steganography*. Paper presented at the 11th International

- ISC Conference on Information Security and Cryptology (ISCISC), 2014.  
doi:10.1109/ISCISC.2014.6994020
- Naor, M., & Shamir, A. (1995). Visual cryptography. In A. De Santis (Ed.), *Advances in Cryptology — EUROCRYPT'94* (Vol. 950, pp. 1-12): Springer Berlin Heidelberg.  
doi:10.1007/BFb0053419
- NIST. (2000). FIPS 186-2: Digital Signature Standard (DSS). Gaithersburg MD: National Institute of Standards and Technology.
- Ontiveros, B., Soto, I., & Carrasco, R. (2006). Construction of an elliptic curve over finite fields to combine with convolutional code for cryptography. *IEEE Proceedings - Circuits, Devices and Systems*, 153(4), 299-306. doi:10.1049/ip-cds:20050117
- Pal, S. K. (2007, 5-7 March 2007). *Fast, Reliable & Secure Digital Communication Using Hadamard Matrices*. Paper presented at the International Conference on Computing: Theory and Applications, 2007. ICCTA '07. doi:10.1109/ICCTA.2007.61
- Paszkievicz, A., Górska, A., Górski, K., Kotulski, Z., Kulesza, K., & Szczepański, J. (2001). *Proposals of Graph Based Ciphers, Theory and Implementations*. Paper presented at the Proceedings of the Regional Conference on Military Communication and Information Systems. CIS Solutions for an Enlarged NATO, RCMIS.
- Pateriya, R. K., & Vasudevan, S. (2011, 3-5 June 2011). *Elliptic Curve Cryptography in Constrained Environments: A Review*. Paper presented at the International Conference on Communication Systems and Network Technologies (CSNT), 2011. doi:10.1109/CSNT.2011.32
- Petit, C., & Quisquater, J.-J. (2012). On polynomial systems arising from a Weil descent *Advances in Cryptology—ASIACRYPT 2012* (pp. 451-466): Springer.
- Polak, M., Romańczuk, U., Ustimenko, V., & Wróblewska, A. (2013). On the applications of Extremal Graph Theory to Coding Theory and Cryptography. *Electronic Notes in Discrete Mathematics*, 43, 329-342.
- Polak, M., & Ustimenko, V. (2013, 8-11 Sept. 2013). *Examples of Ramanujan and expander graphs for practical applications*. Paper presented at the Federated Conference on Computer Science and Information Systems (FedCSIS), 2013.
- Prachi, Dewan, S., & Pratibha. (2015, 21-22 Feb. 2015). *Comparative Study of Security Protocols to Enhance Security*. Paper presented at the Fifth International Conference on Advanced Computing & Communication Technologies (ACCT), 2015. doi:10.1109/ACCT.2015.34



- Priyadarsini, P. L. K. (2015). A Survey on some Applications of Graph Theory in Cryptography. *Journal of Discrete Mathematical Sciences and Cryptography*, 18(3), 209-217. doi:10.1080/09720529.2013.878819
- Priyadarsini, P. L. K., & Ayyagari, R. (2013, 22-25 Aug. 2013). *Ciphers based on special graphs*. Paper presented at the International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2013 doi:10.1109/ICACCI.2013.6637215
- Qu, Y., & Hu, Z. (2010, 21-24 May 2010). *Research and design of elliptic curve cryptography*. Paper presented at the 2nd International Conference on Future Computer and Communication (ICFCC), 2010. doi:10.1109/ICFCC.2010.5497370
- Riaz, F., & Ali, K. M. (2011, 26-28 July 2011). *Applications of Graph Theory in Computer Science*. Paper presented at the Third International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN), 2011. doi:10.1109/CICSyN.2011.40
- Rivest, R. L., & Schuldt, J. C. (2014). Spritz-a spongy RC4-like stream cipher and hash function. *Proceedings of the Charles River Crypto Day, Palo Alto, CA, USA, 24*.
- Ross, A., & Othman, A. (2011). Visual Cryptography for Biometric Privacy. *IEEE Transactions on Information Forensics and Security*, 6(1), 70-81. doi:10.1109/TIFS.2010.2097252
- Schneier, B. (1993). *Description of a new variable-length key, 64-bit block cipher (Blowfish)*. Paper presented at the International Workshop on Fast Software Encryption.
- Setiadi, I., Kistijantoro, A. I., & Miyaji, A. (2015, 19-22 Aug. 2015). *Elliptic curve cryptography: Algorithms and implementation analysis over coordinate systems*. Paper presented at the 2nd International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA), 2015. doi:10.1109/ICAICTA.2015.7335349
- Shakespeare, W., & Ackroyd, P. (2006). *The Complete Works of William Shakespeare*. Glasgow, UK: Harper Collins.
- Shankar, P. (1997). Error correcting codes. *Resonance*, 2(3), 33-47. doi:10.1007/bf02838967
- Sharma, M., Garg, R. B., & Dwivedi, S. (2014, 8-10 Oct. 2014). *Comparative analysis of NPN algorithm & DES Algorithm*. Paper presented at the 3rd International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)

- (Trends and Future Directions), 2014. doi:10.1109/ICRITO.2014.7014688
- Shor, P. W. (1994). *Algorithms for quantum computation: Discrete logarithms and factoring*. Paper presented at the 35th Annual Symposium on Foundations of Computer Science, 1994.
- Shyu, S. J., Huang, S.-Y., Lee, Y.-K., Wang, R.-Z., & Chen, K. (2007). Sharing multiple secrets in visual cryptography. *Pattern Recognition*, 40(12), 3633-3651. doi:10.1016/j.patcog.2007.03.012
- Silverman, J. H., & Suzuki, J. (1998). Elliptic Curve Discrete Logarithms and the Index Calculus. In K. Ohta & D. Pei (Eds.), *Advances in Cryptology — ASIACRYPT'98: International Conference on the Theory and Application of Cryptology and Information Security Beijing, China, October 18–22, 1998 Proceedings* (pp. 110-125). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/3-540-49649-1\_10
- Singh, L. D., & Debbarma, T. (2014, 8-10 May 2014). *A new approach to Elliptic Curve Cryptography*. Paper presented at the International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), 2014. doi:10.1109/CCAA.2015.7148511
- Singhal, N., & Raina, J. (2011). Comparative analysis of AES and RC4 algorithms for better utilization. *International Journal of Computer Trends and Technology*, 2(6), 177-181.
- Stallings, W. (2014). *Cryptography and Network Security* (6 ed.). New Jersey, USA: Pearson Education Inc.
- Sutter, G. D., Deschamps, J. P., & Imana, J. L. (2013). Efficient Elliptic Curve Point Multiplication Using Digit-Serial Binary Field Operations. *IEEE Transactions on Industrial Electronics*, 60(1), 217-225. doi:10.1109/TIE.2012.2186104
- Targhetta, A. D., Owen, D. E., Israel, F. L., & Gratz, P. V. (2015, 18-21 Oct. 2015). *Energy-efficient implementations of GF(p) and GF(2m) elliptic curve cryptography*. Paper presented at the 33rd IEEE International Conference on Computer Design (ICCD), 2015. doi:10.1109/ICCD.2015.7357184
- Tawalbeh, L., Mowafi, M., & Aljoby, W. (2013). Use of elliptic curve cryptography for multimedia encryption. *IET Information Security*, 7(2), 67-74. doi:10.1049/iet-ifs.2012.0147
- Thakur, J., & Kumar, N. (2011). DES, AES and Blowfish: Symmetric key cryptography algorithms simulation based performance analysis. *International journal of emerging technology and advanced engineering*, 1(2), 6-12.

- Thirananant, N., Kang, Y. J., Kim, T., Jang, W., Park, S., & Lee, H. (2014, 19-21 Dec. 2014). *A Design of Elliptic Curve Cryptography-Based Authentication Using QR Code*. Paper presented at the IEEE 17th International Conference on Computational Science and Engineering (CSE), 2014. doi:10.1109/CSE.2014.135
- Ustimenko, V. (2007). On Graph-Based Cryptography and Symbolic Computations. *Serdica Journal of Computing*, 1(2), 131-156.
- Ustimenko, V. (2014, 7-10 Sept. 2014). *On multivariate cryptosystems based on maps with logarithmically invertible decomposition corresponding to walk on graph*. Paper presented at the Federated Conference on Computer Science and Information Systems (FedCSIS), 2014. doi:10.15439/2014F269
- Ustimenko, V., & Romańczuk, U. (2013). On extremal graph theory, explicit algebraic constructions of extremal graphs and corresponding Turing encryption machines *Artificial Intelligence, Evolutionary Computing and Metaheuristics* (pp. 257-285). Heidelberg, Germany: Springer. doi:10.1007/978-3-642-29694-9\_11
- Vigila, S., & Muneeswaran, K. (2009, 13-15 Dec. 2009). *Implementation of text based cryptosystem using Elliptic Curve Cryptography*. Paper presented at the First International Conference on Advanced Computing, 2009. ICAC 2009. doi:10.1109/ICADVC.2009.5378025
- Wang, R. Z., & Hsu, S. F. (2011). Tagged Visual Cryptography. *IEEE Signal Processing Letters*, 18(11), 627-630. doi:10.1109/LSP.2011.2166543
- Wang, X., Pei, Q., & Li, H. (2014). A Lossless Tagged Visual Cryptography Scheme. *IEEE Signal Processing Letters*, 21(7), 853-856. doi:10.1109/LSP.2014.2317706
- Wu, H. (2008). The stream cipher HC-128 *New stream cipher designs* (pp. 39-47): Springer. doi:10.1007/978-3-540-68351-3\_4
- Wu, X., & Sun, W. (2013). Generalized Random Grid and Its Applications in Visual Cryptography. *IEEE Transactions on Information Forensics and Security*, 8(9), 1541-1553. doi:10.1109/TIFS.2013.2274955
- Xia, L. (2012, 24-27 June 2012). *The application of Elliptic Curve Cryptography in Electronic Commerce*. Paper presented at the IEEE Symposium on Electrical & Electronics Engineering (EEESYM), 2012. doi:10.1109/EEESym.2012.6258715
- Yan, S. Y. (2008). *Cryptanalytic attacks on RSA*. New York, USA: Springer US. doi:10.1007/978-0-387-48742-7
- Ye, L., & Liu, F. (2011, 24-26 Dec. 2011). *Overview of scalar multiplication in elliptic curve cryptography*. Paper presented at the International Conference on Computer

Science and Network Technology (ICCSNT), 2011. doi:10.1109/ICCSNT.2011.6182515

Zhi, Z., Arce, G. R., & Di Crescenzo, G. (2006). Halftone visual cryptography. *IEEE Transactions on Image Processing*, 15(8), 2441-2453. doi:10.1109/TIP.2006.875249

## Appendix A: Glossary

<i>Term</i>	<i>Definition</i>
<i>Adjacency list</i>	A method of implementing graphs in computer code. Enumerates all nodes which are connected by an edge.
<i>Adjacency matrix</i>	A method of implementing graphs in computer code. A 2-dimensional matrix where there is a 1 entry if the two nodes are connected by an edge. Else, there is a 0 entry.
<i>AES</i>	Advanced Encryption Standard. Introduced to replace DES in 2001, through the National Institute of Standards and Technology (NIST). Based on the Feistel cipher structure. Allows for key sizes of 128, 192, or 256-bits. The current standard for symmetric block ciphers.
<i>Alphabetic Cipher</i>	Operates on characters from a given alphabet.
<i>Artificial Intelligence</i>	A technological theory and current research area into the possibility of developing an artificial mind which has the ability to mimic human consciousness.
<i>ASCII</i>	A character set utilized in most HTML sites. Based around UTF-8 encoding, and encodes text-based characters into unique bytes from 0-255. Stands for American Standard Code for Information Interchange.
<i>Asymmetric Encryption</i>	Also termed <i>Public-key encryption</i> . These schemes have two keys, a <i>private</i> or <i>secret</i> key, and a <i>public</i> key. The public key is used to encrypt the data but cannot be used for decryption. The private key is used to decrypt the data. These systems form the basis of many Internet technologies, and can also be used to create digital signatures and certificates.
<i>Avalanche effect</i>	The resulting change in the ciphertext after altering a single bit of the plaintext. A high level of change in the ciphertext is a desirable security feature of modern ciphers.
<i>Bernoulli trials</i>	A series of trials each of which will either succeed or fail. The probability of $n$ successes in $k$ trials can be calculated through the binomial probability formula.

<i>Binary codewords</i>	Used in error-correcting codes as introduced by Richard Hamming (1950). Binary strings of a set length with a particular Hamming weight that allows the system to detect errors in transmission. See also <i>Hamming distance</i> , <i>Error-correcting codes</i> .
<i>Binomial probability</i>	An experiment with only two outcomes: success or failure. See also <i>Bernoulli trials</i> .
<i>Bit-flipping attack</i>	An attack wherein the adversary alters the ciphertext to give a predictable alteration in the resulting plaintext.
<i>Block cipher</i>	A cryptographic method which operates on plaintext data block-by-block, usually with a block size of 64 to 128-bits. See <i>AES</i> .
<i>Brute force attack</i>	A simple attack wherein all possible keys are attempted. On average, a brute force attack requires that one half of all possible keys be attempted.
<i>Caesar cipher</i>	The earliest known cipher. A substitution scheme attributed to Julius Caesar, in which a plaintext message was encrypted by replacing each character with the letter 3 places to the right. See <i>symmetric cipher</i> and <i>substitution</i> .
<i>Cayley table</i>	A 2-dimensional <i>matrix</i> which gives the result of the binary operation on each combination of elements in a set.
<i>CBC mode</i>	An encryption mode in AES. Cipher Block Chaining mode. Each new block of plaintext is combined with the previous ciphertext block through an XOR operation prior to encryption.
<i>CFB mode</i>	An encryption mode in AES. Cipher Feedback mode. Turns AES into a stream cipher, but is very similar in functionality to CBC.
<i>Chosen plaintext attack</i>	A cryptanalytic attack wherein a malicious adversary chooses the plaintext to be encrypted to take advantage of particular tendencies or features of a cryptographic system.
<i>Ciphertext</i>	The resulting output of an encryption algorithm.
<i>Common share RGVC</i>	A version of <i>Asymmetric encryption</i> from <i>Random Grid VC</i> which creates shares based on a single key share which becomes the key for the overall scheme.

<i>Computational complexity</i>	The difficulty of completing the required computations for cracking a particular cryptographic scheme, usually due to technological limitations.
<i>Contrast constraints</i>	The set delineation in <i>Visual Cryptography</i> between a pixel that is considered black and a pixel that is considered white. Defined by the percentage of subpixels in the recombined pixel that are black. See also <i>Pixel expansion</i> .
<i>Coordinate Matrix Encryption</i>	The proposed symmetric stream cipher which utilizes matrices and coordinates and blank padding spaces to encrypt data.
<i>Cryptanalysis</i>	The art of forcibly decoding ciphertext messages, either by exhaustive searches (see <i>Brute force attack</i> ) or by manipulation of known trends and tendencies within an encryption algorithm (see <i>Chosen plaintext attack</i> , <i>known plaintext attack</i> , <i>bit flipping attack</i> ).
<i>Decryption Algorithm</i>	The method by which the ciphertext is turned into plaintext.
<i>DES</i>	Data Encryption Standard. Developed in 1970s by Horst Feistel, and was the official standard from 1977 to 2001. Used a key size of 64 bits (56 bits for computation), which was proven insufficient for modern technologies. Superceded by <i>AES</i> .
<i>Dictionary attacks</i>	A simple attack in which a given dictionary of possible keys is tried exhaustively. A variation of the <i>brute-force attack</i> .
<i>Diffie-Hellman problem</i>	See <i>Discrete Logarithm Problem</i> .
<i>Digital certificate</i>	Electronic document used to verify online ownership. See also <i>Asymmetric encryption</i> , <i>digital signature</i> .
<i>Digital signature</i>	A method by which identity can be verified over the Internet. Uses asymmetric encryption to create a code which can be verified by anyone with access to the user's public key, but can only be created using a private key. See also <i>Asymmetric encryption</i> , <i>public key</i> , <i>secret key</i> , and <i>Digital certificate</i> .
<i>Digital watermarks</i>	The method of stamping a piece of digital material with a mark so as to prevent copyright infringement. Usually involves the use of <i>steganography</i> .

<i>Discrete Logarithm Problem</i>	The computation of logarithms in a finite field. No general method outside of quantum computing currently exists which allows this problem to be solved in polynomial time or less. The DLP is the <i>one-way function</i> utilized in the security of ECC. See also <i>Integer Factorization Problem</i> .
<i>Eavesdropping attack</i>	A network layer attack. Involves listening to all transmissions on a network in hopes of capturing sensitive information.
<i>ECB mode</i>	An encryption mode in AES. Electronic Codebook mode. Each block of plaintext is encrypted individually.
<i>ECRYPT</i>	The European Network of Excellence for Cryptography.
<i>Elliptic Curve Cryptography</i>	An asymmetric encryption system based on the calculation of affine points on elliptic curves over finite fields. Allows for the use of smaller key sizes than that of RSA. See <i>RSA, Asymmetric encryption</i> .
<i>Encryption algorithm</i>	The method of turning the plaintext into ciphertext.
<i>Error-correcting codes</i>	Introduced by Richard Hamming (1950). Allow for the transmission of data using codewords that are capable of detecting and correcting errors within the transmission. See also <i>Hamming distance, binary codewords</i> .
<i>eSTREAM Project</i>	Launched by ECRYPT, a project to standardize new symmetric stream ciphers for use in cryptographic protocols. Currently contains seven stream ciphers for use in either software or hardware. See <i>ECRYPT</i> and <i>Stream cipher</i> .
<i>Extended VC</i>	<i>Visual Cryptography</i> schemes which encode each share into a specific target image. Requires two separate <i>contrast constraints</i> .
<i>Feedback with carry shift register</i>	Extends the <i>Linear feedback shift register</i> and implements carry over arithmetic. Used in <i>Stream ciphers</i> .
<i>Feistel cipher</i>	Proposed by Horst Feistel. A cipher that alternates between permutations and substitutions.
<i>Finite field arithmetic</i>	Modular arithmetic. All operations are constrained within a set field, for example $2^8$ , or 256, meaning that no result of any operation will be outside the range [0,256].



<i>FMS Attack</i>	An attack against RC4 posited by Fluhrer, Mantin & Shamir (2001). The attack exploits a weakness in the construction of session keys. See <i>Rivest Cipher 4</i> .
<i>Frequency Analysis</i>	A statistical method of cryptanalysis, using the statistical properties of the ciphertext to determine the key.
<i>Galois field</i>	See <i>Finite field arithmetic</i> .
<i>Generalized RGVC</i>	A version of <i>Random Grid VC</i> which gives adjustable light transmission, or <i>contrast constraint</i> .
<i>Graph decomposition</i>	The act of deconstructing a graph into smaller sub-graphs without the loss of any information from the original graph.
<i>Graph-based EVCS</i>	<i>Extended VC</i> based on graphs. The scheme contains multiple subsets of authorized participants, each set of which are able to decode a particular secret. Each participant is designated as a node on the graph. If an edge exists between two nodes, then that pair share a given secret.
<i>Graphic-based ciphers</i>	Cryptographic technologies that base their design on topology and graph or group theory. See <i>Elliptic Curve Cryptography</i> or <i>Visual Cryptography</i> .
<i>Grover's algorithm</i>	A quantum algorithm for the computing of possible keys within a key space. Reduces the complexity of symmetric encryption systems from $O(N)$ to $O(N^{1/2})$ .
<i>Hadamard code</i>	An <i>Error-correcting code</i> derived from a <i>Hadamard matrix</i> . Capable of producing high-levels of error correction.
<i>Hadamard matrices</i>	A special family of matrices with all entries either +1 or -1. When the matrix H is multiplied against its inverse the result is the identity matrix multiplied by the scalar $n$ .
<i>Hamming distance</i>	Also known as the Hamming weight. The number of bits in the same position that differ between two binary codewords. The basis for the minimum distance of an error-correcting code. See also <i>Error-correcting codes, binary codewords</i> .
<i>HC-128</i>	A <i>stream cipher</i> currently included in the suite of ciphers resulting from the <i>eSTREAM project</i> . Developed by Hongjun Wu (2008).

<i>Incidence list</i>	A method of implementing graphs in computer code. Gives all edges that are adjacent to a given node $n$ .
<i>Incidence matrix</i>	A method of implementing graphs in computer code. A 2-dimensional matrix with all nodes down one side and all edges along the other. A 1 entry occurs when a node is adjacent to a given edge. Elsewhere, the entry is 0.
<i>Index calculus</i>	A method of computing discrete logarithms using probability and field arithmetic. See also <i>Discrete Logarithm Problem</i> .
<i>Initialisation Vector</i>	A sequence used to initialize the state of a cryptographic function.
<i>Integer Factorization Problem</i>	The currently unresolved problem of factorizing large numbers. In cryptography, the IFP is usually based around very large prime factors. Currently, no known polynomial time algorithm exists outside the realm of quantum computing. This problem is addressed in <i>Quantum computing</i> by <i>Shor's algorithm</i> . The IFP forms the basis of security in RSA. See also <i>RSA</i> , <i>Asymmetric encryption</i> , <i>Shor's algorithm</i> , <i>one-way function</i> .
<i>Internet of Things</i>	The vast and ever-expanding web of networked technologies, such as smart watches, cars, and appliances.
<i>Java Cryptography Architecture</i>	JCA. Inbuilt library of cryptographic functions available as of JDK 1.1.
<i>Java Development Kit</i>	JDK. The released platform version of Java for use by developers. The current JDK is JDK 8u91.
<i>Key matrix</i>	The alternative key structure utilized in the proposed CME scheme. Each key matrix contains all possible bit strings of a given length, and is exactly half-full.
<i>Keyspace</i>	The number of possible keys in the system.
<i>Keystream</i>	The key used in modern stream ciphers. Usually a random or pseudo-random string of bits. See <i>Stream cipher</i> and <i>Rivest Cipher 4</i> .
<i>Known plaintext attack</i>	A cryptanalytic attack wherein a malicious adversary has possession of a plaintext-ciphertext pair to analyze for clues as to the key.
<i>Linear feedback shift register</i>	A <i>linear register</i> wherein the input is the result from some linear function applied to the previous state. Used in <i>stream ciphers</i> .

<i>Man-in-the-middle attack</i>	An attack in which the communicated key or share is intercepted by a malicious adversary, who then creates their own key/share and sends it on to the intended recipient, thereby compromising all future communications between the parties.
<i>Matrix</i>	An array of elements.
<i>MQ problem</i>	The one-way function utilized in <i>multivariate cryptography</i> . The difficulty of solving many different quadratic equations over multiple fields using many variables. See also <i>One-way function</i> .
<i>Multithreading</i>	The use of multiple cores in computers to allow for parallel processing and increased computing power.
<i>Multivariate cryptography</i>	Cryptographic systems based around systems of multivariate equations.
<i>Non-linear feedback shift register</i>	Extends the <i>Linear feedback shift register</i> , and introduces non-linearity through some given function. As a result, it provides better protection against cryptanalysis. Used in <i>stream ciphers</i> .
<i>Non-singular mapping</i>	A given plaintext corresponds to multiple ciphertext outputs for a single key.
<i>NP-complete</i>	A problem for which the solution can be checked in polynomial time, but has no efficient method of discovering a solution. Referred to as nondeterministic polynomial time.
<i>OFB mode</i>	An encryption mode in AES. Output Feedback mode. This mode alters AES into a stream cipher, and uses keystreams for each encryption block.
<i>One-way function</i>	A trapdoor computation which is simple to execute in one direction, and difficult to reverse. The basis for asymmetric encryption. See <i>Asymmetric encryption</i> .
<i>Padding characters</i>	Blank coordinates used in the CME scheme to add confusion to the ciphertext output.
<i>Parity check</i>	Also known as the parity bit. A single bit of data added at the end of a sequence of bits to give the sequence an even number of 1 bits. If there is a corruption or error in the data transmitted, the parity check will result in an uneven number of 1 bits. See also <i>Error-correcting codes</i> .
<i>Perfect secrecy</i>	A cryptographic scheme that is theoretically secure. That is, a scheme whose security does not rest on its computational complexity, and is secure even against an adversary with unlimited computing power.

<i>Permutation</i>	The order in which elements in the plaintext occur is altered by some algorithmic means.
<i>Pixel expansion</i>	The phenomenon in <i>Visual Cryptography</i> by which the number of subpixels required to encode a particular pixel in the scheme increases with the number of nodes in the scheme. See also <i>Contrast constraints</i> .
<i>PKCS5 Padding</i>	A method by which the plaintext to be encrypted is padded using modular arithmetic, with mod 8.
<i>Plaintext</i>	The message/text/data that forms the input to an encryption algorithm.
<i>Pollard rho method</i>	Developed by John Pollard. There are versions for factorizing integers, and for calculating discrete logarithms.
<i>Post-quantum cryptography</i>	Cryptographic methods that are resistant to currently known quantum algorithms for cryptanalysis. See also <i>Shor's algorithm</i> and <i>Grover's algorithm</i> .
<i>Pseudorandom number</i>	A number that appears random but was generated through some algorithmic means.
<i>Pseudorandom number generator</i>	PRNG. An algorithm that returns a <i>pseudorandom number</i> . See also <i>Pseudorandom sequence</i> .
<i>Pseudorandom sequence</i>	A sequence that exhibits the properties of randomness, but is generated by some algorithmic means, is not truly random.
<i>Public-key encryption</i>	See <i>asymmetric encryption</i> .
<i>QR codes</i>	Quick-response codes. 2-dimensional matrix barcodes.
<i>Quantum computing</i>	The current technological theory and research area studying the application of quantum theories of superposition and entanglement to enable calculations to take place.
<i>Random Grid VC</i>	A method of <i>Visual Cryptography</i> which allows the control of <i>pixel expansion</i> . Uses a binary basis matrix to select whether a given pixel is black or white with equal probability. The first share is created by a random coin toss operation, and the second share is then created based on the first share.
<i>Reed Solomon codes</i>	Alternative to the Hamming codes introduced by Richard Hamming (1950). Reed-Solomon codes operate on bytes rather than bits. See also <i>Error-correcting codes</i> .

<i>Rivest Cipher 4 (RC4)</i>	Developed by Ron Rivest. The most widely used symmetric stream cipher. Has since been proven to be insecure. See <i>Stream cipher</i> .
<i>Round keys</i>	Utilized in many modern symmetric block ciphers. The key is expanded and split by some algorithmic means into a predetermined number of keys which are used in order in each round of the encryption and decryption processes.
<i>RSA</i>	Named for Rivest, Shamir and Adleman. An asymmetric encryption system which uses the <i>IFP</i> as its one-way function. Requires key lengths of 1024 bits or above for security. See also <i>Asymmetric encryption, Integer Factorization Problem</i> .
<i>Secret key</i>	The key used to decrypt an encrypted message in a public key/asymmetric system. In a symmetric or private key system, the secret key is used for both encryption and decryption.
<i>SecureRandom</i>	An inbuilt Java function which allows for the generation of a secure <i>pseudorandom number</i> .
<i>SET protocols</i>	Secure Electronic Transactions. Protocols implemented in e-commerce.
<i>Shor's algorithm</i>	A quantum algorithm for the computing of discrete logarithms and factorizing integers. Allows for the completion of such problems in polynomial time.
<i>Singular mapping</i>	A given plaintext maps to exactly one ciphertext output for a given key.
<i>Sparse matrices</i>	A matrix in which the majority of elements are zero.
<i>Spritz</i>	A stream cipher proposed by Rivest & Schuldt (2014) as an update to the now insecure RC4. See also <i>Rivest Cipher 4</i> .
<i>SSL</i>	Stands for Secure Sockets Layer. The predecessor to TLS. See <i>TLS</i> .
<i>Steganography</i>	The art of hiding messages. Rather than encrypting a secret message, the existence of the message is hidden. Often produces undesirable overheads in computation.
<i>Stream cipher</i>	A cryptographic method which operates on data either bit-by-bit or byte-by-byte. May be defined as a block cipher with a block size of smaller than 64 bits. Stream ciphers are wither word-oriented (operating byte-by-byte) or bit-oriented (operating bit-by-bit). See <i>Rivest Cipher 4</i> .

<i>Substitution</i>	An element within the message/data that makes up the plaintext is swapped for a ciphertext element by some algorithmic means.
<i>Symmetric Encryption</i>	A scheme in which both parties share a single secret key and algorithm, which is used to encrypt and decrypt messages/data. The security of the scheme rests on keeping the key secure.
<i>Systematic codes</i>	See <i>Error-correcting codes</i> .
<i>Timing attack</i>	A side-channel attack. Involves the timing of the execution of each stage of the algorithm.
<i>TLS</i>	Stands for Transport Layer Security. A protocol for enabling secure transmission at the network layer.
<i>Topology</i>	The spatial and geometric properties of given elements.
<i>Transposition</i>	See <i>Permutation</i> .
<i>Turing machines</i>	Automata that perform operations on sequential pieces of input based on a predefined set of rules.
<i>UTF-8</i>	Encodes all possible unicode characters in 8-bit code units. See also <i>ASCII</i> .
<i>Visual Cryptography</i>	A secret sharing scheme originally proposed by Naor and Shamir (1995). Uses a trading scheme made up of black and white pixels to create shares of an original secret image, which can then only be recreated when the authorized participants recombine their shares. VC makes use of matrices for share creation.

## Appendix B: Source Code

### B-1: GENERATION OF PSEUDO-RANDOM BINARY STRINGS

```
import java.io.Console;
class GenerateBinaryString {
    public static void main(String[] args) {
        Console cons = System.console();
        String length = cons.readLine("Enter length of random binary string : ");
        int stringLength = Integer.parseInt(length);
        int toss;
        String randString = "";
        for (int i = 0; i < stringLength; i++) {
            toss = (int)(Math.random()*2);
            if (toss == 0) { randString = randString+"0"; }
            else { randString = randString+"1"; }
        }
        System.out.println("Random generated string : "+randString);
    }
}
```

### B-2: AES AND RC4 CODE AND ANALYSIS PROGRAMS

#### B-2i: AES implementation

```
import java.security.MessageDigest;
import java.util.Arrays;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.spec.IvParameterSpec;
import java.io.Console;

import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.*;
```

```

import java.security.SecureRandom;

public class AES {
    static String plaintext;
    static String encryptionKey;
    static byte[] initVBytes;
    static SecureRandom pseudoRNG;
    static long megabyte = 1024L*1024L;
    static Console cons;

    public static void main(String [] args) {
        try {
            Runtime running = Runtime.getRuntime();
            running.gc();
            System.out.println("AES Encryption with Randomly Generated Key");
            cons = System.console();
            Boolean keepEncrypting = true;
            int encryptYN = 0;
            long setUpTime, setUpStart, setUpEnd;

            //Generate a random 128-bit key and initialisation vector.
            encryptionKey = "128";
            setUpStart = System.currentTimeMillis();
            KeyGenerator secretKey = KeyGenerator.getInstance("AES");
            secretKey.init(Integer.parseInt(encryptionKey));
            SecretKey randomAESKey = secretKey.generateKey();
            initVBytes = new byte[(Integer.parseInt(encryptionKey)/8)];
            pseudoRNG = new SecureRandom();
            pseudoRNG.nextBytes(initVBytes);
            setUpEnd = System.currentTimeMillis();
            setUpTime = setUpEnd-setUpStart;
            System.out.println("Set up complete, time taken: "+setUpTime+" ms");
            running.gc();
            long memoryInUse = running.totalMemory() - running.freeMemory();
            System.out.println("Total          memory          used:

```



```

"+((memoryInUse*1.0)/megabyte)+" MB");

//Encrypt the user entered data.
while (keepEncrypting) {
    plaintext = cons.readLine("Enter plaintext: ");
    int encryptTimes = Integer.parseInt(cons.readLine("Enter times to
encrypt the data: "));
    byte[] previous = new byte[0];
    System.out.println("plain: " + plaintext);
    for (int i = 0; i < encryptTimes; i++) {
        running.gc();
        long startTime, endTime, encryptTime, decryptTime;
        startTime = System.currentTimeMillis();
        byte[] cipher = encrypt(plaintext, randomAESKey);
        endTime = System.currentTimeMillis();
        encryptTime = endTime-startTime;
        running.gc();
        long    encryptMem    =    running.totalMemory()    -
running.freeMemory();

        //The following code measures the change in ciphertext
from the previous output to the current one.
        if (i > 0) {
            AvalancheEffect    avEffect    =    new
AvalancheEffect(cipher, previous);

            double diffBits = avEffect.calculateBits();
            double diffPos = avEffect.calculatePositions();
            System.out.print((i)+",");
            System.out.print((diffBits*100)+",");
            System.out.println((diffPos*100));
        }

        AnalyzeFrequencies    freqAnalysis    =    new
AnalyzeFrequencies(cipher);
        freqAnalysis.displayFrequenciesAES();

```

```

        running.gc();
        startTime = System.currentTimeMillis();
String decrypted = decrypt(cipher, randomAESKey);
        endTime = System.currentTimeMillis();
        decryptTime = endTime-startTime;
        running.gc();
        long    decryptMem    =    running.totalMemory()    -
running.freeMemory();

        previous = cipher;
        System.out.println("decrypt: " + decrypted);

        //Print the time taken to encrypt and decrypt the data.
        System.out.print(encryptTime+",");
        System.out.print(decryptTime+",");
        System.out.print((cipher.length*8));
        System.out.println();

        //The following code changes a single bit of one randomly
chosen byte of the plaintext, and is only used when measuring the avalanche effect.
        int            toChange            =            (int)
Math.floor(Math.random()*plaintext.length());
        char temp = plaintext.charAt(toChange);
        String tempStr = temp+"";
        MessageToBinary    toBin    =    new
MessageToBinary(tempStr);
        tempStr = toBin.getBinaryString();
        int            toChangeToo    =            (int)
Math.floor(Math.random()*tempStr.length());
        String changed = "";
        if (tempStr.charAt(toChangeToo) == '0') {
            changed            =
tempStr.substring(0,toChangeToo)+"1"+tempStr.substring(toChangeToo+1,tempStr.len
gth());
        } else {
            changed            =

```

```

tempStr.substring(0,toChangeToo)+"0"+tempStr.substring(toChangeToo+1,tempStr.length());
    }
    byte tmpByte = (byte)(Integer.parseInt(changed, 2));
    temp = (char)(tmpByte & 0xFF);
    plaintext =
plaintext.substring(0,toChange)+temp+plaintext.substring(toChange+1,
plaintext.length());

    //Analyze the frequencies of the ciphertext.
    freqAnalysis.displayFrequenciesAES();

    //The below code checks the overall memory used for the
processes of encryption and decryption.
    running.gc();
    memoryInUse = running.totalMemory() -
running.freeMemory();

    System.out.println((i+1)+", "+((encryptMem*1.0)/megabyte)+", "+((decryptMem
*1.0)/megabyte));
    }

    encryptYN = Integer.parseInt(cons.readLine("Do you want to
keep encrypting with this key? 1=Y, 2=N : "));
    if (encryptYN == 2) {
        keepEncrypting = false;
    }

}

} catch (Exception e) {
    e.printStackTrace();
}
}

```

```

public static byte[] encrypt(String plaintext, SecretKey secretKey) throws Exception {
    Cipher encryption = Cipher.getInstance("AES/CBC/PKCS5Padding");
    encryption.init(Cipher.ENCRYPT_MODE, secretKey, new
IvParameterSpec(initVBytes));
    return encryption.doFinal(plaintext.getBytes("UTF-8"));
}

```

```

public static String decrypt(byte[] ciphertext, SecretKey secretKey) throws Exception{
    Cipher decryption = Cipher.getInstance("AES/CBC/PKCS5Padding");
    decryption.init(Cipher.DECRYPT_MODE, secretKey, new
IvParameterSpec(initVBytes));
    return new String(decryption.doFinal(ciphertext),"UTF-8");
}
}

```

### **B-2ii: RC4 implementation**

```

import java.security.*;
import javax.crypto.*;
import java.io.Console;

class RC4 {
    static String plaintext;
    static String encryptionKey;
    static byte[] initVBytes;
    static SecureRandom pseudoRNG;
    static long megabyte = 1024L*1024L;
    static Console cons;
    static SecretKey randomRC4Key;
    static Cipher rC4Cipher;

    public static void main(String[] args) throws Exception {
        Runtime running = Runtime.getRuntime();
        System.out.println("RC4 Encryption with Random 128 bit key");
        cons = System.console();
        long setUpTime, setUpStart, setUpEnd;

```

```

//Initiate setup.
running.gc();
setUpStart = System.currentTimeMillis();
SetUp();
Boolean keepEncrypting = true;
int encryptYN = 0;
setUpEnd = System.currentTimeMillis();
setUpTime = setUpEnd-setUpStart;
System.out.println("Set up complete, time taken: "+setUpTime+" ms");
running.gc();
long memoryInUse = running.totalMemory() - running.freeMemory();
System.out.println("Total          memory          used:
"+((memoryInUse*1.0)/megabyte)+" MB");

while (keepEncrypting) {
    String plaintext = cons.readLine("Enter plaintext to encrypt: ");
    int encryptTimes = Integer.parseInt(cons.readLine("Enter number
of times to encrypt the data: "));
    byte[] previous = new byte[0];
    for (int i = 0; i < encryptTimes; i++) {
        running.gc();
        long startTime, endTime, encryptTime, decryptTime;
        startTime = System.currentTimeMillis();
        byte[] cipher = encrypt(plaintext);
        endTime = System.currentTimeMillis();
        encryptTime = endTime-startTime;
        running.gc();
        long    encryptMem    =    running.totalMemory()    -
running.freeMemory());

        //The following code measures the change in ciphertext
from the previous output to the current one.
        if (i > 0) {

```

```

        AvalancheEffect    avEffect    =    new
AvalancheEffect(cipher, previous);
        double diffBits = avEffect.calculateBits();
        double diffPos = avEffect.calculatePositions();
        System.out.print((i)+",");
        System.out.print((diffBits*100)+",");
        System.out.println((diffPos*100));
    }

        AnalyzeFrequencies    freqAnalysis    =    new
AnalyzeFrequencies(cipher);
        freqAnalysis.displayFrequenciesAES();

        running.gc();
        startTime = System.currentTimeMillis();
String decrypted = decrypt(cipher);
        endTime = System.currentTimeMillis();
        decryptTime = endTime-startTime;

        running.gc();
        long    decryptMem    =    running.totalMemory()    -
running.freeMemory();
        previous = cipher;
        System.out.println("decrypt: " + decrypted);

        //Print the time taken to encrypt and decrypt the data.
        System.out.print(encryptTime+",");
        System.out.print(decryptTime+",");
        System.out.print((cipher.length*8));
        System.out.println();

        //The following code changes a single bit of one randomly
chosen byte of the plaintext, and is only used when measuring the avalanche effect.
        int    toChange    =    (int)
Math.floor(Math.random()*plaintext.length());

```

```

        char temp = plaintext.charAt(toChange);
        String tempStr = temp+"";
        MessageToBinary toBin = new
MessageToBinary(tempStr);
        tempStr = toBin.getBinaryString();
        int toChangeToo = (int)
Math.floor(Math.random()*tempStr.length());
        String changed = "";
        if (tempStr.charAt(toChangeToo) == '0') {
            changed =
tempStr.substring(0,toChangeToo)+"1"+tempStr.substring(toChangeToo+1,tempStr.len
gth());
        } else {
            changed =
tempStr.substring(0,toChangeToo)+"0"+tempStr.substring(toChangeToo+1,tempStr.len
gth());
        }
        byte tmpByte = (byte)(Integer.parseInt(changed, 2));
        temp = (char)(tmpByte & 0xFF);
        plaintext =
plaintext.substring(0,toChange)+temp+plaintext.substring(toChange+1,
plaintext.length());

        //The below code checks the overall memory used for the
processes of encryption and decryption.
        running.gc();
        memoryInUse = running.totalMemory() -
running.freeMemory();

        System.out.println((i+1)+" "+((encryptMem*1.0)/megabyte)+" "+((decryptMem
*1.0)/megabyte));
    }

    encryptYN = Integer.parseInt(cons.readLine("Do you want to
keep encrypting with this key? 1=Y, 2=N : "));

```

```

        if (encryptYN == 2) {
            keepEncrypting = false;
        }
    }

}

public static void SetUp() throws Exception {
    SecureRandom initVector = new SecureRandom();
    KeyGenerator kGen = KeyGenerator.getInstance("RC4");
    kGen.init(128);
    randomRC4Key = kGen.generateKey();
    rC4Cipher = Cipher.getInstance("RC4");
}

public static byte[] encrypt(String plaintext) throws Exception {
    rC4Cipher.init(Cipher.ENCRYPT_MODE, randomRC4Key);
    byte[] ciphertext = rC4Cipher.doFinal(plaintext.getBytes());
    return ciphertext;
}

public static String decrypt(byte[] ciphertext) throws Exception {
    rC4Cipher.init(Cipher.DECRYPT_MODE, randomRC4Key);
    byte[] plaintext = rC4Cipher.doFinal(ciphertext);
    return new String(plaintext, "UTF-8");
}
}

```

### **B-2iii: AES/RC4 Frequency Analysis Program**

```

class AnalyzeFrequencies {
    private Frequency[] frequencies;
    private int totalValues;
    private int[] occurrences;
}

```



```

public static void main(String[] args) {

}

//Takes as input an array of bytes and measures the occurrences of each byte.
public AnalyzeFrequencies(byte[] ciphertext) {
    totalValues = ciphertext.length;
    frequencies = new Frequency[totalValues];
    for (int i = 0; i < totalValues; i++) {
        frequencies[i] = new Frequency("", 0, true, "none", "");
    }
    String temp, xString, yString;
    String actualVal = "none";
    String bitValue = "";
    Boolean exists = false;
    int noOfEntries = 0;
    int x,y;
    byte[] buffer = new byte[1];
    Boolean matrixEntryEmpty = true;
    for (int i = 0; i < ciphertext.length-1; i++) {
        temp = (new Integer(ciphertext[i])+"");
        for (int j = 0; j < totalValues; j++) {
            if (frequencies[j].valueEqual(temp)) {
                frequencies[j].updateOccurrences();
                exists = true;
                break;
            }
        }
        if (!exists) {
            noOfEntries++;
            frequencies[noOfEntries].setFreqValue(temp);
            frequencies[noOfEntries].updateOccurrences();
            frequencies[noOfEntries].setEmpty(false);
        }
    }
}

```

```

        exists = false;
    }

}

public void displayFrequenciesAES() {
    maxOccurBytes();
    for (int i = 1; i < occurances.length; i++) {
        System.out.print(occurances[i] + ",");
    }
    System.out.println();
}

public void maxOccurBytes() {
    int max = 0;
    for (int i = 0; i < totalValues; i++) {
        if (max < frequencies[i].getOccurances()) {
            max = frequencies[i].getOccurances();
        }
    }
    occurances = new int[max+1];
    int current = 0;
    for (int i = 0; i < totalValues; i++) {
        current = frequencies[i].getOccurances();
        occurances[current]++;
    }
}

}

class Frequency {
    private String freqValue;
    private int noOfOccurances;
    private Boolean isEmpty;
    private String actualValue;
}

```

```

private String bitValue;

public Frequency(String value, int occurrences, Boolean empty, String val, String
bits) {
    freqValue = value;
    noOfOccurrences = occurrences;
    isEmpty = empty;
    actualValue = val;
    bitValue = bits;
}

public void setBitValue(String bits) { bitValue = bits; }

public String getBitValue() { return bitValue; }

public void updateOccurrences() { noOfOccurrences++; }

public int getOccurrences() { return noOfOccurrences; }

public void setEmpty(Boolean empty) { isEmpty = empty; }

public Boolean isEmpty() { return isEmpty; }

public void setActualVal(String val) { actualValue = val; }

public String getActualVal() { return actualValue; }

public void setFreqValue(String value) { freqValue = value; }

public Boolean valueEqual(String toCheck) {
    if (toCheck.equals(freqValue)) {
        return true;
    } else {
        return false;
    }
}

```

```

    }

    public String getValue() { return freqValue; }
}

```

### **B-2iv: AES/RC4 Avalanche Effect Program**

```

import java.io.Console;

class AvalancheEffect {

    private static int bitsDiffer, positionsDiffer;
    private static byte[] bOne, bTwo;

    public AvalancheEffect(byte[] bytesOne, byte[] bytesTwo) {
        bOne = bytesOne;
        bTwo = bytesTwo;
    }

    public AvalancheEffect() {}

    //Calculate the total number of the same bytes occurring in the two ciphertexts.
    public double calculateBits() {
        int matches = 0;
        double percentMatch;
        for (int i = 0; i < bOne.length; i++) {
            for (int j = 0; j < bTwo.length; j++) {
                if (bOne[i] == bTwo[j]) {
                    matches++;
                    break;
                }
            }
        }
        percentMatch = ((matches*1.0)/bOne.length);
        return percentMatch;
    }
}

```

//Calculate the total number of bytes occurring in the same positions in the two ciphertexts.

```
public double calculatePositions() {
    int matches = 0;
    double percentMatch;
    for (int i = 0; i < bOne.length; i++) {
        if (bOne[i] == bTwo[i]) {
            matches++;
        }
    }
    percentMatch = ((matches*1.0)/bOne.length);
    return percentMatch;
}
}
```

### **B-2v: AES/RC4 Message to binary string conversion**

```
import java.math.BigInteger;
```

```
class MessageToBinary {
    private char[] charSet;
    private byte[] byteSet;
    private static String binaryString;

    public static void main(String[] args) throws Exception {
        MessageToBinary toBinary = new MessageToBinary(args[0]);
    }

    public MessageToBinary(String toConvert) throws Exception {
        byteSet = toConvert.getBytes("UTF-8");
        BigInteger binaryInt = new BigInteger(byteSet);
        binaryString = binaryInt.toString(2);
    }
}
```

```

public String getBinaryString () {
    return binaryString;
}

public String convertToCharacters(String binaryToConvert) {
    BigInteger toHex = new BigInteger(binaryToConvert,2);
    byte[] temp = toHex.toByteArray();
    String toReturn = "";
    try {
        toReturn = new String(temp, "UTF-8");
    } catch (Exception e) {}
    return toReturn;
}
}

```

### **B-3: ELLIPTIC CURVE IMPLEMENTATION**

#### **B-3i: Generate EC Key**

```

import java.security.*;
import java.security.spec.*;

class GenerateECCKey {

    private static PublicKey sharedKey;
    private static PrivateKey secretKey;

    public GenerateECCKey(String[] args) throws Exception {
        main(args);
    }

    public static void main(String[] args) throws Exception {
        KeyPairGenerator generate;
        generate = KeyPairGenerator.getInstance("EC", "SunEC");
        ECGenParameterSpec specs;

```

```

        specs = new ECGenParameterSpec("secp192r1");
        generate.initialize(specs);

        KeyPair pair = generate.genKeyPair();
        secretKey = pair.getPrivate();
        sharedKey = pair.getPublic();
    }

    public PrivateKey getPrivateKey() { return secretKey; }

    public PublicKey getPublicKey() { return sharedKey; }
}

```

### **B-3ii: Complete ECDH protocol**

```

import java.math.BigInteger;
import java.security.*;
import java.security.spec.*;
import javax.crypto.KeyAgreement;

class ECCKeyExchange {

    private static KeyAgreement keyAV, keyAU;
    private static BigInteger secretU, secretV;
    private static long megabyte = 1024L*1024L;

    public static void main(String[] args) throws Exception {
        Runtime running = Runtime.getRuntime();
        running.gc();
        long setupStart = System.currentTimeMillis();
        GenerateECCKey keyPairU = new GenerateECCKey(args);
        GenerateECCKey keyPairV = new GenerateECCKey(args);

        keyAU = KeyAgreement.getInstance("ECDH");
        keyAU.init(keyPairU.getPrivateKey());
    }
}

```

```

keyAU.doPhase(keyPairV.getPublicKey(), true);

keyAV = KeyAgreement.getInstance("ECDH");
keyAV.init(keyPairV.getPrivateKey());
keyAV.doPhase(keyPairU.getPublicKey(), true);

secretU = new BigInteger(1, keyAU.generateSecret());
secretV = new BigInteger(1, keyAV.generateSecret());

long setupEnd = System.currentTimeMillis();
long setupTotalTime = setupEnd-setupStart;
running.gc();
long memoryInUse = running.totalMemory() - running.freeMemory();
System.out.println("Secret      computed      by      U:      "+
(secretU.toString(16)).toUpperCase());
System.out.println("Secret      computed      by      V:      "+
(secretV.toString(16)).toUpperCase());
System.out.println("Total time for setup: "+setupTotalTime+" ms");
System.out.println("Total          memory          used:
"+((memoryInUse*1.0)/megabyte)+" MB");

    }
}

```

#### **B-4: VC IMPLEMENTATION**

##### **B-4i: 2-out-of-2 VC Encryption scheme**

```

import java.io.Console;

class VisualCryptoBinaryEncryption {

    public static int possibleSubPixelStates = 6;
    public static String randomShareOne, shareTwo;
    private static long megabyte = 1024L*1024L;

```



```

public static void main(String[] args) {
    Runtime running = Runtime.getRuntime();
    running.gc();
    Console cons = System.console();
    long setupStart = System.currentTimeMillis(); //start counter for setup
time.

    String[] subPixelsOne = subPixelStatesOne();
    String[] subPixelsTwo = subPixelStatesTwo();
    long setupEnd = System.currentTimeMillis(); //end counter for setup time.
    long setupTotal = setupEnd-setupStart;
    running.gc();
    long setupMem = running.totalMemory() - running.freeMemory();
    System.out.println("Set up complete. Time taken: "+setupTotal+" ms.");
    System.out.println("Total          memory          used:
"+((setupMem*1.0)/megabyte)+" MB");
    System.out.println();

    String plaintext = cons.readLine("Enter plaintext to encrypt into shares:
"); //get the binary plaintext string.
    String encryptTimes = cons.readLine("Enter number of times
encryption/decryption should be performed: "); //get the number of repetitions.
    int repetitions = Integer.parseInt(encryptTimes);
    String previousShareOne = "";
    long encryptMem, decryptMem;
    for (int i = 0; i < repetitions; i++) { //perform the repeated encryptions.
        running.gc();
        long encryptStart = System.currentTimeMillis();
        generateRandomShare(plaintext, subPixelsOne, subPixelsTwo);
//split the binary plaintext string into shares.
        long encryptEnd = System.currentTimeMillis();
        long encryptTotal = encryptEnd-encryptStart;
        running.gc();
        encryptMem = running.totalMemory() - running.freeMemory();
        System.out.println("Encryption #"+(i+1));
        System.out.println("Shares    generated.    Time    taken:

```

```

"+encryptTotal+" ms."); //display the shares.
    System.out.println("Share one: "+randomShareOne);
    System.out.println("Share two: "+shareTwo);
    System.out.println("Share length: "+shareTwo.length()+" bits.");
    if (i > 0) {
        AvalancheEffect avEffect = new AvalancheEffect();
        double percentSame =
avEffect.stringPos(previousShareOne, randomShareOne);
        System.out.println(i+", "+(percentSame*100));
    }
    running.gc();
    long decryptStart = System.currentTimeMillis();
    String combined = recombineShares(randomShareOne,
shareTwo); //recombine the shares into decrypted plaintext.
    long decryptEnd = System.currentTimeMillis();
    long decryptTotal = decryptEnd-decryptStart;
    decryptMem = running.totalMemory() - running.freeMemory();

    System.out.println((i+1)+", "+((encryptMem*1.0)/megabyte)+", "+((decryptMem
*1.0)/megabyte));
    System.out.println("Shares recombined. Time taken:
"+decryptTotal+" ms.");
    System.out.println("Decrypted plaintext: "+combined);
    Boolean matches = combined.equals(plaintext); //check combined
shares matches original plaintext.
    System.out.println("Decrypted plaintext matches original data:
"+matches);

    System.out.println();
    previousShareOne = randomShareOne;
    //The following code alters the inputted plaintext by exactly one
bit, allowing for measure of the avalanche effect.
    int randPos =
(int)Math.floor((Math.random()*(plaintext.length())));
    String temp = "";
    if (plaintext.charAt(randPos) == '0') {

```

```

        temp = plaintext.substring(0,
(randPos))+"1"+plaintext.substring((randPos+1), plaintext.length());
    } else {
        temp = plaintext.substring(0,
(randPos))+"0"+plaintext.substring((randPos+1), plaintext.length());
    }
    plaintext = temp;
}

}

public static String[] subPixelStatesOne() {
    String[] subPixelStringsOne = {"0101", "1010", "1100", "0011", "0110",
"1001"}; //generate the first array of possible subpixel states.
    return subPixelStringsOne;
}

public static String[] subPixelStatesTwo() {
    String[] subPixelStringsTwo = {"1010", "0101", "0011", "1100", "1001",
"0110"}; //generate the second array of possible subpixel states.
    return subPixelStringsTwo;
}

public static void generateRandomShare(String toSplit, String[]
subPixelStringsOne, String[] subPixelStringsTwo) {
    randomShareOne = "";
    shareTwo = "";
    int randomSubPixel;
    char currentPixel;
    for (int i = 0; i < toSplit.length(); i++) {
        randomSubPixel =
(int)Math.floor(Math.random()*possibleSubPixelStates); //pick a random subpixel state.
        currentPixel = toSplit.charAt(i);
        randomShareOne = randomShareOne +
subPixelStringsOne[randomSubPixel]; //assign the random subpixel state to the first

```

share.

```
        if (currentPixel == '1') {                                //if the pixel equals
1, assign the opposite subpixel state to the second share.
                shareTwo = shareTwo +
subPixelStringsTwo[randomSubPixel];
        } else {                                                //if
the pixel equals 0, assign the same subpixel state to the second share.
                shareTwo = shareTwo +
subPixelStringsOne[randomSubPixel];
        }
    }
}
```

```
public static String recombineShares(String sOne, String sTwo) {
    String combined = "";
    String tempOne, tempTwo;
    for (int i = 0; i < sOne.length()-3; i+=4) {
        tempOne = sOne.substring(i,i+4);
        tempTwo = sTwo.substring(i,i+4);
        if (tempOne.equals(tempTwo)) {
            combined = combined+"0";
        } else {
            combined = combined+"1";
        }
    }
    return combined;
}
}
```

#### **B-4ii: VC Avalanche effect**

```
import java.io.Console;
```

```
class AvalancheEffect {
```

```
    private static int bitsDiffer, positionsDiffer;
```

```

private static int[] bOne, bTwo;
private static byte[] byOne, byTwo;

public AvalancheEffect() {}

public static double stringPos(String one, String two) {
    int matches = 0;
    double percentMatch;
    for (int i = 0; i < one.length(); i++) {
        if (one.charAt(i) == two.charAt(i)) {
            matches++;
        }
    }
    percentMatch = ((matches*1.0)/one.length());
    return percentMatch;
}
}

```

## **B-5: CME BYTE IMPLEMENTATION**

### **B-5i: CME Byte setup and ByteCE classes**

```

import java.util.*;
import java.io.*;
import java.io.PrintWriter;
import java.util.Arrays;
import java.io.Console;
import java.lang.Math;
import java.math.BigInteger;

class SetUpByteCE {

    public static int totalStrings = 0;
    private static ByteCE[][] matrix;
    public static ByteCE[] bitStrings;

```

```

public static ByteCE[] blankEntries;
public static int stringLength = 8;
public static int totalLocations;
public static int numberOfBlanks;

public SetupByteCE(String[] args) {
    main(args);
}

public static void main(String[] args) {
    long startTime = System.currentTimeMillis(); //start stopwatch
    Console cons = System.console();
    if (cons == null) {
        System.err.println("No console available.");
        System.exit(1);
    }
    PrintWriter consOut = cons.writer();
    bitStrings = generateBitStrings(stringLength); //generate the array of all
possible bit strings of length n.

    matrix = new ByteCE[totalStrings][totalStrings]; //generate the coordinate
n^4 matrix.
    int numberPerString = (int)
(Math.pow(2,(2*stringLength))/(2*Math.pow(2,stringLength)));
    numberOfBlanks = totalStrings;
    System.out.println("Number of occupied spaces:
"+(totalStrings*numberPerString));
    System.out.println("Number of blank spaces:
"+(totalStrings*numberPerString));
    blankEntries = new ByteCE[totalStrings]; //generate the array of all blank
entries.
    try {
        String currentLine;
        int[] x, y;
        for (int i = 0; i < totalStrings; i++) {

```

```

        x = new int[totalLocations];
        y = new int[totalLocations];
        for (int j = 0; j < totalLocations; j++) {
            x[j] =
(int)Math.floor(Math.random()*totalStrings);
            y[j] =
(int)Math.floor(Math.random()*totalStrings);
            while (!(matrix[x[j]][y[j]] == null)) {
                x[j] =
(int)Math.floor(Math.random()*(totalStrings));
                y[j] =
(int)Math.floor(Math.random()*(totalStrings));
            }
            matrix[x[j]][y[j]] = bitStrings[i];
        }
        bitStrings[i].setLocationsX(x);
        bitStrings[i].setLocationsY(y);
    }
    int blanks = 0;
    int randomBlank = 0;
    for (int i = 0; i < totalStrings; i++){
        x = new int[totalLocations];
        y = new int[totalLocations];
        blankEntries[i] = new ByteCE(true);
        for (int j = 0; j < totalLocations; j++) {
            x[j] =
(int)Math.floor(Math.random()*totalStrings);
            y[j] =
(int)Math.floor(Math.random()*totalStrings);
            while (!(matrix[x[j]][y[j]] == null)) {
                x[j] =
(int)Math.floor(Math.random()*(totalStrings));
                y[j] =
(int)Math.floor(Math.random()*(totalStrings));
            }

```

```

        matrix[x[j]][y[j]] = blankEntries[i];
    }
    blankEntries[i].setLocationsX(x);
    blankEntries[i].setLocationsY(y);
}
consOut.println("Total          matrix          size:
["+totalStrings+", "+totalStrings+"]");

    long endTime = System.currentTimeMillis();
    long timeTaken = endTime-startTime;
} catch (Exception e) {
    consOut.println("Unknown exception occurred. Operation
terminated. Stack trace below.");
    e.printStackTrace(System.out);
}
}

public static ByteCE[] generateBitStrings(int stringLength) {
    int maxStrings = (int)Math.pow(2.0,((double)stringLength));
    totalStrings = maxStrings;
    totalLocations          =          (int)
(Math.pow(2,(2*stringLength))/(2*Math.pow(2,stringLength)));
    ByteCE[] bitStrings = new ByteCE[(int)maxStrings];
    byte temp;
    int max = (int)maxStrings;
    int lengthDifference = 0;
    for (int i = 0; i < max; i++) {
        temp = (byte) i;
        bitStrings[i] = new ByteCE(temp, false);
    }
    System.out.println("Total strings: "+maxStrings);
    return bitStrings;
}

public ByteCE[][] getMatrix() {

```



```

        return matrix;
    }
}

class ByteCE {

    private byte bitValue;
    private Boolean isEmpty;
    private int locationX;
    private int locationY;
    public int[] locationsX;
    public int[] locationsY;

    public static void main(String[] args) {}

    public ByteCE(byte bitVal, Boolean empty) {
        bitValue = bitVal;
        isEmpty = empty;
    }

    public ByteCE(Boolean empty) { isEmpty = empty; }

    public Boolean entryEmpty() { return isEmpty; }

    public byte entryValue() { return bitValue; }

    public int getLocationX() { return locationX; }

    public void setLocationX(int x) { locationX = x; }

    public int getLocationY() { return locationY; }

    public void setLocationY(int y) { locationY = y; }

    public int[] getLocationsX() { return locationsX; }
}

```

```

public void setLocationsX(int[] x) { locationsX = x; }

public int[] getLocationsY() { return locationsY; }

public void setLocationsY(int[] y) { locationsY = y; }
}

```

### **B-5ii: CME Byte code**

```

import java.util.*;
import java.io.*;
import java.io.Console;
import java.lang.Math;

class CMEByteFixed {

    private static ByteCE[][] matrix;
    private static ByteCE[] blanks;
    private static ByteCE[] strings;
    private static int totalStrings, noOfBlanks;
    private static SetUpByteCE newMatrix;
    private static Console cons;
    private static String length;
    private static long megabyte = 1024L*1024L;
    private static int[] stringKey;

    public static void main(String[] args) throws Exception {
        //The following code completes the setup of a 256 by 256 key matrix
        containing all possible bytes.
        Runtime running = Runtime.getRuntime();
        running.gc();
        long setUpStart = System.currentTimeMillis();
        cons = System.console();
        String[] arg = {"8"}; //Send bit string length as argument to set up.
        newMatrix = new SetUpByteCE(arg); //Create the randomized new matrix
    }
}

```

set up.

```
matrix = newMatrix.getMatrix(); //Get the matrix pointer.
blanks = newMatrix.blankEntries; //Get the blank entries.
strings = newMatrix.bitStrings; //Get the array of bit strings.
totalStrings = newMatrix.totalStrings; //Get the total number of bit strings.
noOfBlanks = newMatrix.numberOfBlanks; //Get the total number of
blank entries.
//Assign the key locations for Exclusive-OR - always the first x coordinate
for each byte. List is the same for each key matrix.
stringKey = new int[totalStrings];          for (int i = 0; i < totalStrings;
i++) {
    stringKey[i] = strings[i].locationsX[0];
}
long setUpEnd = System.currentTimeMillis();
long setUpTotal = setUpEnd - setUpStart;
running.gc();
long memoryInUse = running.totalMemory() - running.freeMemory();
System.out.println("Total          memory          used:
"+((memoryInUse*1.0)/megabyte)+" MB");
System.out.println("Set up complete, time taken: "+setUpTotal+" ms");

//This code section repeats the encryption process until the user ends it.
Boolean continueEncrypting = true;
String continueEncYN;
int continueYN;
while (continueEncrypting) {
    performEncryptDecrypt();
    continueEncYN = cons.readLine("Encrypt more data? 1 = Y, 2 =
N : ");
    continueYN = Integer.parseInt(continueEncYN);
    if (continueYN == 1) {
        continueEncrypting = true;
    } else {
        continueEncrypting = false;
    }
}
```

```

    }

}

public static void performEncryptDecrypt() throws Exception {
    //Get data and run encryption/decryption.
    Runtime running = Runtime.getRuntime();
    String toEncrypt = cons.readLine("Enter data to encrypt: "); //get the
plaintext.

    int stringLength = toEncrypt.length(); //check plaintext length.
    System.out.println("Entry length: "+(stringLength*8));
    String repeatEncryptions = cons.readLine("How many times do you want
to encrypt and decrypt the data? : ");
    int repeats = Integer.parseInt(repeatEncryptions);
    int[] cipher, previous = new int[0];
    byte[] paddedVersion, plain, bitVersion;
    AnalyzeFrequencies freqOfCT;
    int intLength = stringLength;
    String convertedPT = toEncrypt;
    for (int i = 0; i < repeats; i++) { // complete the encryption/decryption
process as many times as required.

        //Encryption: Time taken & memory in use are measured. String is
converted to bytes.

        //When not measuring memory, gc() calls should be commented
out.

        running.gc();
        long startTime = System.currentTimeMillis();
        bitVersion = convertedPT.getBytes("UTF-8");
        cipher = encrypt(matrix, bitVersion, 8); //encrypt the plaintext.
        long endTime = System.currentTimeMillis();
        long encryptTime = endTime-startTime;
        running.gc();
        long    encryptMem    =    running.totalMemory()    -
running.freeMemory());

```

//Decryption: Time taken & memory in use are measured. Plaintext is converted back into a string.

//When not measuring memory, gc() calls should be commented out.

```
running.gc();
startTime = System.currentTimeMillis();
plain = decrypt(matrix, cipher); //decrypt the plaintext.
try {
    convertedPT = new String(plain, "UTF-8");
} catch (Exception e) {}
endTime = System.currentTimeMillis();
long decryptTime = endTime-startTime;
running.gc();
long    decryptMem    =    running.totalMemory()    -
running.freeMemory();
```

//This prints the time taken for encryption and decryption, and the resulting ciphertext length.

```
System.out.print((encryptTime)+" ");
System.out.print(decryptTime+" ");
System.out.print((cipher.length*8));
System.out.println();
```

//Print out the total memory used for encryption and decryption.

```
System.out.println((i+1)+" "+((encryptMem*1.0)/megabyte)+" "+((decryptMem*1.0)/megabyte));
```

//Analyze the frequencies of bytes occurring in the ciphertext.

```
freqOfCT = new AnalyzeFrequencies(cipher, matrix);
freqOfCT.displayFrequenciesAES();
```

//The following section measures the difference in bytes between the current and previous ciphertexts.

```

        if (i > 0) {
            AvalancheEffect avEffect = new AvalancheEffect(cipher,
previous);

            double diffBits = avEffect.calculateBits();
            double diffPos = avEffect.calculatePositions();
            System.out.print((i)+" ");
            System.out.print((diffBits*100)+" ");
            System.out.println((diffPos*100));
        }
        previous = cipher;

        //The following code changes a single bit of one randomly chosen
byte of the plaintext, and is only used when measuring the avalanche effect.
        int toChange = (int) Math.floor(Math.random()*plain.length);
        byte temp = plain[toChange];
        String tempStr = ((char) (temp & 0xFF))+"";
        MessageToBinary toBin = new MessageToBinary(tempStr);
        tempStr = toBin.getBinaryString();
        int toChangeToo = (int)
Math.floor(Math.random()*tempStr.length());
        String changed = "";
        if (tempStr.charAt(toChangeToo) == '0') {
            changed =
tempStr.substring(0,toChangeToo)+"1"+tempStr.substring(toChangeToo+1,tempStr.len
gth());
        } else {
            changed =
tempStr.substring(0,toChangeToo)+"0"+tempStr.substring(toChangeToo+1,tempStr.len
gth());
        }
        temp = (byte)(Integer.parseInt(changed, 2));
        plain[toChange] = temp;
        convertedPT = new String(plain,"UTF-8");
    }
    System.out.println("Original plaintext: " + convertedPT);

```

```

    }

    public static int[] encrypt(ByteCE[][] key, byte[] plaintext, int stringLength) {
        int coinToss, location, randomBlank, current, toXOR, currentX, currentY;
        int i = 0, j = 0, k = 0;
        String temp = "";
        int numberPerString = (int)
(Math.pow(2,(2*stringLength))/(2*Math.pow(2,stringLength))); //Get the total possible
locations for each character.
        ByteCE tempByte;
        byte currentByte;
        int blankPadding = (plaintext.length);
        int[] cipher = new int[blankPadding*4];
        while ((j < plaintext.length || k < blankPadding) && i < cipher.length) { //
Ciphertext length should be exactly (plaintext*4).
            coinToss = (int)Math.floor(Math.random()*2.0); //Randomly
distribute enciphered message characters among padding characters.

            //If coin results in heads, insert coordinates for message character.
            if (coinToss == 1 && j < plaintext.length) {
                location =
(int)Math.floor(Math.random()*numberPerString);
                //Exclusive-OR the plaintext character with the next
location of the key.

                toXOR = (stringKey[(j%totalStrings)]);
                current = (plaintext[j] ^ toXOR);
                if (current < 0) { current = current+255; } //If the integer of
a byte is negative, move its range into [0,255].
                cipher[i] = strings[current].locationsX[location];
                cipher[i+1] = strings[current].locationsY[location];
                j++;
                i+=2;
            } else if (k < blankPadding) { //If tails, add empty padding
coordinates.

                randomBlank =

```

```

(int)Math.floor(Math.random()*totalStrings);
        location =
(int)Math.floor(Math.random()*numberPerString);
        tempByte = blanks[randomBlank];
        cipher[i] = tempByte.locationsX[location];
        cipher[i+1] = tempByte.locationsY[location];
        k++;
        i+=2;
    }

}

return cipher;

}

public static byte[] decrypt(ByteCE[][] key, int[] ciphertext) {
    byte[] plaintext = new byte[ciphertext.length/4]; // Plaintext is exactly 1/4
the length of the ciphertext.
    int current = 0;
    int x, y, tempInt, toXOR;
    ByteCE temp;
    for (int i = 0; i < ciphertext.length-1 && current < plaintext.length; i=i+2)
    {
        //Decrypt ciphertext coordinates two at a time.
        x = ciphertext[i];
        y = ciphertext[i+1];
        temp = key[x][y];
        if (!(temp.entryEmpty())) {
            //Exclusive-OR the resulting character with the next
position in the key and add the result to the plaintext.
            toXOR = (stringKey[(current%totalStrings)]);
            tempInt = (temp.entryValue() ^ toXOR);
            plaintext[current] = (byte)tempInt;
            current++;
        }
    }
}

```



```

        }
        return plaintext;
    }
}

```

## **B-6: CME STRING IMPLEMENTATION**

### **B-6i: CME String setup and Entry classes**

```

import java.util.*;
import java.io.*;
import java.io.PrintWriter;
import java.util.Arrays;
import java.io.Console;
import java.lang.Math;

class SetUp {

    public static int totalStrings = 0;
    private static CoordinateEntry[][] matrix;
    public static CoordinateEntry[] bitStrings;
    public static CoordinateEntry[] blankEntries;
    public static int stringLength;
    public static int totalLocations;
    public static int numberOfBlanks;

    public SetUp(String[] args) {
        main(args);
    }

    public static void main(String[] args) {
        stringLength = 0;
        Console cons = System.console();

```

```

if (cons == null) {
    System.err.println("No console available.");
    System.exit(1);
}
PrintWriter consOut = cons.writer();
String length = args[0];
try {
    stringLength = Integer.parseInt(length); //turn length into an
integer.
} catch (Exception e) {
    consOut.println("Error. Enter Numbers only.");
}
bitStrings = generateBitStrings(stringLength); //generate the array of all
possible bit strings of length n.

```

```

String fileInput = "MatrixInput.txt";
String fileOutput = "MatrixOutput.txt";
BufferedReader br = null;
BufferedWriter bw = null;
matrix = new CoordinateEntry[totalStrings][totalStrings]; //generate the
coordinate n^4 matrix.

```

```

    int                numberPerString                =(int)
(Math.pow(2,(2*stringLength))/(2*Math.pow(2,stringLength)));
    numberOfBlanks = totalStrings;
    System.out.println("Number        of        occupied        spaces:
"+(totalStrings*numberPerString));
    System.out.println("Number        of        blank        spaces:
"+(numberOfBlanks*numberPerString));
    blankEntries = new CoordinateEntry[numberOfBlanks]; //generate the
array of all blank entries.

```

```

try {
    String currentLine;
    int[] x,y;
    for (int i = 0; i < totalStrings; i++) {
        x = new int[totalLocations];

```

```

        y = new int[totalLocations];
        for (int j = 0; j < totalLocations; j++) {
            x[j] =
(int)Math.floor(Math.random()*totalStrings);
            y[j] =
(int)Math.floor(Math.random()*totalStrings);
            while (!(matrix[x[j]][y[j]] == null)) {
                x[j] =
(int)Math.floor(Math.random()*(totalStrings));
                y[j] =
(int)Math.floor(Math.random()*(totalStrings));
            }
            matrix[x[j]][y[j]] = bitStrings[i];
        }
        bitStrings[i].setLocationsX(x);
        bitStrings[i].setLocationsY(y);
    }
    int blanks = 0;
    for (int i = 0; i < totalStrings; i++){
        x = new int[totalLocations];
        y = new int[totalLocations];
        blankEntries[i] = new CoordinateEntry("", true);
        for (int j = 0; j < totalLocations; j++) {
            x[j] =
(int)Math.floor(Math.random()*totalStrings);
            y[j] =
(int)Math.floor(Math.random()*totalStrings);
            while (!(matrix[x[j]][y[j]] == null)) {
                x[j] =
(int)Math.floor(Math.random()*(totalStrings));
                y[j] =
(int)Math.floor(Math.random()*(totalStrings));
            }
            matrix[x[j]][y[j]] = blankEntries[i];
        }
    }

```

```

        blankEntries[i].setLocationsX(x);
        blankEntries[i].setLocationsY(y);
    }
    consOut.println("Total          matrix          size:
["+totalStrings+", "+totalStrings+"]");
    } catch (Exception e) {
        consOut.println("Unknown exception occurred. Operation
terminated. Stack trace below.");
        e.printStackTrace(System.out);
    }
}

public static CoordinateEntry[] generateBitStrings(int stringLength) {
    int maxStrings = (int)Math.pow(2.0,((double)stringLength));
    totalStrings = maxStrings;
    totalLocations = (int)
(Math.pow(2,(2*stringLength))/(2*Math.pow(2,stringLength)));
    if (stringLength >= 63) {
        System.out.println("Error. Length must be less than 63.");
    }
    CoordinateEntry[] bitStrings = new CoordinateEntry[(int)maxStrings];
    String temp;
    int max = (int)maxStrings;
    int lengthDifference = 0;
    for (int i = 0; i < max; i++) {
        temp = Integer.toBinaryString(i);
        if (temp.length() != stringLength) {
            lengthDifference = stringLength-temp.length();
            for (int j = 0; j < lengthDifference; j++){
                temp = "0"+temp;
            }
        }
        bitStrings[i] = new CoordinateEntry(temp, false);
        temp = "";
    }
}

```

```

    }
    System.out.println("Total strings: "+maxStrings);
    return bitStrings;
}

public CoordinateEntry[][] getMatrix() {
    return matrix;
}

public void displayMatrix() {
    for (int i = 0; i < bitStrings.length; i++) {
        for (int j = 0; j < bitStrings.length; j++) {
            System.out.print("[");
            if (matrix[i][j].entryEmpty()) {
                for (int k = 0; k < stringLength; k++) {
                    System.out.print("-");
                }
            } else {
                System.out.print(matrix[i][j].entryValue());
            }
            System.out.print("]");
        }
        System.out.println();
    }
}
}

```

```

class CoordinateEntry {

    private String bitValue;
    private Boolean isEmpty;
    private int locationX;
    private int locationY;
    public int[] locationsX;
    public int[] locationsY;
}

```

```

public static void main(String[] args) {}

public CoordinateEntry(String bitVal, Boolean empty) {
    bitValue = new String(bitVal+ "");
    isEmpty = empty;
}

public Boolean entryEmpty() { return isEmpty; }

public String entryValue() { return bitValue; }

public int getLocationX() { return locationX; }

public void setLocationX(int x) { locationX = x; }

public int getLocationY() { return locationY; }

public void setLocationY(int y) { locationY = y; }

public int[] getLocationsX() { return locationsX; }

public void setLocationsX(int[] x) { locationsX = x; }

public int[] getLocationsY() { return locationsY; }

public void setLocationsY(int[] y) { locationsY = y; }
}

```

**B-6ii: CME string code**

```

import java.util.*;
import java.io.*;
import java.io.PrintWriter;
import java.util.Arrays;
import java.io.Console;

```

```

import java.lang.Math;

class CoordinateEncryptionAlgorithm {

    private static CoordinateEntry[][] matrix;
    private static CoordinateEntry[] blanks;
    private static CoordinateEntry[] strings;
    private static int totalStrings;
    private static SetUp newMatrix;
    private static int noOfBlanks;
    private static Console cons;
    private static String length, xorString;
    private static long megabyte = 1024L*1024L;
    private static byte[] forXOR;

    public static void main(String[] args) throws Exception {
        Runtime running = Runtime.getRuntime();
        running.gc();
        long setupStart = System.currentTimeMillis();
        cons = System.console();
        length = "4";
        String[] arg = {length}; //Send bit string length as argument to set up.
        newMatrix = new SetUp(arg); //Create the randomized new matrix set up.
        matrix = newMatrix.getMatrix(); //Get the matrix pointer.
        blanks = newMatrix.blankEntries; //Get the blank entries.
        strings = newMatrix.bitStrings; //Get the array of bit strings.
        totalStrings = newMatrix.totalStrings; //Get the total number of bit strings.
        noOfBlanks = newMatrix.numberOfBlanks; //Get the total number of
blank entries.

        forXOR = new byte[totalStrings];
        int temp = 0;
        for (int i = 0; i < totalStrings; i++) {
            temp = strings[i].locationsX[0];
            forXOR[i] = (byte)temp;
        }
    }
}

```

```

        MessageToBinary bin = new MessageToBinary(new
String(forXOR,"UTF-8"));
        xorString = bin.getBinaryString();
        double divisor =
((double)xorString.length()/((double)newMatrix.stringLength));
        if (Math.floor(divisor) != divisor) {
            xorString = padPlaintext(xorString, divisor,
Integer.parseInt(length));
        }
        long setupEnd = System.currentTimeMillis();
        long setupTotal = setupEnd-setupStart;
        newMatrix.displayMatrix();
        running.gc();
        long setupMem = running.totalMemory() - running.freeMemory();
        System.out.println("Set up complete, time taken: "+setupTotal+" ms.");
        System.out.println("Total memory used:
"+((setupMem*1.0)/megabyte)+" MB");
        Boolean continueEncrypting = true;
        String continueEncYN;
        int continueYN;
        while (continueEncrypting) { //Continue encrypting & decrypting until the
user ends the process.
            performEncryptDecrypt();
            continueEncYN = cons.readLine("Encrypt more data? 1 = Y, 2 =
N : ");
            continueYN = Integer.parseInt(continueEncYN);
            if (continueYN == 1) {
                continueEncrypting = true;
            } else {
                continueEncrypting = false;
            }
        }
    }
}

```



```

public static void performEncryptDecrypt() throws Exception {
    String toEncrypt = cons.readLine("Enter data to encrypt: "); //get the
plaintext.

    String original = toEncrypt;
    Boolean isAlpha;
    String alphaYN = cons.readLine("Is the text in binary format? Y/N: ");
    alphaYN = alphaYN.toUpperCase();
    MessageToBinary toBin = new MessageToBinary();
    if (alphaYN.equals("N")){
        toBin = new MessageToBinary(toEncrypt);
        toEncrypt = toBin.getBinaryString();
        isAlpha = true;
    }
    else { isAlpha = false; }
    int stringLength = toEncrypt.length(); //check plaintext length.
    System.out.println("Entry length: "+stringLength);
    double                divisor                =
((double)stringLength/((double)newMatrix.stringLength)); //check if plaintext requires
padding.

    String repeatEncryptions = cons.readLine("How many times do you want
to encrypt and decrypt the data? : ");
    int repeats = Integer.parseInt(repeatEncryptions);
    System.out.println("Divisor: "+divisor);
    String cipher, paddedVersion;
    String plain = "";
    Boolean padded;
    AnalyzeFrequencies freqOfCT;
    long padStart = System.currentTimeMillis();
    long encryptMem, decryptMem;
    Runtime running = Runtime.getRuntime();
    if (Math.floor(divisor) == divisor) {
        System.out.println("Entry does not require padding.");
        padded = false;
        paddedVersion = toEncrypt;
    } else {

```

```

        paddedVersion    =    padPlaintext(toEncrypt,    divisor,
Integer.parseInt(length));
        padded = true;
    }
    long padEnd = System.currentTimeMillis();
    long padTotal = padEnd-padStart;
    int intLength = Integer.parseInt(length);
    String converted = paddedVersion;
    String previous = "";
    for (int i = 0; i < repeats; i++) { // complete the encryption/decryption
process as many times as required.
        //Encryption: Time taken & memory in use are measured.
        //When not measuring memory, gc() calls should be commented
out.

        running.gc();
        long startTime = System.currentTimeMillis();
        paddedVersion    =    toBin.xor(converted,    xorString,
Integer.parseInt(length));
        cipher = encrypt(matrix, paddedVersion, intLength, intLength);
//encrypt the plaintext.
        long endTime = System.currentTimeMillis();
        long encryptTime = endTime-startTime;
        running.gc();
        encryptMem = running.totalMemory() - running.freeMemory();

        //Decryption: Time taken & memory in use are measured.
        //When not measuring memory, gc() calls should be commented
out.

        running.gc();
        startTime = System.currentTimeMillis();
        plain = decrypt(matrix, cipher, intLength); //decrypt the plaintext.
        converted = toBin.xor(plain, xorString, Integer.parseInt(length));
        endTime = System.currentTimeMillis();
        long decryptTime = endTime-startTime;
        running.gc();

```

```

System.out.println("Plaintext:");
System.out.println(plain);
System.out.println("Ciphertext:");
System.out.println(cipher);
decryptMem = running.totalMemory() - running.freeMemory();

//This prints the time taken and memory in use for encryption &
decryption.
System.out.println((encryptTime+padTotal)+"
"+(decryptTime+padTotal)+" "+cipher.length());

System.out.println((i+1)+", "+((encryptMem*1.0)/megabyte)+" "+((decryptMem
*1.0)/megabyte));

//The following measures the frequencies of the characters in the
ciphertext.
freqOfCT = new AnalyzeFrequencies(cipher, intLength, matrix);
freqOfCT.displayFrequencies();

//This section measures the avalanche effect of the algorithm,
comparing the current and previous ciphertexts.
if (i > 0) {
    AvalancheEffect avEffect = new AvalancheEffect();
    double percentSame = avEffect.stringPos(cipher,
previous);

    System.out.println(i+" "+(percentSame*100));
}
previous = cipher;

//This section changes the plaintext by exactly one bit. It is only
used when measuring the avalanche effect.
int randPos = (int)Math.floor((Math.random()*(plain.length())));
if (plain.charAt(randPos) == '0') {
    toEncrypt = plain.substring(0,
(randPos))+ "1"+plain.substring((randPos+1), plain.length());

```

```

        } else {
            toEncrypt = plain.substring(0,
(randPos))+"0"+plain.substring((randPos+1), plain.length());
        }
    }
    plain = toBin.xor(plain, xorString, Integer.parseInt(length));
    if (padded) {
        plain = removePadding(plain, divisor, Integer.parseInt(length));
    }
    if (isAlpha) {
        plain = toBin.convertToCharacters(plain);
    }
    String convertedPT = "";
    System.out.println("Decoded binary string matches original input:
"+(plain.equals(original)));
}

```

```

    public static String encrypt(CoordinateEntry[][] key, String plaintext, int
stringLength, int length) {
        String ciphertext = "";
        int coinToss = 0;
        String temp = "";
        int location, lengthDifference, randomBlank, blankX, blankY;
        String xBit, yBit;
        int i = 0;
        int j = 0;
        int blankPadding = plaintext.length();
        while (i < plaintext.length() || j < blankPadding) {
            coinToss = (int) Math.floor(Math.random()*2.0);
            if (coinToss == 0 && i < plaintext.length()) {
                temp = temp + plaintext.substring(i, i+length); //get the
next n bits of the string.
                location = Integer.parseInt(temp, 2); //find the int
equivalent of the bit string
                if (location < 0) { location+=255; }

```

```

        int          chosenLocation          =          (int)
Math.floor(Math.random()*newMatrix.totalLocations);
        xBit          =
Integer.toBinaryString(strings[location].locationsX[chosenLocation]);
        if (xBit.length() != stringLength) { //make sure the bit
string for location x is nbits long.
                lengthDifference = stringLength-xBit.length();
                for (int k = 0; k < lengthDifference; k++){
                        xBit = "0"+xBit;
                }
        }
        yBit          =
Integer.toBinaryString(strings[location].locationsY[chosenLocation]);
        if (yBit.length() != stringLength) { //make sure the bit
string for location y is nbits long.
                lengthDifference = stringLength-yBit.length();
                for (int k = 0; k < lengthDifference; k++){
                        yBit = "0"+yBit;
                }
        }
        ciphertext = ciphertext+xBit+yBit; //update ciphertext with
new piece of string, bit strings for location x & y.
        temp = ""; //clear temp for next bit section.
        i = i+length; //move ahead to next bit section.
        location = 0;
    } else if (j < blankPadding) {
        randomBlank          =
(int)Math.abs(Math.random()*noOfBlanks);
        int          chosenLocation          =          (int)
Math.floor(Math.random()*newMatrix.totalLocations);
        blankX          =
blanks[randomBlank].locationsX[chosenLocation];
        blankY          =
blanks[randomBlank].locationsY[chosenLocation];
        xBit = Integer.toBinaryString(blankX);

```

```

        if (xBit.length() != stringLength) { //make sure the bit
string for location x is nbits long.
            lengthDifference = stringLength-xBit.length();
            for (int k = 0; k < lengthDifference; k++){
                xBit = "0"+xBit;
            }
        }
        yBit = Integer.toBinaryString(blankY);
        if (yBit.length() != stringLength) { //make sure the bit
string for location y is nbits long.
            lengthDifference = stringLength-yBit.length();
            for (int k = 0; k < lengthDifference; k++){
                yBit = "0"+yBit;
            }
        }
        ciphertext = ciphertext+xBit+yBit; //update ciphertext with
new piece of string, bit strings for location x & y.
        j=j+length;
    }
}

return ciphertext;
}

```

```

public static String decrypt(CoordinateEntry[][] key, String ciphertext, int
stringLength) {
    String plaintext = "";
    int i = 0;
    int x,y,j;
    String xBit, yBit;
    String temp = "";
    while (i < ciphertext.length()) {
        temp = ciphertext.substring(i,i+stringLength);
        i=i+stringLength;//update i for next location.
    }
}

```

```

        x = Integer.parseInt(temp, 2); //get integer value for binary string.
        temp = ""; //clear temp for next location.
        temp = ciphertext.substring(i,i+stringLength);
        i = i+stringLength; //update i for next location.
        y = Integer.parseInt(temp, 2); //get integer value for binary string.
        temp = ""; //clear temp for next location.
        if (!matrix[x][y].entryEmpty()) { //test if entry is padding or
message
                plaintext = plaintext+matrix[x][y].entryValue(); //if
message, add value to cipher text.
                }
        }

        return plaintext;

}

```

```

public static String padPlaintext(String plaintext, double divisor, int length) {
    int padding = (int) (Math.ceil(divisor)*length);
    padding = padding-plaintext.length();
    for (int i = 0; i < padding; i++) {
        plaintext = plaintext + "0";
    }
    return plaintext;
}

```

```

public static String removePadding(String ciphertext, double divisor, int length)
{
    double padding = (divisor*length);
    int paddingToRemove = (int) ((Math.ceil(divisor)*length)-padding);
    System.out.println("Amount of padding to remove:
"+paddingToRemove);
    String temp = "";
    for (int i = 0; i < (ciphertext.length()-paddingToRemove); i++) {

```

```

        temp = temp+ ciphertext.charAt(i);
    }
    return ciphertext = temp;

}

}

```

## **B-7: CME ANALYSIS PROGRAMS**

### **B-7i: Frequency analysis**

```

class AnalyzeFrequencies {
    private Frequency[] frequencies;
    private int totalValues, noOfEntries;
    private int[] blankOccur, fullOccur, bytesOccur;
    private String[] mostOccurrences;

    public static void main(String[] args) {

    }

    public AnalyzeFrequencies(String ciphertext, int valueLength,
CoordinateEntry[][] matrix) {
        totalValues =
(int)Math.abs(Math.ceil(ciphertext.length()/(valueLength)));
        frequencies = new Frequency[totalValues];
        for (int i = 0; i < totalValues; i++) {
            frequencies[i] = new Frequency("", 0, true, "none", "");
        }
        String temp, xString, yString;
        String actualVal = "none";
        String bitValue = "";
        Boolean exists = false;
        noOfEntries = 0;
        int x,y;

```



```

byte[] buffer = new byte[1];
Boolean matrixEntryEmpty = true;
for (int i = 0; i <= (ciphertext.length()-(valueLength*2));
i=i+(2*valueLength)) {
    temp = ciphertext.substring(i,i+(2*valueLength));
    xString = temp.substring(0, (valueLength));
    yString = temp.substring((valueLength), temp.length());
    x = Integer.parseInt(xString, 2);
    y = Integer.parseInt(yString, 2);
    if (!(matrix[x][y].entryEmpty())) {
        matrixEntryEmpty = false;
        bitValue = matrix[x][y].entryValue();
    }

    for (int j = 0; j < totalValues; j++) {
        if (frequencies[j].valueEqual(temp)) {
            frequencies[j].updateOccurrences();
            exists = true;
            break;
        }
    }
    if (!exists) {
        frequencies[noOfEntries].setFreqValue(temp);
        frequencies[noOfEntries].updateOccurrences();
        frequencies[noOfEntries].setEmpty(false);
        frequencies[noOfEntries].setBitValue(bitValue);
        noOfEntries++;
    }
    exists = false;
    bitValue = "";
}
}

```

```

public void displayFrequencies() {
    maxOccurrences();
    for (int i = 1; i < blankOccur.length; i++) {
        System.out.print(blankOccur[i]+" "+fullOccur[i]+" ");
    }
    System.out.println();
}

public void maxOccurrences() {
    int maxBlank = 0;
    int maxFull = 0;
    for (int i = 0; i < noOfEntries; i++) {
        if (!(frequencies[i].bitEqual("")) && (maxFull <
frequencies[i].getOccurrences())) {
            maxFull = frequencies[i].getOccurrences();
        } else if (frequencies[i].bitEqual("")) && (maxBlank <
frequencies[i].getOccurrences())) {
            maxBlank = frequencies[i].getOccurrences();
        }
    }
    int largest = Math.max(maxBlank,maxFull);
    blankOccur = new int[largest+1];
    fullOccur = new int[largest+1];
    int current = 0;
    for (int i = 0; i < noOfEntries; i++) {
        current = frequencies[i].getOccurrences();
        if (!(frequencies[i].bitEqual(""))) {
            fullOccur[current]++;
        } else if (frequencies[i].bitEqual("")) {
            blankOccur[current]++;
        }
    }
}

public AnalyzeFrequencies(int[] ciphertext, ByteCE[][] matrix) {

```

```

totalValues = ciphertext.length;
frequencies = new Frequency[totalValues];
for (int i = 0; i < totalValues; i++) {
    frequencies[i] = new Frequency("", 0, true, "none", "");
}
String temp, xString, yString;
String actualVal = "none";
String bitValue = "";
Boolean exists = false;
noOfEntries = 0;
int x,y;
int[] buffer = new int[2];
Boolean matrixEntryEmpty = true;
int i = 0;
while (i < (ciphertext.length-1)) {
    x = ciphertext[i];
    y = ciphertext[i+1];
    temp = x+","+y;
    i+=2;
    for (int j = 0; j < totalValues; j++) {
        if (frequencies[j].valueEqual(temp)) {
            frequencies[j].updateOccurances();
            exists = true;
            actualVal = "none";
            break;
        }
    }
    if (!(matrix[x][y].entryEmpty())) {
        actualVal = ""+(char)(matrix[x][y].entryValue()&
0xFF);
    } else {
        actualVal = "none";
    }
    if (!exists) {
        frequencies[noOfEntries].setFreqValue(temp);

```

```

        frequencies[noOfEntries].updateOccurrences();
        frequencies[noOfEntries].setEmpty(false);
        frequencies[noOfEntries].setActualVal(actualVal);
        noOfEntries++;
    }
    exists = false;
}
}

```

```

public void displayFrequenciesAES() {
    maxOccurBytes();
    for (int i = 1; i < blankOccur.length; i++) {
        System.out.print(blankOccur[i] + "," + fullOccur[i] + ",");
    }
    System.out.println();
}

```

```

public void maxOccurBytes() {
    int maxBlank = 0;
    int maxFull = 0;
    for (int i = 0; i < totalValues; i++) {
        if (!(frequencies[i].valEqual("none"))) && (maxFull <
frequencies[i].getOccurrences()) {
            maxFull = frequencies[i].getOccurrences();
        } else if (frequencies[i].valEqual("none") && (maxBlank
< frequencies[i].getOccurrences())) {
            maxBlank = frequencies[i].getOccurrences();
        }
    }
    int largest = Math.max(maxBlank,maxFull);
    blankOccur = new int[largest+1];
    fullOccur = new int[largest+1];
    mostOccurrences = new String[largest+1];
    int current = 0;
}

```

```

        for (int i = 0; i < noOfEntries; i++) {
            current = frequencies[i].getOccurrences();
            if (!(frequencies[i].valEqual("none"))) {
                fullOccur[current]++;
                mostOccurrences[current]
mostOccurrences[current]+frequencies[i].getActualVal()+" ";
            } else if (frequencies[i].valEqual("none")) {
                blankOccur[current]++;
            }
        }
    }
}

```

```

class Frequency {
    private String freqValue;
    private int noOfOccurrences;
    private Boolean isEmpty;
    private String actualValue;
    private String bitValue;

    public Frequency(String value, int occurrences, Boolean empty, String val, String
bits) {
        freqValue = value;
        noOfOccurrences = occurrences;
        isEmpty = empty;
        actualValue = val;
        bitValue = bits;
    }

    public void setBitValue(String bits) { bitValue = bits; }

    public String getBitValue() { return bitValue; }
}

```

```

public Boolean bitEqual(String check) {
    Boolean toReturn = (check.equals(bitValue));
    return toReturn;
}

public void updateOccurrences() { noOfOccurrences++; }

public int getOccurrences() { return noOfOccurrences; }

public void setEmpty(Boolean empty) { isEmpty = empty; }

public Boolean isEmpty() { return isEmpty; }

public void setActualVal(String val) { actualValue = val; }

public String getActualVal() { return actualValue; }

public Boolean valEqual(String check) {
    Boolean toReturn = (check.equals(actualValue));
    return toReturn;
}

public void setFreqValue(String value) { freqValue = value; }

public Boolean valueEqual(String toCheck) {
    Boolean toReturn = (toCheck.equals(freqValue));
    return toReturn;
}

public String getValue() { return freqValue; }
}

```

### **B-7ii: Avalanche effect**

```
import java.io.Console;
```

```

class AvalancheEffect {

    private static int bitsDiffer, positionsDiffer;
    private static int[] bOne, bTwo;

    public static void main(String[] args) {
        Console cons = System.console();
        String sOne = cons.readLine("Enter ciphertext one: ");
        String sTwo = cons.readLine("Enter ciphertext two: ");
        double posChanged = stringPos(sOne, sTwo);
        System.out.println("Bits same: "+(posChanged*100)+"%");
    }

    public AvalancheEffect(int[] bytesOne, int[] bytesTwo) {
        bOne = bytesOne;
        bTwo = bytesTwo;
    }

    public AvalancheEffect() {}

    public double calculateBits() {
        int matches = 0;
        double percentMatch;
        for (int i = 0; i < bOne.length; i++) {
            for (int j = 0; j < bTwo.length; j++) {
                if (bOne[i] == bTwo[j]) {
                    matches++;
                    break;
                }
            }
        }
        percentMatch = ((matches*1.0)/bOne.length);
        return percentMatch;
    }
}

```

```

public double calculatePositions() {
    int matches = 0;
    double percentMatch;
    for (int i = 0; i < bOne.length; i++) {
        if (bOne[i] == bTwo[i]) {
            matches++;
        }
    }
    percentMatch = ((matches*1.0)/bOne.length);
    return percentMatch;
}

public double stringBits(String one, String two) {
    int matches = 0;
    double percentMatch;
    for (int i = 0; i < one.length(); i++) {
        for (int j = 0; j < two.length(); j++) {
            if (one.charAt(i) == two.charAt(j)) {
                matches++;
                break;
            }
        }
    }
    percentMatch = ((matches*1.0)/one.length());
    return percentMatch;
}

public static double stringPos(String one, String two) {
    int matches = 0;
    double percentMatch;
    for (int i = 0; i < one.length(); i++) {
        if (one.charAt(i) == two.charAt(i)) {
            matches++;

```



```

        }
    }
    percentMatch = ((matches*1.0)/one.length());
    return percentMatch;
}
}

```

### **B-7iii: CME UTF-8 string to binary conversion**

```

import java.math.BigInteger;

class MessageToBinary {
    private char[] charSet;
    private byte[] byteSet;
    private static String binaryString;

    public static void main(String[] args) throws Exception {
        MessageToBinary toBinary = new MessageToBinary(args[0]);
    }

    public MessageToBinary(String toConvert) throws Exception {
        byteSet = toConvert.getBytes("UTF-8");
        BigInteger binaryInt = new BigInteger(byteSet);
        binaryString = binaryInt.toString(2);
    }

    public String xor(String one, String two, int stringLength) {
        String toReturn = "";
        for (int i = 0; i < (one.length()); i++) {
            if (one.charAt(i) == two.charAt((i)%two.length())){
                toReturn = toReturn+"0";
            } else {
                toReturn = toReturn+"1";
            }
        }
        return toReturn;
    }
}

```

```
}

public MessageToBinary() {}

public String getBinaryString () {
    return binaryString;
}

public String convertToCharacters(String binaryToConvert) {
    BigInteger toHex = new BigInteger(binaryToConvert,2);
    byte[] temp = toHex.toByteArray();
    String toReturn = "";
    try {
        toReturn = new String(temp, "UTF-8");
    } catch (Exception e) {}
    return toReturn;
}
}
```

## Appendix C: Testing Data

### C-1: DATA USED IN COMPARISON OF AES, RC4 AND CME

[The following test data was taken from Austen (2006), p.3]

*Data Size: 304*

IT is a truth universally acknowledged

*Data Size: 928*

IT is a truth universally acknowledged, that a single man in possession of a good fortune must be in want of a wife.

*Data Size: 3024*

IT is a truth universally acknowledged, that a single man in possession of a good fortune must be in want of a wife. However little known the feelings or views of such a man may be on his first entering a neighbourhood, this truth is so well fixed in the minds of the surrounding families, that he is considered as the rightful property of some one or other of their daughters.

[The following test data was taken from Shakespeare & Ackroyd (2006), p. 1100]

*Data Size: 4408*

HAMLET: To be, or not to be--that is the question/Whether 'tis nobler in the mind to suffer/The slings and arrows of outrageous fortune/Or to take arms against a sea of troubles/And by opposing end them. To die, to sleep--/No more--and by a sleep to say we end/The heartache, and the thousand natural shocks/That flesh is heir to. 'Tis a consummation/Devoutly to be wished. To die, to sleep/To sleep--perchance to dream: ay, there's the rub/For in that sleep of death what dreams may come/When we have shuffled off this mortal coil/Must give us pause.

*Data Size: 8144*

HAMLET: To be, or not to be--that is the question/Whether 'tis nobler in the mind to suffer/The slings and arrows of outrageous fortune/Or to take arms against a sea of troubles/And by opposing end them. To die, to sleep--/No more--and by a sleep to say we



110010010100110001111111011000111010110111111101111001111100011001100  
1011110001110100101110010001111011011001101011010111101011

*Data Size: 256*

0100101001000110100101111011101011101101000110100111110010101011011110  
0110001100111001001100101000100010000100000110000110101100011100111011  
0000110011100110100110101101010001100111001011111100100010001000000101  
1000101110100000011010101011110011101100000000

*Data Size: 512*

01000101000001000001001001111010111011011110110011100111001100100111001000  
1000101100000011011110000010010010001001100100110110101100111111111101  
1001000010011000110111010110000011000110101001100010110110111101001100  
0000110010110110101011000101011111110000001001100100101101010000101111  
0101000101111101001000010010011100111110100111100110010101010001010111  
0000010000010101111110010001010011110101100011100100010111110011010011  
0011000011110101110011011100101010011101010000101001010100010101010111  
1100111110010001101000

## Appendix D: Example Results

### D-1: EXAMPLE RESULT FROM AES

AES Encryption with Randomly Generated Key  
Set up complete, time taken: 202 ms  
Total memory used: 3.6447067260742188 MB  
Enter plaintext: IT is a truth universally acknowledged  
Enter times to encrypt the data: 1  
plain: IT is a truth universally acknowledged  
37,5,  
decrypt: IT is a truth universally acknowledged  
10,0,384  
37,5,  
1,1.4362640380859375,1.3887176513671875  
Do you want to keep encrypting with this key? 1=Y, 2=N : 2

### D-2: EXAMPLE RESULT FROM RC4

RC4 Encryption with Random 128 bit key  
Set up complete, time taken: 783 ms  
Total memory used: 2.3383026123046875 MB  
Enter plaintext to encrypt: IT is a truth universally acknowledged  
Enter number of times to encrypt the data: 1  
33,2,  
decrypt: IT is a truth universally acknowledged  
5,0,304  
1,1.3647308349609375,1.3632888793945312  
Do you want to keep encrypting with this key? 1=Y, 2=N : 2

### D-3: EXAMPLE RESULT FROM ECDH

Secret computed by U:  
1F41B47533CF7128ED4B0C12335A8AD7F96850EB3A704B83  
Secret computed by V:  
1F41B47533CF7128ED4B0C12335A8AD7F96850EB3A704B83  
Total time for setup: 157 ms  
Total memory used: 1.19158935546875 MB

### D-4: EXAMPLE RESULT FROM VC

Set up complete. Time taken: 0 ms.  
Total memory used: 0.40460205078125 MB  
  
Enter plaintext to encrypt into shares: 1101000110111011  
Enter number of times encryption/decryption should be performed: 1  
Encryption #1  
Shares generated. Time taken: 1 ms.  
Share one:

0101110001101010100101100011011011000110100101101001100110011010  
Share two:  
1010001101100101100101100011100100110110011010010110100101100101  
Share length: 64 bits.  
1,0.44730377197265625,1.7277297973632812  
Shares recombined. Time taken: 0 ms.  
Decrypted plaintext: 1101000110111011  
Decrypted plaintext matches original data: true

#### **D-5: EXAMPLE RESULT FROM BYTE CME**

Total strings: 256  
Number of occupied spaces: 32768  
Number of blank spaces: 32768  
Total matrix size: [256,256]  
Total memory used: 1.2040176391601562 MB  
Set up complete, time taken: 74 ms  
Enter data to encrypt: IT is a truth universally acknowledged  
Entry length: 304  
How many times do you want to encrypt and decrypt the data? : 1  
0,0,1216  
1,1.2431411743164062,1.2433090209960938  
38,38,  
Original plaintext: IT is a truth universally acknowledged  
Encrypt more data? 1 = Y, 2 = N : 2

#### **D-6: EXAMPLE RESULT FROM BIT-STRING CME**

Total strings: 16  
Number of occupied spaces: 128  
Number of blank spaces: 128  
Total matrix size: [16,16]  
Set up complete, time taken: 24 ms.  
Total memory used: 0.43534088134765625 MB  
Enter data to encrypt: 1101000110111011  
Is the text in binary format? Y/N: Y  
Entry length: 16  
How many times do you want to encrypt and decrypt the data? : 1  
Divisor: 4.0  
Entry does not require padding.  
Plaintext:  
0011000100011011  
Ciphertext:  
0001101010100001011100000001001100100011001010001011110100110011  
0 0 64  
1,0.47379302978515625,1.7539520263671875  
4,4,  
Decoded binary string matches original input: true  
Encrypt more data? 1 = Y, 2 = N : 2

## FORM PGR15 DEPOSIT OF THESIS/EXEGESIS/DISSERTATION IN THE AUT LIBRARY

**PLEASE NOTE**

- This form must be typed. Handwritten forms will not be accepted.
- The completed and signed form should be bound into the copy of the thesis/exegesis intended for the AUT University Library
- If the work is to be treated as confidential or is embargoed for a specified time, form PGR16 must also be completed and bound into the thesis/exegesis.

<b>Student ID No</b>	1112604	<b>Name</b>	Erin Chapman
<b>Faculty</b>	Computer and Mathematical Sciences	<b>School/Dept</b>	ECMS
<b>Programme</b>	MCIS	<b>Year of submission (for examination)</b>	2016
<b>Research Output</b>	Thesis <input checked="" type="checkbox"/>	Exegesis <input type="checkbox"/>	Dissertation <input type="checkbox"/>
<b>Thesis Title</b>	Using Graphic Based Systems to Improve Cryptographic Algorithms		
		<b>Points Value</b>	<b>120</b>

### DECLARATION

I hereby deposit a print and digital copy of my thesis/exegesis with the Auckland University of Technology Library. I confirm that any changes required by the examiners have been carried out to the satisfaction of my primary supervisor and that the content of the digital copy corresponds exactly to the content of the print copy in its entirety.

This thesis/exegesis is my own work and, to the best of my knowledge and belief, it contains:

- no material previously published or written by another person (except where explicitly defined in the acknowledgements);
- no material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

### CONDITIONS OF USE

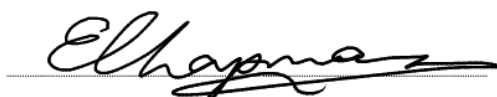
From the date of deposit of this thesis/exegesis or the cessation of any approved access restrictions, the conditions of use are as follows:

1. This thesis/exegesis may be consulted for the purposes of private study or research provided that:
  - (i) appropriate acknowledgement is made of its use;
  - (ii) my permission is obtained before any material contained in it is published.
2. The digital copy may be made available via the Internet by the AUT University Library in downloadable, read-only format with unrestricted access, in the interests of open access to research information.
3. In accordance with Section 56 of the Copyright Act 1994, the AUT University Library may make a copy of this thesis/exegesis for supply to the collection of another prescribed library on request from that library.

### THIRD PARTY COPYRIGHT STATEMENT

I have either used no substantial portions of third party copyright material, including charts, diagrams, graphs, photographs or maps, in my thesis/exegesis or I have obtained permission for such material to be made accessible worldwide via the Internet. If permission has not been obtained, I have asked/will ask the Library to remove the third party copyright material from the digital copy.

**Student's Signature**



**Date** 23 November 2016