No Restrictions.

ERRATA

p. 37 para 2, 2nd line: replace “restrivtion” with “restriction”

p. 80 para 1, last line: insert “to” between “required” and “execute”

p. 80 para 2, 1st sentence: replace “Section 0” with “Section D.3.3”

A Novel MPEG-1 Partial Encryption Scheme for the Purposes of Streaming Video

Jason But

BE (Eng) (Hons), BSc (Comp. Sci.)

A thesis submitted for the degree of

Doctor of Philosophy

in the

Department of Electrical and Computer Systems Engineering

Monash University

Clayton, Victoria 3168, Australia

January 2004

Table Of Contents

Table Of Contents ______ i
List Of Figures ______ vii
List Of Tables ______ ix
Abstract ______ xi
Statement ______ xiii
Acknowledgments ______ xv

Chapter 1 Introduction ______ 1
1.1 Internet Applications ______ 1
1.2 Video-on-Demand as an Application ______ 2
1.3 Problems Faced by Video-on-Demand ______ 4
1.4 Copyright Protection ______ 6
1.5 Structure and Contributions of the Thesis ______ 8
1.6 Final Remarks ______ 9

Chapter 2 Copyright Protection of Streaming MPEG Video ______ 11
2.1 Next Generation Internet ______ 12
2.1.1 Improved Bandwidth ______ 12
2.1.2 Quality Of Service ______ 14
2.1.3 Applications ______ 15
2.1.4 Video On Demand Systems ______ 16
2.1.4.1 Network QoS Requirements ______ 17
2.1.4.2 Server Requirements ______ 17
2.1.4.3 Client Requirements ______ 18
2.2 Streaming Server Implementation ______ 19
2.2.1 Indexed Playback Mode Implementation ______ 19
2.2.2 High-Speed Playback Mode Implementation ______ 20
2.2.3 Installation of Assets ______ 22
2.3 Design of Video on Demand Systems ______ 22
2.3.1 Single Server Design ______ 23
2.3.2 Distributed Server Design ______ 24
2.3.3 Multi-Party Distributed Server Design ______ 28
2.3.4 Non-Technical Issues ______ 31
2.3.4.1 Payment for Access Privileges ______ 31
2.3.4.2 Copyright Concerns ______ 31
2.4 Copyright ______ 32
2.4.1 Digital Rights ______ 32
2.4.2 Digital Theft ______ 33
2.4.2.1 Theft from the Central Server ______ 33
2.4.2.2 Theft from Streaming Servers ______ 34
2.4.2.3 Theft in Transit ______ 34
2.4.2.4 Client Theft ______ 35
2.5 Video Encryption Requirements ______ 35
2.5.1 Copyright Owner Requirements ______ 36
2.5.2 Distributed Server Arrangement Requirements ______ 37
2.5.3 Streaming Server Requirements ______ 38
2.5.4 Client Requirements ______ 38
2.6 Conclusion ______ 39

Chapter 3 Existing MPEG-1 Encryption techniques ______ 41
3.1 Network and Transport Layer Encryption ______ 41

3.1.1 IPSec Encryption 42
3.1.2 SSL Encryption 43
3.2 Full Encryption 44
3.3 Partial Encryption 44
3.3.1 SECMPEG 45
3.3.2 Zig-Zag Permutation Algorithm 46
3.3.3 Video Encryption Algorithm 48
3.3.4 Video Encryption Algorithm – Number 2 49
3.3.5 Frequency Domain Scrambling Algorithm 51
3.3.6 A Unique Cipher 52
3.3.7 Multi-Layer Encryption 52
3.3.8 Selective Macroblock Encryption 53
3.3.9 AEGIS Algorithm 55
3.4 Conclusion 55
Chapter 4 A Novel MPEG-1 Partial Selection Scheme for the Purposes of Encryption 57
4.1 MPEG-1 System Stream Encryption 57
4.1.1 Examination of the MPEG-1 System Stream 58
4.1.2 Processing the MPEG-1 System Stream 59
4.1.2.1 Parsing the MPEG-1 System Stream 60
4.1.3 Summary of MPEG-1 System Stream Encryption 63
4.2 MPEG-1 Video Stream Encryption 64
4.2.1 Examination of the MPEG-1 Video Stream 64
4.2.1.1 Restrictions on Encryption of Macroblocks 66
4.2.1.2 Analysis of Selection Criteria 66
4.2.2 Processing the MPEG-1 Video Stream 67
4.2.2.1 Designing the Partial Stream Selection State Machine 67
4.2.2.2 Designing the Prototype Cipher 70
4.2.2.3 Support for Indexed and High-Speed Playback Modes 71
4.2.3 Summary of MPEG-1 Video Stream Encryption 72
4.3 MPEG-1 Audio Stream Encryption 72
4.3.1 Examination of the MPEG-1 Audio Stream 73
4.3.2 Processing the MPEG-1 Audio Stream 73
4.3.3 Summary of MPEG-1 Audio Stream Encryption 74
4.4 Prototype System Testing 75
4.4.1 Trial Conditions 76
4.4.1.1 Input Files 76
4.4.1.2 Test Applications 76
4.4.1.3 Test Platforms 77
4.4.2 Trial Results 77
4.4.2.1 Percentage of the MPEG-1 File Encrypted 77
4.4.2.2 Is the Encryption Process Repeatable and Reversible 78
4.4.2.3 CPU Requirements for Encryption/Decryption 79
4.4.2.4 Verification of Functionality with Existing Streaming Video Servers 81
4.5 Summary 83
Chapter 5 A Novel MPEG-1 Partial Encryption Scheme 85
5.1 Selecting a Secure Cipher 85
5.1.1 Restrictions on the Cipher 86
5.1.2 Public Key Ciphers 86
5.1.3 Private Key Ciphers – Block Ciphers 87
5.1.4 Private Key Ciphers – Stream Ciphers 87
5.1.4.1 Feedback Shift Registers 88
5.1.4.2 RC4 89
5.1.4.3 SEAL 92
5.1.5 Conclusion 98
5.2 MPEG-1 Video Stream Encryption 98

5.2.1 Incorporation of SEAL Stream Cipher into Existing System 99
5.2.1.1 Security Provided by the Modified SEAL Cipher 100
5.2.2 Resynchronisation of Cipher at each Picture 103
5.3 MPEG-1 Audio Stream Encryption 106
5.3.1 Incorporation of SEAL Stream Cipher Into Existing System 107
5.3.2 Lack of Resynchronisation Points within the Audio Stream 107
5.3.3 Calculating the Audio Cipher Resynchronisation Value – *n* 108
5.4 System Testing 110
5.4.1 Is the Encryption Process Repeatable and Reversible 110
5.4.2 CPU Requirements for Encryption/Decryption 111
5.4.3 Verification of Functionality with Existing Video Servers 112
5.4.4 Summary 113
5.5 Effects on Streaming Video Implementations 113
5.5.1 Effect of a Bit Error 114
5.5.2 Effect of Lost or Dropped Bits 116
5.5.3 Effect of Lost or Dropped Packets 116
5.5.4 Effect of Late Delivery of a Packet 118
5.6 Conclusion 118
Chapter 6 Application to Streaming MPEG-2 121
6.1 MPEG-2 Scrambling 121
6.2 MPEG-2 Stream Format 121
6.2.1 MPEG-2 Program Stream 122
6.2.2 MPEG-2 Transport Stream 122
6.2.3 MPEG-2 Video Stream 123
6.2.4 MPEG-2 Audio Stream 123
6.3 Compatibility with the Proposed Cipher 124
6.3.1 MPEG-2 Video Stream 124
6.3.2 MPEG-2 Audio Stream 125
6.4 Conclusion 126
Chapter 7 Conclusion 129
Appendix A The MPEG-1 Bitstream Format 137
A.1 History of MPEG-1 137
A.2 MPEG-1 138
A.2.1 MPEG-1 System Stream 140
A.2.1.1 ISO 11172 Layer 140
A.2.1.2 Pack Layer 141
A.2.1.3 Packet Layer 143
A.2.2 MPEG-1 Video Compression 145
A.2.3 MPEG-1 Video Stream 146
A.2.3.1 Sequence Layer 147
A.2.3.2 Group Of Pictures Layer 149
A.2.3.3 Picture Layer 150
A.2.3.4 Slice Layer 152
A.2.3.5 Macroblock Layer 152
A.2.3.6 Block Layer 154
A.2.4 MPEG-1 Audio Compression 154
A.2.5 MPEG-1 Audio Stream 156
A.2.5.1 MPEG-1 Audio Stream Packet Header 156
A.2.5.2 MPEG-1 Audio Layer I Data Representation 157
A.2.5.3 MPEG-1 Audio Layer II Data Representation 158
A.2.5.4 MPEG-1 Audio Layer III Data Representation 158
A.3 MPEG-2 160
A.4 MPEG-4 161

Appendix B An Introduction to Cryptography 163
B.1 Basic Cryptographic Techniques 163
B.1.1 Terminology 164
B.1.2 Cryptographic Keys 166
B.1.3 One Time Pad Cipher 167
B.1.4 Cryptographic Attacks 169
B.1.4.1 Ciphertext Only Attack 169
B.1.4.2 Known Plaintext Attack 170
B.1.4.3 Chosen Plaintext Attack 170
B.1.4.4 Adaptive Chosen Plaintext Attack 170
B.2 Public Key Cryptography 171
B.2.1 Principles Behind Public Key Cryptography 171
B.2.1.1 One-Way Functions 171
B.2.1.2 Trapdoor One-Way Functions 173
B.2.1.3 Generating Prime Numbers 173
B.2.1.4 Relative Prime Numbers 173
B.2.1.5 Extended Euclidean Algorithm 174
B.2.1.6 Weaknesses and Practicalities 174
B.2.2 Current Public Key Cryptographic Algorithms 175
B.2.2.1 RSA Public Key Cryptosystem 175
B.2.2.2 Other Public Key Cryptosystems 176
B.2.2.3 Recommended Key Lengths for Public Key Cryptosystems 177
B.2.3 The Trusted Authority Public Key Database 178
B.2.4 Amenability to Encryption of Streaming Multimedia 179
B.3 Private Key Cryptography 179
B.3.1 Block Ciphers 180
B.3.1.1 Block Cipher Modes 180
B.3.1.2 Principles of Block Cipher Cryptology 183
B.3.1.3 Weaknesses and Practicalities 187
B.3.1.4 DES Block Cipher 187
B.3.1.5 DES Variations 191
B.3.1.6 Recommended Key Lengths for Block Ciphers 192
B.3.1.7 Amenability to Encryption of Streaming Multimedia 194
B.3.2 Stream Ciphers 194
B.3.2.1 Principles of Stream Cipher Cryptology 195
B.3.2.2 Weaknesses and Practicalities 199
B.3.2.3 Recommended Key Lengths for Stream Ciphers 200
B.3.2.4 Amenability to Encryption of Streaming Multimedia 200
B.4 Conclusion 201

Appendix C Source Code 203
C.1 ClassFactory and StreamCipherBase Classes 203
C.2 Stream Ciphers 204
C.2.1 XORStreamCipher Class 204
C.2.2 SEALStreamCipher Class 204
C.3 Parsing and Encrypting the Video and Audio Streams 207
C.3.1 MPEGVideoParser Class 207
C.3.2 MPEGAudioParserClass 208
C.4 Parsing and Encrypting the System Stream 209
C.5 Listings 209
C.6 Applications 230
C.6.1 MPEG-1 File Encryption 230
C.6.2 DirectShow MPEG-1 Cipher Filter 230
C.6.3 DirectShow Stream Playback Application 231
C.6.4 MediaBase Playback Application 231
C.6.5 Other Applications 232

Appendix D Experimental Results 233
D.1 Input Files 233
D.2 Proportion of Stream Selected for Encryption 235
D.3 Prototype Cipher 236
D.3.1 Testing Repeatability 237
D.3.2 Testing Reversibility 239
D.3.3 Performance Testing 241
D.3.4 Testing Functionality 247
D.3.4.1 Microsoft NetShow Theatre Streaming Server 247
D.3.4.2 SGI Mediabase 3.1 Streaming Server 250
D.3.4.3 Apple QuickTime Streaming Server 252
D.4 SEAL Based Cipher 254
D.4.1 Testing Repeatability 254
D.4.2 Testing Reversibility 255
D.4.3 Performance Testing 257
D.4.4 Testing Functionality 259
D.4.4.1 Microsoft NetShow Theatre Streaming Server 259
D.4.4.2 SGI Mediabase 3.1 Streaming Server 260
D.4.4.3 Apple QuickTime Streaming Server 261
D.5 Conclusions 262

Appendix E References 263

List Of Figures

Figure 1-1: Digital Networked Applications and their Traditional Counterparts ___ 2
Figure 1-2: Different Video-on-Demand Applications ___ 3
Figure 1-3: Parts of a Copyright Protection Scheme for Streaming Video ___ 7
Figure 2-1: Modified MPEG-1 Video Sequence to Implement High-Speed Playback ___ 21
Figure 2-2: Monolithic Single Server VoD System Design ___ 23
Figure 2-3: Distributed Server VoD System Design ___ 25
Figure 2-4: Multi-Party Distributed Server VoD System ___ 28
Figure 2-5: Customer Authentication Procedure ___ 36
Figure 4-1: State Machine to Encrypt an MPEG-1 System Stream ___ 62
Figure 4-2: Selective Encryption of an MPEG-1 Video Stream ___ 67
Figure 4-3: Mechanics of the MPEG-1 Slice Bit Sequence ___ 68
Figure 4-4: Algorithm to Encrypt Macroblocks Within an MPEG-1 Video Stream ___ 69
Figure 4-5: State Machine to Encrypt an MPEG-1 Video Stream ___ 70
Figure 4-6: Generalised Cipher Function for MPEG-1 Video Stream Encryption ___ 70
Figure 4-7: Simple Cipher for use in MPEG-1 Video Stream Encryption ___ 71
Figure 4-8: Simple Cipher for use in MPEG-1 Audio Stream Encryption ___ 74
Figure 4-9: State Machine to Encrypt an MPEG-1 Audio Stream ___ 75
Figure 4-10: Performance Results Using the Prototype Encryption Scheme ___ 80
Figure 5-1: Linear Feedback Shift Register Random Stream Generator ___ 89
Figure 5-2: RC4 S-Box Initialisation ___ 90
Figure 5-3: RC4 Pseudo-Random Sequence Generation ___ 91
Figure 5-4: SEAL $G_{key}(n)$ function ___ 93
Figure 5-5: SEAL Table Generation ___ 94
Figure 5-6: SEAL Random String Generation ___ 95
Figure 5-7: Modification of SEAL XOR Function for use in Video Stream Encryption ___ 100
Figure 5-8: Format of 25-bit Time-Stamp within GOP Header ___ 104
Figure 5-9: Modified State Machine to Encrypt an MPEG-1 Video Stream ___ 105
Figure 5-10: Modification of SEAL XOR Function for use in Audio Stream Encryption ___ 107
Figure 5-11: Modified State Machine to Encrypt an MPEG-1 Audio Stream ___ 110
Figure 5-12: Performance Results Using the Secure Encryption Scheme ___ 112
Figure A-1: MPEG-1 System Stream ___ 140
Figure A-2: MPEG-1 Pack Header Definition ___ 142
Figure A-3: MPEG-1 System Header Definition ___ 143
Figure A-4: MPEG-1 Packet Header Definition ___ 144
Figure A-5: MPEG-1 Encoded Frame Order ___ 146
Figure A-6: MPEG-1 Video Stream ___ 147
Figure A-7: MPEG-1 Video Sequence Header Definition ___ 148
Figure A-8: MPEG-1 Group Of Pictures Header Definition ___ 150
Figure A-9: MPEG-1 Picture Header Definition ___ 151
Figure A-10: MPEG-1 Slice Header Definition ___ 152
Figure A-11: MPEG-1 Macroblock Header Definition ___ 153
Figure A-12: MPEG-1 Audio Header Definition ___ 157
Figure B-1: Basic Cryptographic Framework ___ 165
Figure B-2: Cryptographic System Inclusive of Keys ___ 166
Figure B-3: Private Key Cryptographic System ___ 167
Figure B-4: CBC Mode Encryption/Decryption Process ___ 181
Figure B-5: CFB Mode Encryption/Decryption Process ___ 182
Figure B-6: OFB Mode Encryption/Decryption Process ___ 183
Figure B-7: DES Encryption Algorithm ___ 189
Figure B-8: DES Fiestel Network f() function ___ 190
Figure B-9: DES Key Schedule ___ 191
Figure B-10: Linear Feedback Shift Register Random Stream Generator ___ 196
Figure B-11: Shift Register Selection Combination ___ 198
Figure C-1: SEAL Sample Implementation ___ 206

Figure D-1: DirectShow Filter Graph for Encrypted Video Playback from NetShow Theatre _245

List Of Tables

Table 4-1 Proportions of Test Bitstreams Selected for Encryption ____ 78
Table 4-2 Reversing the MPEG-1 System Stream Encryption ____ 79
Table 4-3 Required CPU Load for Plaintext Playback ____ 81
Table 4-4 Streaming an Encrypted MPEG-1 File from a Streaming Video Server ____ 83
Table 5-1 SEAL Keys Used in Testing Encryption Scheme ____ 111
Table 5-2 Reversing the MPEG-1 System Stream Encryption ____ 111
Table 5-3 Streaming an Encrypted MPEG-1 File from a Streaming Video Server ____ 113
Table 5-4 Proportion of Video Stream given Frame type ____ 117
Table A-1 MPEG-1 System Stream Unique 32 Bit Byte Aligned Start Codes ____ 141
Table A-2 MPEG Video Stream Unique 32 Bit Byte Aligned Start Codes ____ 148
Table B-1 Common terminology used in the science of cryptology ____ 164
Table B-2 RSA Encryption Recommendations ____ 176
Table B-3 Summary of Block Cipher Modes of Operation ____ 184
Table B-4 Block Cipher cracking times based on key length ____ 193
Table D-1 MPEG-1 Test File Details ____ 234
Table D-2 Proportions of Bitstreams Selected for Encryption ____ 236
Table D-3 Encryption Statistics and Repeatability of Encryption Process – Prototype Cipher ____ 237
Table D-4 Byte Count Distribution in Input Streams ____ 238
Table D-5 Comparison of Decrypted File with Original File given (En/De)cryption Key Pair ____ 240
Table D-6 Test Platform Specifications ____ 241
Table D-7 Basic Performance Results on Test Platform 1 ____ 243
Table D-8 Basic Performance Results on Test Platform 2 ____ 244
Table D-9 Real-time Decryption and Playback Performance Results on Test Platform 1 ____ 246
Table D-10 Real-time Decryption and Playback Performance Results on Test Platform 2 ____ 247
Table D-11 Functionality Tests with Microsoft NetShow Theatre Streaming Server ____ 250
Table D-12 Functionality Tests with SGI Mediabase 3.1 Streaming Server ____ 252
Table D-13 Encryption Statistics and Repeatability of Encryption Process – SEAL Cipher ____ 254
Table D-14 Comparison of Decrypted File with Original File – SEAL Cipher ____ 256
Table D-15 Basic Performance Results on Test Platform 1 – SEAL Cipher ____ 257
Table D-16 Basic Performance Results on Test Platform 2 – SEAL Cipher ____ 258
Table D-17 Real-time Playback Performance Results on Test Platform 1 – SEAL Cipher ____ 258
Table D-18 Real-time Playback Performance Results on Test Platform 2 – SEAL Cipher ____ 259
Table D-19 Functionality Tests with Microsoft NetShow Theatre Streaming Server – SEAL ____ 260
Table D-20 Functionality Tests with SGI Mediabase 3.1 Streaming Server – SEAL ____ 261

Abstract

Video-on-Demand (VoD) has often been touted as one of the next killer Internet applications. VoD however, is yet to capitalise on its potential to become a highly patronised service, that is has not succeeded is due to many reasons, technological, economical and legal. Recent technological advances have largely negated the first issue and significantly contributed to reducing costs. It is now possible to build a functional VoD service at viable cost. It now becomes equally important to consider the legal aspects of providing a VoD service, both guarantee of payment and protection of the copyright ownership. In this thesis, I explore the issue of copyright protection of streaming MPEG-1 Video and present a novel MPEG-1 Encryption Technique that is compatible with a wide range of both existing and future streaming video servers. I will show how the concepts used in this approach can also be applied to copyright protection of streaming MPEG-2 Video.

Statement

To the best of my knowledge and beiief, this thesis does not contain any material previously published or written by any other person, except where due reference is made. The thesis contains no material that has been accepted for the award of any other degree or diploma in any university or other institution.

[redacted]

Acknowledgments

Production of this thesis has been a long and laborious task, often leaving me wondering whether it would ever be completed. The fact that it has been finished is largely due to the input, assistance, love and badgering of many people. I would like to take the time to thank each of these people individually.

Firstly, thanks to all members of the Centre for Telecommunications and Information Engineering (CTIE) at Monash University for convincing me to undertake the task of attempting a PhD.

Particular thanks go to my supervisor – Greg Egan – both for his continuing reassurance that this thesis was on the right track, and for his confidence in the eventual outcome of my work.

Thanks also to those who helped proof read this document –James, and Joe – your comments brought some much needed polish to the final presentation.

To my family, who encouraged me throughout the entire process. Special thanks to my wife Eleonora, who knew both when I needed encouragement and when I required the proverbial 'kick up the backside' when my effort was failing.

And finally to my newborn daughter Katia, who has suffered more in the first few weeks of her life than I ever have. She has shown me what life is all about, and taught me what is really important. This thesis is for her.

Chapter 1

Introduction

High-quality streaming Video-on-Demand (VoD) has often been touted as one of the next killer applications, after email and Web Browsing, to succeed on the Internet. VoD however, is yet to capitalise on its potential to become a highly patronised service. That is has not succeeded where these other applications have is due to many reasons: technological, economical and legal. Recent technological advances have largely negated the first issue and significantly contributed to reducing costs. While these first two issues are by no means rendered insignificant, and further research will undoubtedly uncover improved solutions, it has now become possible to build a VoD service that is functional at viable costs. As such, it becomes equally as important to consider the legal aspect of providing a VoD service. Much of this issue rests on the guarantee of payment and the protection of the content owners copyright. In this thesis, I will explore the issue of copyright protection and develop an encryption technique whereby digital content is encrypted while in transit on the Internet to protect against theft. Copyright protection is an issue that must be addressed prior to profitable VoD content being made available for streaming video services.(Gemmell et al., 1995; Fist, 1994; Viña et al., 1994; Chang et al., 1994; Hsing et al., 1993; Little and Venkatesh, 1994; Memon and Wong, 1998)

1.1 Internet Applications

While the Internet hosts many applications, there are two that have acquired a user base that is large enough that the applications themselves have become the predominant reasons for using the Internet. These applications are email and the World Wide Web (WWW). Email enables users on a network to communicate written messages and files between each other. It is both cheap and simple to use, and fulfils a perceived need amongst network users – the replacement of the non-digital postal service. Similarly, the Web is an application that enables network users to publish information which can then be searched, read and downloaded by other users on the network. Like email, the Web offered a digital replacement for existing services – the publishing of information. The advantages of these applications over their non-digital counterparts in part ensured their success. Email offered a free, instantaneous personal communications tool, while the Web enabled low cost, large scale, anonymous publishing to all users, providing individuals with the same capability and reach as major corporations.

As an application, Video-on-Demand (VoD) also seeks to become a digital replacement for an existing service, in this case the Video Hire store. To succeed in a commercial sense, VoD must offer both a better and cheaper service than its counterpart. VoD has two classes of users:

- **Content providers** – people who own and distribute the media content.
- **Content consumers** – people who access the provided material.

It is important that both these classes of users find a VoD service to be useful for it to be heavily patronised. Only patronage by many users will ensure the success of VoD as an application. Figure 1-1 shows the aforementioned applications and their traditional counterparts, these applications are in competition with each other.

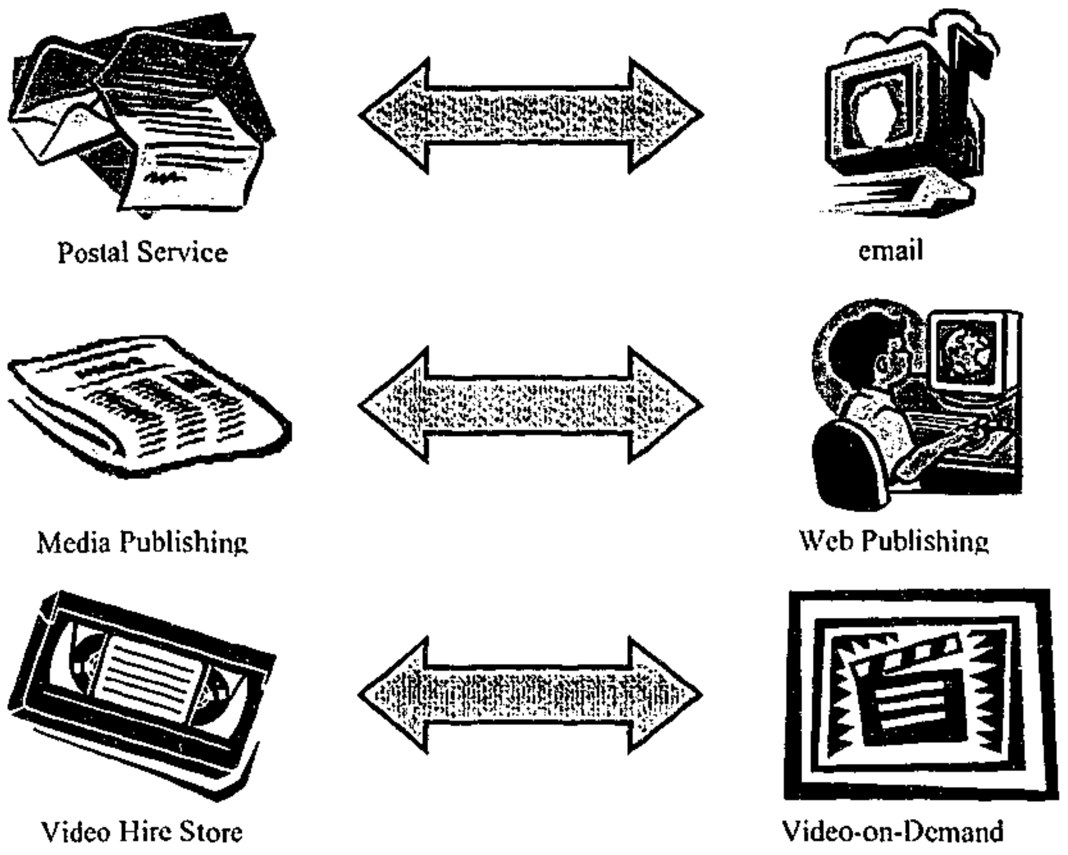


Figure 1-1: Digital Networked Applications and their Traditional Counterparts

1.2 Video-on-Demand as an Application

Video-on-Demand is an all-encompassing term describing a wide range of applications that all involve the streaming of individual video content across a network. These applications are outlined in Figure 1-2:

- **Low Bit-rate Wireless VoD** – this type of application covers the concept of streaming video to wireless devices such as mobile phones or palm-type hand-held computers. These lead to a unique environment for implementation where available bandwidth is low with a significant bit error rate, while the computing power available at the client is low with a subsequent small screen size. While computing power can always be increased, the screen size is likely to remain small for the foreseeable future. The small screen size coupled with low network bandwidth ensures that this type of video will be at a low bit-rate and quality. If we consider possible usages for this application, we can see people watching shows during a commute on public transport – the short time available means we will see short shows, perhaps the news or television comedy shows – or perhaps retrieving information such as movie reviews prior to attending the cinema.(Microsoft, 2002; Apple, 2002a)

- **Playback to an Internet connected PC** – this type of application is similar to the scope of VoD services available on the Internet today: low bitrate streaming video served from a single central server. This type of application is not scalable to provide higher quality streaming or service a larger customer base, however it is finding some success in introducing Internet users to the concept of streaming video. At this stage, the content is predominantly either copyright free, of little value and therefore not worth stealing, or of sufficiently poor quality to negate against theft. This type of application can be found in Internet streaming of sporting events, concerts, news media, advertising, and other applications. In these instances, while access to the media may be protected and require prior payment, the media content itself is not of sufficient quality or value to require protection of the copyright. While higher-quality video may become popular in this type of application, this would require a change in the server architecture to better support the streaming of higher bit-rate streams over the Internet.(Branch and Durran, 1996; Apple, 2002a; Fist, 1994; Microsoft, 2002; Wu et al., 2001)

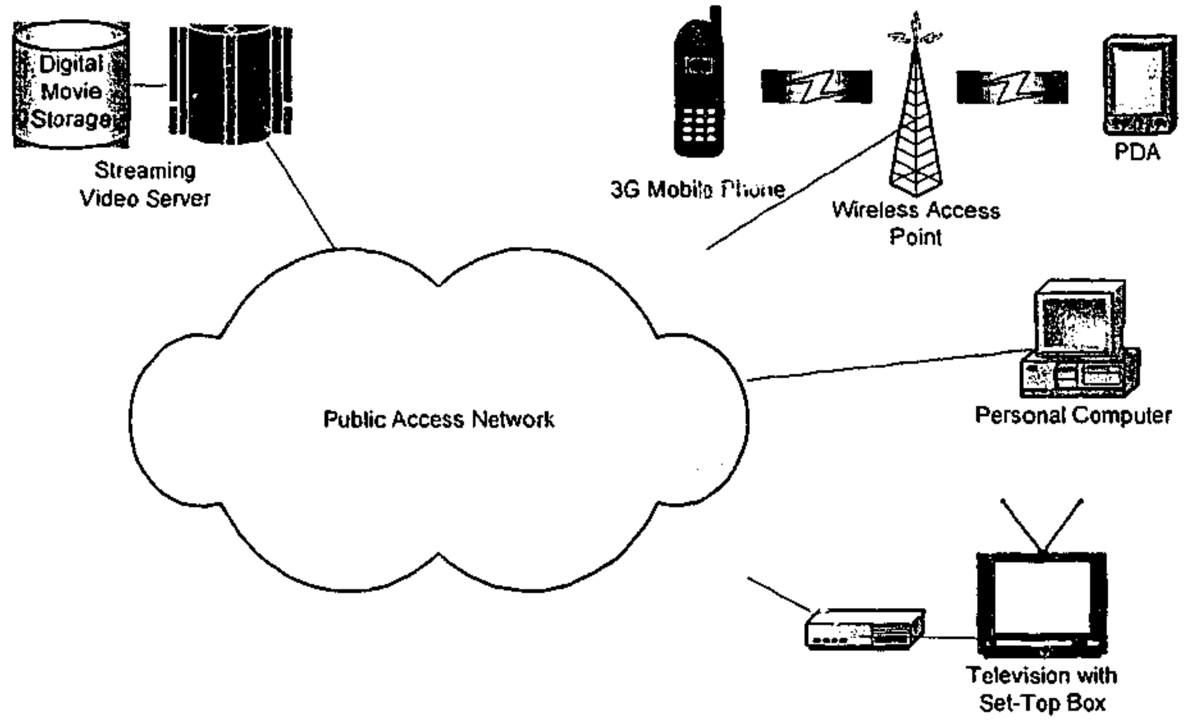


Figure 1-2: Different Video-on-Demand Applications

- **Entertainment quality VoD** – this type of application involves the streaming of high bitrate video, most likely to a black box connected to a television set in the users living room, but also potentially to an Internet connected PC. This application is where VoD seeks to replace the local Video Hire store, allowing users to select and play a video without leaving their house. In this instance, the quality must be high enough to allow playback on a large screen television set at a quality matching VHS tape in the short term and DVD in the long term. This is also the type of VoD application where copyright protection becomes more important as the quality of the digital material is higher and therefore more valuable.(Rangan et al., 1992; Little and Venkatesh, 1994; Lin et al., 2001; Hsing et al., 1993; Fist, 1994)

One could conceive of other applications that VoD could be applied to, but those mentioned above are the obvious applications that are most likely to be adopted at this stage. Of interest in this thesis is the third type – streaming of entertainment quality video. It is this application where copyright protection becomes an issue to the content owner and therefore, also where it becomes a major issue in the implementation of this type of VoD service.

1.3 Problems Faced by Video-on-Demand

The major problems that VoD has to overcome are economics and the fact that computer and network requirements for providing a VoD service are higher than for other Internet applications such as email and the Web. For some time, VoD was not economically feasible, the existing Internet could not support a large scale implementation. Similarly, the costs involved in building a Local Area Network and Streaming Server capable of supporting such an application were extremely high, much higher that the existing non-networked video application – that of hiring a video tape or DVD from a local video hire store for viewing at home. Beyond any further considerations, this immediately meant that technology at the time was not ready for a commercial streaming video application and any prototype systems that were implemented were simply that, prototype applications to showcase the potential and what could be achieved should implementation costs decrease to a suitable level. More recently we have seen three changes that can be used to argue the case that the time of VoD and Streaming Video services is nearing:(Cocchi et al., 1993; Cornall, 1998; Hsing et al., 1993; Jung et al., 2000; Little and Venkatesh, 1994)

- The extremely low cost of implementing a high-bandwidth Local Area Network. Not long ago, any network equipment other than 10Mb/s Ethernet Repeaters were extremely expensive. Today 100Mb/s Fast Ethernet Switches cost less than these simple repeaters did only five years ago. Similarly, the cost of Gigabit Ethernet (1000Mb/s) Switches and Network Interface Cards (NICs) is also dropping rapidly. This means that implementation of a network capable of supporting high quality streaming video within a fixed structure has become cheap enough that it will not provide a major impact on the overall cost of providing this service. While it has not become commonplace yet, many private residential homes are being wired up with either 100Mb/s Switched Ethernet or 54Mb/s Wireless Ethernet. Also, many believe it is only a matter of time before both existing and new apartment buildings will be supplied with an internal high-speed network as a matter of course. This would not only supply residents with shared access to a broadband Internet link, but could also be used to provide access to an in-house Video Streaming application.(Cocchi et al., 1993)

- The performance of the Internet itself, both in increased bandwidth and better support for real-time applications. Video Streaming usually requires both sufficient bandwidth and quality-of-service guarantees. While this can be implemented across the entire network, others have shown that it is more technologically and economically feasible to build a distributed Video Streaming application utilising real-time streaming at the edge of the network rather than a Centralised Service that requires real-time functionality throughout the core of the Internet. Of more interest is the recent increase in the bandwidth available in the core of the network. This means that it has become both cheaper to transfer large files across the Internet as well as more likely that these transfers can be completed in a time frame that could support a large scale – globalised or nationwide – video streaming service.(Cornall, 1998; Hsing et al., 1993; Nelson, 1998; Pentland, 1999; Viña et al., 1994)

- There has been a more widespread acceptance and usage of low bit-rate video streaming by the Internet community. As the available bandwidth through the core of the network increased, it improved the performance of a Central Streaming Server Service as long as the required bit-rate for each individual stream was kept low. By keeping the bit-rate low, service providers can guarantee to support a high number of consecutive streams from a single server, while at the same time lowering the possibility of a problem occurring due to insufficient bandwidth on a single link between the server and destination client computer. Unfortunately while these applications provide video quality which is fine for viewing in a 320x200 window on a desktop display, they do not provide the necessary quality for viewing on a 72cm television in a residential living room. Also, these existing services implement the most basic form of a streaming video service, utilising a single central server rather than a number of distributed servers closer to each client, thereby lowering the requirements on the core of the network. Nevertheless, these applications has fostered the acceptance in the minds of Internet users of the use of the Internet in providing a service such as video and audio streaming. By alerting potential customers to this possibility, it is more likely that a commercial service may be subscribed to in the future.(Microsoft, 2002; Apple, 2002a)

Many of the other problems of providing a Streaming Video service have also been solved to a certain degree. The use of distributed servers improves the failsafe capability of the service, while at the same time imposing fewer requirements on the core of the Internet, requiring only that sufficient network bandwidth and quality of service issues be addressed at the edge of the network. A Distributed Server Design also significantly lowers the implementation costs of providing the service, however as shown in Chapter 2, these costs are still too high to enable a true service to be deployed and put into use.(But and Egan, 2002b; But and Egan, 2002a; Chan and Tobagi, 1999; Le, 1998; Jung et al., 2000; Ramarao and Ramamoorthy, 1991; Rangan et al., 1992; Viña et al., 1994; Wu et al., 2002)

Also in Chapter 2, I propose a modification to the basic distributed streaming server design that will significantly lower the implementation cost by avoiding duplication of equipment by competing service providers, while allowing smaller independent film producers and content suppliers to reach a large customer base. Using a distributed server approach will allow a viable commercial service to be built up over a period of time, with each individual addition enabling another group of users to access a high bit-rate streaming video service. We can envisage the concept of an apartment block with a high-speed internal network and a streaming server in the basement providing a high-quality video streaming service to all residents of the apartment block. A second apartment block could provide a similar service to its residents. By linking these two servers over the slower Internet core to a video distributor, we can begin to build a distributed video server arrangement that continuously adds more apartment blocks and their residents to its customer base. Eventually, the addition of private residential services could be considered.(But and Egan, 2002b)

In theory, these approaches can be used to solve the major economic questions facing a viable VoD implementation. In practice however, there are other issues that must be addressed before any commercial streaming video application can become economically viable. One of these issues is providing a service that is attractive to both sets of users – content providers and content consumers. Content consumers may be willing to pay for access to high-quality streaming video, but only if the content provided is of sufficient interest to them. Therefore it becomes important also that content providers are able to provide *interesting* content. In order to provide *interesting* content, providers must placate the concerns of Copyright owners, who have issues which do not form any technical impediment to the implementation of a digital streaming system.(Memon and Wong, 1998)

1.4 Copyright Protection

When we consider a digital video streaming application from a Copyright owners point of view, it becomes obvious that their major concerns are economic rather than technical. There are four parts of a functioning VoD system that are related to Copyright protection of streaming video, outlined in Figure 1-3.

- **Passive protection of content** – Via the use of watermarking, content can be embedded with a non-visible digital signal. This approach does not actively protect against theft of the digital asset but can be used to determine the source of the material should theft occur.(Memon and Wong, 1998; Bao, 2000; Bao et al., 1998; Abdulaziz, 2001)

- **Guaranteed payment for access** – In today's economy, people invest large amounts of money to own the Copyright on a particular media asset. This ownership gives them the right to control all screenings and presentations of that asset. In real terms, this means that a fee is charged for each presentation, whether that be at a cinema screening, video hire/purchase or television broadcast. Monies received due to these screenings form the return on the investment made in purchasing the Copyright – like any other investment, a return is expected by the Copyright owners. This in turn requires a guaranteed payment scheme for any commercially implemented video streaming application. This issue will not be addressed in this thesis, but is an important problem that must be addressed before Copyright owners will make content available for use in commercial VoD applications.(Bridie, 1997; Bridie and Branch, 1998)

- **Encryption of Streaming Video** – This relates strongly to the issue of theft of Copyright material in a digital form. The concept is that all content should be encrypted at all times and decrypted at the user end station prior to playback. The concept of encrypting the video stream is not only to prevent viewing by unauthorised persons, but also to prevent making digital copies of the media stream. In this thesis I will endeavour to provide a possible solution to this problem with the development of an MPEG-1 Video encryption scheme that will function with a range of existing streaming Video Server products. This approach that has been developed can also be applied to the encryption of streaming MPEG-2 video.(Qiao and Nahrstedt, 1996)

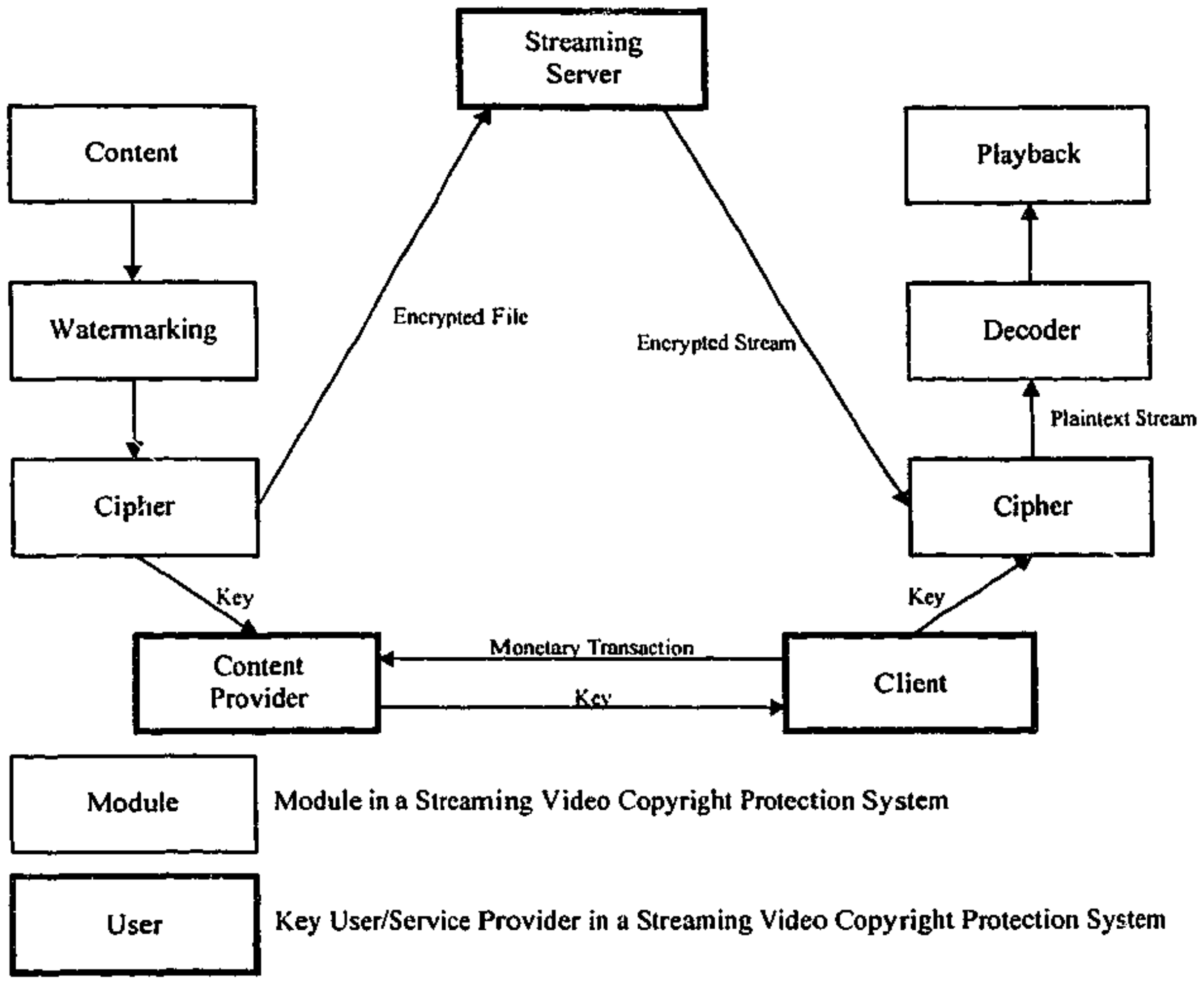


Figure 1-3: Parts of a Copyright Protection Scheme for Streaming Video

- **Key Management** – At the heart of any encryption scheme is the cipher key and its management. As secure as a cipher is in scrambling the plaintext data, if Key Management is poor, then the cipher key, and therefore also the plaintext, can be easily retrieved. While recognising that Key Management is an important aspect of any functional Cipher Protocol, in this thesis I concentrate only on the issue of encryption of the bitstream. The issue of Key Management is left to further research in this area, most likely involving some form of Public Key Infrastructure to securely manage media decryption keys and their transfer across the Internet.(deCarmo, 2000; Menezes et al., 1997; Rivest et al., 1978; Schneier, 1996a; RSA, 1996)

The concerns of the Copyright holders of media assets stem from their rights and expectations, they expect to make a return on their investment in purchasing the Copyright. A major part of this problem is the issue of theft of material. This has been a problem for some time, with users able to make copies of material from legitimate sources – one can record a television broadcast signal, and many VHS recorders have facilities to bypass Macrovision protection to record a video source. While this theft has been going on for some time, it is of relatively minor concern to Copyright owners, since each copy made is analogue, with the quality of each copy degrading in turn. A VHS recording from a DVD source similarly is of lower quality that the original. Even a digital recording made from a DVD source suffers degradation due to the conversion of the digital image to analogue prior to conversion back to digital for recording.(Memon and Wong, 1998; Bao, 2000; Bloom et al., 1999)

Once we consider a digital video streaming application, the original content is in digital form, and it is then transferred in digital form over a publicly accessible network to an end users equipment, where it again exists in digital form. The fear held by many Copyright owners is that a digital copy of this stream can be made – either from the network stream or at the end user equipment. Any digital copy of the original movie asset would be indistinguishable from the original, with absolutely no degradation in picture quality for that or any subsequent copies. What this means in real terms is that many copies of equal quality can be made and then sold by a potential video pirate, making money for themselves while at the same time minimising the return on the investment of the original Copyright owner. This type of theft is serious and is the primary concern of this thesis.

1.5 Structure and Contributions of the Thesis

The layout and content of this thesis is discussed in the following points, indicating what is presented in each of the chapters as well as my specific contributions to the field.

- In Chapter 2 of this thesis I discuss in more detail the requirements of a VoD application. I show why a Monolithic Central Server design is not suitable for large scale VoD applications and go on to show that while a basic Distributed Streaming Server design fixes many problems, it still has technical and economic issues that must be addressed. After considering existing solutions to the concept of a Video Streaming solution, I propose a range of modifications to the Distributed Server design that allows a more economically feasible solution while also enabling smaller players to enter the market. This new system design will decrease implementation costs while at the same time increasing the size of the serviced customer base. I proceed to explore how streaming servers implement different playback modes such as indexed and high-speed playback. Finally, I explore the requirements of Copyright protection as concerns Copyright owners as well as how these concerns relate to end user expectations and requirements. I list a set of requirements for any proposed video encryption algorithm such that it will function in either a standard Distributed Streaming Server design or my proposed Third-Party Distributed Streaming Server design. These requirements are used to show in Chapter 3 that no existing video encryption algorithms are suitable for use in encryption of streaming video.

- Chapter 3 forms an examination of existing MPEG-1 Ciphers. I conclude that none of the existing proposals are suitable for encryption of streaming video. This is to be expected as the primary goal of the algorithm designers was the encryption and protection of stored video. Even so, it is necessary to examine the properties of each algorithm to determine its suitability for streaming video applications.

- In Chapter 4 I propose a novel MPEG-1 Partial Selection Scheme for the purposes of encryption. This scheme is combined with a simple XOR based cipher that offers little protection. The prototype cipher is then tested with a range of streaming server products to ensure that functionality is retained through all the different playback modes. This verifies the concept behind the partial selection scheme and ensures that full functionality is maintained. I demonstrate that although the plaintext is easily retrieved due to the poor security offered by the XOR cipher, the encrypted bitstream is completely obscured – it cannot be passed through a decoder to retrieve parts of the original video or audio content.

- In Chapter 5, I expand upon the concepts developed in Chapter 4, first tying the partial selection scheme to a more secure Stream based cipher, as well as making minor modifications to the partial selection scheme to accommodate the extra requirements of the more secure cipher. Again full functionality is retained through all playback modes. I also examine the effects of network transmission errors and dropped datagrams on encrypted streaming video.

- In Chapter 6 I explore the idea of streaming MPEG-2 video, noting that DVD media is encoded using the MPEG-2 Format and that it would be likely that as network bandwidth further improves, streaming of the even higher bit-rate MPEG-2 bitstream is inevitable. I show how the approach taken to design the MPEG-1 cipher can also be applied to encryption of an MPEG-2 bitstream. These modifications are however, only discussed in broad terms, actual implementation is left to later study.

- The Appendices present an introductory review of both the MPEG-1 bitstream and cryptographic techniques in general. This material can be used to help judge existing video ciphers and in designing any MPEG-1 partial selection scheme. The Appendices also provide a detailed overview of the source code provided on the accompanying CD as well as presenting the full set of experimental results.

1.6 Final Remarks

While much work has been done in the area of streaming video, all of this work has focussed on solving the technical limitations inherent in providing such a service. The success of any networked application however, depends on more than overcoming any technical limitations. It also depends on how useful the service appears to be to all users. As such, now that technical advances have put a streaming VoD service within our technological reach, it has become necessary to consider any non-technical limitations that may hinder the rapid adoption of such a service. In this thesis I identify one such issue: the need to protect the rights of the Copyright owners of media assets that are to be streamed. It is necessary to protect all streaming media against digital theft which would lead to a loss of return for this set of users. Also affected by the same issue is the paying customer, a service where undeterred theft occurs would be bereft of content as Copyright owners withdraw their assets from the service. A lack of assets will inevitably lead to a loss of patronage by paying customers as well. In this thesis I develop a novel MPEG-1 encryption scheme that can be used as part of a Copyright protection scheme for streaming video.

Chapter 2

Copyright Protection of Streaming MPEG Video

In this chapter I explain why there is a requirement for the development of Video Encryption schemes. The issues that are important include not only the perceived need for protection of digital assets on the network, but also how this would be accomplished in a likely Video on Demand (VoD) system. Design considerations of such a system restrict the choices available to us when considering how to protect the content provided by the system. The potential success of any copyright protection scheme increases if the process is generic and not tied to a particular brand of Streaming Server platform.

Another important factor is the perceived need for an encryption system in the first place. In this case, the concerns of copyright holders on digital assets dictate the requirement for encryption of the asset. Economical success of a networked VoD system depends on the availability of material that consumers are willing to pay money to view. Before providing this material, copyright owners need to be assured that their digital material will not be subject to theft. If this assurance cannot be made, then the desired material will not be made available and any VoD implementation will fail due to lack of patronage. Whilst not the only non-technical issue that must be conquered before VoD becomes a reality, the issue of encryption is an important component that must be in place before a true entertainment VoD system can ever be implemented.

The Internet of the future will support improved bandwidth to the client as well as some Quality of Service (QoS) features within the network. High bandwidth will provide extra speed for most existing applications whilst QoS will assist in the provision of both current and new streaming applications. VoD is one application that has been previously proposed but did not succeed, due mainly to both lack of bandwidth and QoS. VoD implementations are going to become increasingly possible to implement in the near future, and their requirements need to be studied. I will examine the different possible designs for VoD systems and show the disadvantages associated with large single server systems as well as distributed server systems owned by a single company. I present a distributed server design operated by multiple parties and explain its advantages over traditional VoD server designs.

Other important issues relating to VoD system design will also be discussed, including both non-technical – that of copyright and protection against digital media theft – and technical – how streaming servers provide advanced playback features such as indexed and high-speed playback and how this impacts on implementation of a copyright protection scheme. Following this, I list the requirements of a Video Encryption system that will meet both the concerns of the copyright owners as well as the issues of supporting a range of different streaming server products.

2.1 Next Generation Internet

The Next Generation Internet is a term that is widely used today, often going by the terminology Internet2. While both of these terms imply that a second worldwide network will be built to replace the current Internet, this is not really the case. Internet2 is a term that embraces the continual improvements to the current Internet that will bring modern network technology slowly into the Internet. Whilst new network and bandwidth management concepts are being trialled in network testbeds throughout different research institutions, those concepts that are found to work well will be integrated into new network products which will then find their way into the existing Internet. The eventual upshot is that there will be small sections throughout the Internet that support these advanced services. These islands of advanced functionality will grow as time progresses and the Internet is progressively replaced with the Next Generation Internet.(Fowler, 1999; Saunders & Master, 1997)

There are a number of important new features being integrated into the Next Generation Internet, including secure data transmission, a potential larger number of connected hosts, better mobility on the network and better support of existing applications such as host naming. Whilst these features are undoubtedly important, the Next Generation Internet will introduce two other concepts that are of far greater importance, Improved Bandwidth and Quality of Service (QoS). These two features together provide the technical possibility to support a host of real-time applications that are not possible using today's technology. The existence of improved bandwidth will allow more data intensive applications to be possible whilst the provision of QoS will allow better traffic management of the network to properly support these applications.(Sikora, 2001)

Video on Demand (VoD) has long been recognised as a potential network application but the concept has never come to fruition due to the limited capability of the existing Internet. The Next Generation Internet promises to overcome the technical limitations that have prevented a true VoD implementation.(Chang, 1998)

2.1.1 Improved Bandwidth

One feature of the Next Generation Internet will be improved bandwidth. This bandwidth will not only be present in the core of the network but also available to the end consumer. Limited bandwidth to the end user has been one of the biggest problems with the current Internet, most commonly the restriction of a 56kb/s line as afforded by most dialup connections means that users cannot obtain data from the Internet at a faster rate than this. A maximum possible bandwidth of 56kb/s provides restrictions on what sort of services a common home user can use. However, the link to the user, commonly called the "last mile", is not the only bandwidth bottleneck within the Internet, large routers within the core of the network also provide bandwidth limitations. Indeed, there is a lot of potential bandwidth in the form of unused fibre optic cable throughout the world, but this bandwidth is limited by the processing throughput that can be handled by the routers that manage the network. The Next Generation Internet will attempt to fix the bandwidth problem in both these areas; firstly by improving throughput and network management by routers to alleviate bandwidth problems in the core

of the network; and secondly through better access technologies such as Digital Subscriber Line (DSL) and cable modem to increase the available bandwidth over the "last mile".

Bandwidth problems in the core of the network are mainly due to the problems that routers have to face. Routers have the responsibility of routing IP packets between physical (or virtual) network ports that they support. Due to the fact that IP packets can be both fragmented and of variable size, most of the routing of packets is done in software. Unfortunately, the amount of Internet traffic has grown at a greater rate than the growth of processing power and routers are falling behind in trying to process Internet traffic. Due to the amount of processing power required to route a single packet, routers can often fall behind when a large number of IP packets arrive simultaneously. This increases the queue length of packets waiting to be processed within the router, which both lowers the overall throughput within the core of the network and decreases the utilisation of the raw bandwidth provided by fibre optic links. Similarly, lengthy queues within the routers can cause IP packets to be dropped when the queue is full, thus forcing a retransmission of the packet, further underutilising the available bandwidth. In essence, the problem in the core of the network is not the lack of bandwidth, but rather its underutilisation due to slow and overloaded IP routers.(Ashmawi et al., 2001; Wang et al., 2002)

Advances in recent times have led to more router pre-processing done in hardware, thereby decreasing the required processing power to route the packets in software. Another recent development has been that of determining IP streams and routing these packets in hardware. This has the effect of keeping the router queue lengths down and decreasing the processing time absorbed by each packet within a router. Smaller queue lengths means that the probability of dropping packets and thus requiring retransmission is decreased and there is less actual bandwidth used to transmit the same information. Also, by decreasing processor requirements for each routed packet, the routers can increase their throughput and thus make better utilisation of the raw bandwidth available in the form of fibre optic links between the routers. Overall, new core routers will offer improved throughput, which will make better use of the raw bandwidth in the Internet.(Teitelbaum et al., 1999)

The largest bottleneck in Internet bandwidth is found in the "last mile" connection to the end user. Whilst large businesses can afford the cost of a high speed connection to service the company, until recently, smaller companies and home users could only connect to the Internet using a modem over a standard telephone service. The maximum bandwidth offered by this link is 56kb/s and even this depends on a clean line between the calling parties. This bandwidth, while adequate for simple information services, does not lend itself to many other applications for which an increase in bandwidth to the user is required. Recently, Cable Modem technologies, using the Hybrid Fibre/Co-axial Network employed in Cable Television Services, and Digital Subscriber Line technologies employed over existing telephone twisted pair, have increased the potential bandwidth available to end users. A Cable Modem connected user can expect to share a connection of approximately 5Mb/s with a potential 10-30 other end users. Since not every user will be utilising that bandwidth concurrently, most users could expect at least 1Mb/s of available bandwidth, a factor of 20 increase over that available via a standard modem connection. Similarly, Digital Subscriber Line users can expect a dedicated bandwidth of anywhere between 2Mb/s and 40 Mb/s, at least a factor of 40 improvement over a standard modem connection. Both of these services also offer a permanently online connection with the existing telephone line being available for normal use. Other high bandwidth access technologies such as satellite and 3G mobile telephony will also become more common. These access technologies are already available, and the usage cost is rapidly approaching an amount that most end users would be willing to pay. Overall, high-bandwidth end-user access technologies are becoming more prevalent and most end users will have a high bandwidth connection to the Internet in the near future. This will encourage the development of new, higher network speed applications.

The Next Generation Internet will provide improved bandwidth to the end user using two different techniques. One of these will be the improvement of routers in the core of the network to provide better bandwidth utilisation and thus the appearance of improved bandwidth. The second technique will improve "last mile" bandwidth through the use of new user access technologies. This improved bandwidth will lead to both existing and new applications being developed or modified to utilise this bandwidth to provide a better service to end users.

2.1.2 Quality Of Service

Quality Of Service (QOS) can be defined as some form of guarantee on the provision of network resources. An example is the guarantee of a minimal average available bandwidth and transmission delay. This is different to the traditional implementation of the Internet, which is based on the concept of a packet switched network. This provides little or no QOS, as each packet is delivered using a best-effort approach, there is no concept of guaranteeing the bandwidth or delivery times for individual datagrams or streams of datagrams. As such, increased network usage leads to congestion which affects the perceived network availability of all users. This is different to a circuit switched network like the phone system, which guarantees bandwidth, delay and jitter for all users, regardless of the number of concurrent calls.(Sikora, 2001; Fowler, 1999)

When the Internet was first developed, the bulk of network traffic was not time-critical and depended more on guaranteed arrival than on QOS issues. As networked applications and computer systems developed, transfer of graphical information became popular, followed by audio and then video delivery. Initially, these were done as file transfers that were then accessed and played back locally at the receiving station. Later, the concept of streaming took place where the time dependent multimedia streams were decoded and played back in real-time as they were received from the network. In these cases, it was necessary that the network had ample available bandwidth to ensure that individual datagrams arrived at the destination on time, a congested network resulted in late delivery and data that was unusable. Other networks, such as ATM, have been developed which support the concept of QOS, therefore allowing terminals to request the required bandwidth and datagram delivery constraints. Resources within the network would then be allocated to individual streams and the required QOS would be guaranteed to the two communicating parties.(Fowler, 1999; Saunders-McMaster, 1997; Branch et al., 1996)

The Resource Reservation Protocol (RSVP) has been developed to try to extend the QOS concept and make it applicable in an existing IP network. RSVP has problems and is not widely implemented, many of these problems have to do with scalability. As one of the features that will be built into the Next Generation Internet, QOS will allow not only better management of existing resources, but also the implementation of applications that require real-time or near real-time data transfer. Multimedia streaming is the most common near real-time data application whereby it is more important that parts of the stream are delivered to the destination within a certain timeframe than whether or not they are delivered correctly. In a QOS guranteed network, the client and server can negotiate their bandwidth requirements with the network which will then ensure that the required network resources are available. Total network resources will be managed better as new streams will be refused QOS guarantees if the resources are not available, existing streames will not suffer service failures in a slightly congested network.(Fowler, 1999; Ashmawi et al., 2001)

Two different techniques are usually applied to implement QOS, Differentiated Services (DiffServ) and Integrated Services (IntServ). The DiffServ approach categorises each packet into one of a series of classes. Each packet is then treated differently by routers based on its class type. In this way, packets of one class can receive better service than packets of a different class. All packets within a single class are treated equally. DiffServ does not understand the concept of a stream and operates equally on all packets regardless of their source and destination. The IntServ approach instead identifies individual data streams through the network and provides a different level of service to each stream. While more true to the concept of QOS, IntServ is a more complex approach due to not only the requirement to identify individual streams, but also maintaining different QOS parameters for each of these streams.(Mohammed, 2002; Ashmawi et al., 2001; Teitelbaum et al., 1999)

Due to scalability issues, a wide scale Next Generation Internet implementation will most likely employ Differentiated Services (DiffServ) QOS in the backbone and Integrated Services (IntServ) QOS in the local area. By employing DiffServ in the backbone, network traffic can be allocated into different service classes, with each different class being allocated different priorities when being routed. This will enable certain types of traffic to traverse the network more quickly than other types, but will not provide wide area guaranteed QOS to individual streams. In the local area, IntServ will be employed to provide individual stream QOS between individual communicating parties. This will allow a local multimedia server to stream to all client in the local area with guaranteed QOS to ensure timely data throughput.

2.1.3 Applications

The coming existence of the Next Generation Internet will lead to a host of new applications being developed to take advantage of these new features. New security features will cause greater confidence in digital monetary transactions (Bozoki, 1999; Aslam, 1998; Group, 1996; IETF, 1998c). Many of these new applications will charge for the provision of a service. Today, a connection to the Internet enables consumer access to an information base, future applications will enable the provision of new services over the public network infrastructure for which consumers will be willing to pay for. The new security features of the Next Generation Internet will ensure both that consumers have confidence in the billing system, and that service providers have confidence in the delivery of their service only to paying customers.

Many of these new applications will not require any other features of the Next Generation Internet other than secure transactions, on-line purchases, non-multimedia information delivery (such as stock market information), and support for registered customers being prime examples. These sorts of applications can be implemented today, examples such as Amazon (www.amazon.com), and Internet Banking applications are starting to come on line. More of these types of applications will appear as improved security becomes commonplace within the Internet. Other applications that take advantage of the improved bandwidth should also be developed, as this bandwidth becomes more generally available. These sorts of applications include those that involve collaborative computing and large data or file transfers, perhaps similar to the SETI at home (setiathome.ssl.berkeley.edu) distributed processing project.

Finally there will be a proliferation of new multimedia applications that will take advantage of both the improved bandwidth and Quality of Service. These applications will be an extension to existing low bit rate multimedia applications, such as Microsoft Media Services and RealVideo, to provide improved quality video at higher bit rates. There will also be a better integration of video services with other media to provide a better multimedia environment. One of the new multimedia applications that will be developed is Video on Demand (VoD) for entertainment purposes. This will be examined in more detail in the next section.

2.1.4 Video On Demand Systems

The concept of Video on Demand (VoD) encompasses the idea of a client selecting a video stream that is stored on a remote server, and then having the server stream or deliver that asset to them. The client can then subsequently control the delivered video – pausing, indexed searches, high-speed playback – while viewing. This entails providing the same sort of functionality that a user would have with the movie being delivered from a DVD or VHS player to their television set, but extending the link such that the source is available somewhere else on the network. (Jain, 1999; Fist, 1994)

VoD has often been proposed as the next "killer" application for the Internet, however, today's Internet cannot support this type of application. High quality video for entertainment purposes requires a lot of available bandwidth, at least 2Mb/s (VHS quality MPEG-1 Video) and preferably up to 8Mb/s (DVD quality MPEG-2 Video), for each individual video stream transmitted across the network. Similarly, the real time nature of networked video delivery requires the presence of a certain Quality Of Service (QOS) in order to guarantee the timely delivery of the video stream over the network. As the development of the Next Generation Internet continues, and these capabilities are progressively introduced into the Internet, the chances of a successful VoD implementation will increase. (Gemmell et al., 1995; Hsing et al., 1993; Little and Venkatesh, 1994; Middleton-Williams, 1993)

Previous work has shown that if the network is adequately dimensioned, it is possible to build a VoD system today. However, the cost of building such a network means that the system will be confined to a local Intranet environment within a single company or local area. Indeed, there are many high quality Video Streaming products available on the market, and the weak link holding back a major implementation is the quality of the available networks. Studies show that none of the currently available distribution networks will allow a large scale system to deliver VoD into the home.(Branch, 1996; Branch and Durran, 1996; Branch and Tonkin, 1997)

The implementation of a VoD system has to meet stringent requirements. These include network requirements, server requirements and client requirements, each of which will be discussed in more detail in the following sections. Whilst server and client requirements can be easily serviced with today's technology, the network requirements either require serious over-dimensioning of the network or future QoS implementations of the Next Generation Internet.

2.1.4.1 Network QoS Requirements

In order to support a VoD system, the network must be capable of supporting individual media streams from each server to each client within the network. A VoD stream, usually compressed using the MPEG-1 or MPEG-2 video compression standard, is a variable bit rate stream with a constant average bit rate. In order to simplify both server design and network management, this stream is usually delivered over the network at the average bit rate and then buffered at the client to reproduce the required variable bit rate input into the decoder. Networked video is a real-time application and it is imperative that data streamed over the network arrive at the client within a timeframe determined by the amount of buffering provided by the client playback application. Late delivery due to network congestion renders the data useless as the moment for displaying that data has already passed.(Gemmell et al., 1995; Fist, 1994; Middleton-Williams, 1993)

The average bit rate of high quality video streams is in the range of 2-8 Mb/s, and we must ensure that the network is capable of maintaining these bit rates during transmission. This is relatively simple where the network is over-dimensioned, the available bit rate will always exceed the required bit rate for streaming. However, in a more reasonably dimensioned network, or one that covers a large area, the problem is not as simple as providing adequate average bandwidth. The network must be managed to provide QOS and guarantees on available bandwidth to individual streams. This QOS must extend from each server to each client that it services and must guarantee the required bandwidth to ensure that the video stream arrives at the client side on time.(Wu et al., 2001; Rangan et al., 1992)

2.1.4.2 Server Requirements

Video on Demand servers are usually workstations equipped with a large disk array and a high speed network connection. These systems run software that manage and stream video from disk storage to network connected clients. The raw disk and network connection bandwidth requirements are easily met with today's technology, much of the complexity of VoD server design rests in the VoD software and its management of available resources. This software must manage access to both disk and network bandwidth so as to be able to serve the maximum number of concurrent streams. Whilst VoD use is not widespread, much work has been performed in developing VoD servers and there are commercial products available that perform this function well.(Microsoft, 2002; Apple, 2002b)

The basic design principles of these systems fall into one of two categories, the first design uses a separate thread of execution for each video stream. This thread is responsible for acquiring data from the disk and transmitting it over the network port. This design offers the best resource utilisation as disk space and network bandwidth can potentially be used to peak potential. The difficulties in this design involve scalability issues of how well the system scales to larger numbers of streams and with the resultant scheduling issues of access to shared resources. The second design also uses a separate thread of execution for each video stream, however in this case, resource utilisation is sacrificed for simplicity of scheduling. The system is set up a for streaming a given bit rate and the disk and network resources are divided up on this principle, each thread is then allocated a fixed time slice to acquire data from the disk and stream it. Whilst scalability is improved, utilisation drops as video streams requiring half the system bandwidth still take up both disk and network resources of a full bit rate stream.(Wu et al., 2001; Ramarao and Ramamoorthy, 1991; Jung et al., 2000)

Finally, streaming video is not a simple matter of streaming a single file at a constant rate. A video server must also be able to provide digital video functionality such as indexed and high-speed playback. To provide this functionality, a server must recognise the different video encoding formats and be able to extract the required information for these playback modes. As such, VoD servers support encoding formats to different degrees and will only allow the installation of valid, encoded video bit streams.(Anderson, 1996; Lin et al., 2001)

2.1.4.3 Client Requirements

A VoD client is typically a set-top box that is connected to the Internet and provides Internet World Wide Web browsing and VoD video playback which is output to a television set. This design allows watching the video for entertainment purposes in a comfortable environment, rather than at a desk in an uncomfortable chair on a computer monitor. There is, however, no reason why a standard PC cannot also be used as a client and such a set-top box will invariably be a specially configured PC running similar software to a desktop computer.(Jain, 1999)

At the client end, the requirements for video playback are minimal. The computer need have an adequate network connection to support the required bit rate and be powerful enough to decode and display the encoded video in real time. The minimum requirement for MPEG-1 video playback is a Pentium 200MMX powered computer with 32 MB of RAM. Obviously, the typical configuration of today's computers far exceeds this requirement and therefore all current computers are capable of playing back MPEG-1 compressed video. MPEG-2 video playback is a slightly different story, the higher bit-rate and quality requires more processing power in order to enable playback. A Pentium III configuration would be a borderline case for successful decoding and full screen playback of an MPEG-2 compressed video stream. With the continuing development in processing power and

memory, future computers will be more than capable of decoding MPEG-2 compressed video.(Anderson, 1990; Chiariglione, 1997)

2.2 Streaming Server Implementation

A VoD Streaming Server implementation involves a hardware platform running specialised software to stream locally stored assets to a number of remote clients over a network connection. This software is responsible for managing the server resources – disk capacity, disk bandwidth, network bandwidth, system bandwidth, system memory, CPU clock cycles – in a way that the maximum number of concurrent streams can be provided subject to available resources. Since each stream is independent, developing software to accomplish this task can be complex, especially when the solution must be scalable to support increased available resources.(Gemmell et al., 1995)

In practice, the most complex issue to manage is bandwidth – correctly managing available disk, system and network bandwidth to maximise the data throughput of the server is challenging. Furthermore, the server must manage each stream independently, a task made more complex in that the current playback mode is selectable by the remote client. Streaming an asset at a constant playback rate is a matter of transferring data from disk to network at the required rate, a simple task made challenging by the question of scalability for large numbers of concurrent streams. In order to support different playback modes, the streaming server software must be more aware of the format of the installed bitstreams.(Gemmell et al., 1995)

In this thesis, I am considering the streaming of an MPEG-1 binary bitstream. In order to offer indexed and high speed playback modes, the Streaming Server software must perform some decoding of the MPEG-1 bitstream. Since the Streaming Server has some knowledge of the bitstream format, this will become an issue when considering Copyright protection of installed assets – any changes made to an installed asset must still be compatible with Streaming Server implementations of these advanced playback modes.

2.2.1 Indexed Playback Mode Implementation

When providing Indexed Playback or Seek functionality, a Streaming Server must be able to locate timestamps within the MPEG-1 System Stream. The bitstream as stored on the server disk must be modified in order to allow the remote client playback application to resynchronise its decoder to playback the new bitstream. While implementation can vary between Streaming Server implementations, invariably one of the following approaches is used:

- **Commence Streaming at a Different Pack** – Information contained within a Pack Header includes a clock reference that indicates when the packet should be passed to the decoder. Since a valid MPEG-1 bitstream consists of a series of Packs followed by the ISO 11172 End Code, where the first Pack in the bitstream must have a System Header following the Pack Header; a new bitstream can be constructed by appending a copy of the original System Header to the end of the Pack Header of the new first Pack in the bitstream. All MPEG-1 decoders will be able to decode and playback this modified bitstream. In effect we are editing the original bitstream by removing a segment at the start of the media asset.(Jayanta et al., 1994; Anderson, 1996; Viña et al., 1994; Wu et al., 2001)

- **Commence Streaming at a Different Packet** – It is possible that the granularity provided by timestamp information within the Pack Headers is too coarse. The format of the Packet Header allows – but does not require – further timestamp information. This information could be more regular and therefore allow more fine-grained indexing information to be retrieved. This approach also requires more work in constructing a valid MPEG-1 bitstream, since a new Pack Header and System Header must be prepended to the new first Packet in the bitstream.(Jayanta et al., 1994; Anderson, 1996; Viña et al., 1994; Wu et al., 2001)

In both cases, it may also be necessary to prepend a new Packet to the start of the bitstream that incorporates an MPEG-1 Video Stream Sequence Header, thereby ensuring that the contained Video Stream is a valid MPEG-1 Video bitstream and can be successfully decoded by all decoders. While this may not be necessary for simple Indexed Playback, a change in playback modes from either of Paused, Fast-Forward or Rewind, would require resetting the decoder at the client workstation and therefore the subsequent presentation of a valid bitstream.

2.2.2 High-Speed Playback Mode Implementation

Not all Streaming Server platforms implement support for high-speed playback. Support for these playback modes is complex and uses valuable resources on the server. Similarly, the most common use of high-speed playback modes is to locate a particular scene, this functionality is already provided with indexed playback (if the scene timestamp is known). For the server platforms that do provide this high-speed playback, it is not implemented by streaming the installed bitstream over the network at a higher bit-rate. This increases both the disk and bandwidth requirements for an individual stream, and will result in a lower number of concurrent streams that can be supported by the server.(Lin et al., 2001; Leditschke and Johnson, 1995; Jayanta et al., 1994)

Bandwidth requirements can be minimised in two ways, one is to not stream the encapsulated Audio Stream as there is no need to perform Audio Playback in a high-speed mode. The second technique is to not stream every Video Frame – as playback occurs in high-speed, it is not necessary to display all frames. If this approach is carefully undertaken, it is possible to stream an MPEG-1 bitstream in a high-speed playback mode with similar bandwidth requirements to streaming at normal playback speed.(Anderson, 1996; Shanableh and Ghanbari, 2001; Frimout et al., 1995)

High-speed streaming is generally performed by streaming a modified MPEG-1 Video Stream (containing only I-Frames). As this stream must be decoded at the client end, this newly created bitstream must be a valid MPEG-1 Video Stream. A properly formatted Video Stream can be constructed using the following steps:

- Existing Group Of Pictures (GOP) Headers do not specify the number of frames contained within the GOP. As such, it is possible to remove any number of frames within the GOP without changing the contents of the GOP Header itself.

- Existing Picture Headers specify only the frame number in presentation order within the GOP. Since the I-Frame always forms the first frame of the GOP its frame number will be 1. As such, it is possible to remove other Pictures from the GOP without changing the contents of the Headers of the remaining Pictures.

- An existing Video Stream is still valid if individual Pictures are removed from it. By removing all bar the first Picture within each GOP, the result is a series of GOPs, each containing a solitary I-Frame.

- When streaming this new bitstream in reverse mode, each GOP should be transmitted in reverse order, as each GOP contains one Picture, each I-Frame will be decoded and displayed in reverse order – simulating the effect of a reverse, high-speed playback mode.

- Indexed High-Speed playback can be achieved by commencing the stream at the start of any of the newly created GOPs. This bitstream needs to be pre-pended by a copy of the Video Stream Sequence Header to create a valid MPEG-1 Video Bitstream.

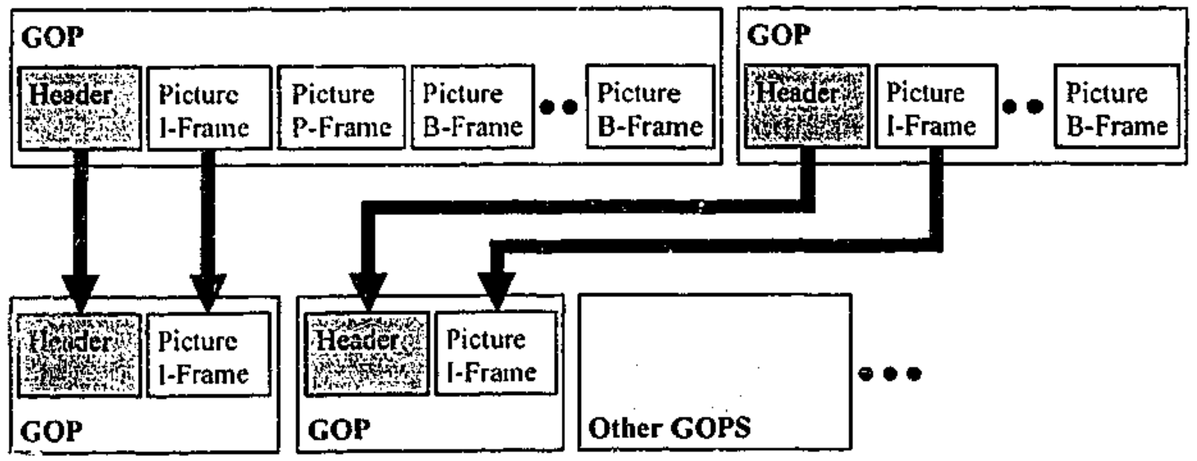


Figure 2-1: Modified MPEG-1 Video Sequence to Implement High-Speed Playback

This approach is shown in Figure 2-1, and is the technique most commonly used to implement high-speed playback modes on existing Streaming Video Servers. When this modified bitstream is retrieved by the client playback application, it can be passed to any existing MPEG-1 Video Decoder which will then display the frames as quickly as they arrive. The typical MPEG-1 Video Stream format is for each GOP to contain 12 frames – 1 I-Frame, 3 P-Frames, and 8 B-Frames. As such, this approach yields a bitstream which will display approximately 1 in every 12 frames of the original video bitstream. This will not yield a high-speed playback rate of 12x, primarily due to the fact that I-Frames usually make up close to half of the data within the original bitstream. The new playback rate will be approximately 2x the original playback speed, while using similar disk and network bandwidth resources at the server.(Frimout et al., 1995)

2.2.3 Installation of Assets

Because streaming server platforms must be able to decode the installed bitstream to a certain level in order to provide advanced streaming functionality, they often check the video file prior to installation to ensure that the bitstream conforms to the required specification. In the case of installation of an MPEG-1 encoded asset, a server would check to ensure that the bitstream conforms to the MPEG-1 System Stream specifications, and that timestamps and Packet Payload information can be retrieved from the file. If the server supported high-speed playback modes, the file would also be checked to ensure that the encoded Video Stream conformed to the MPEG-1 Video Stream specifications, and that individual Pictures can be extracted from the file in order to reconstruct a high-speed bitstream.(Patrick and Moccio, 1998)

In order to improve performance during streaming, indexing and high-speed playback information is usually extracted during installation. As an example, the Silicon Graphics MediaBase 3.1 Streaming Server platform constructs a second high-speed playback Video Stream during installation and saves this new bitstream as a separate file, streaming from this second file when one of the high-speed playback modes is selected. On the other hand, the Apple QuickTime platform does not support high-speed playback, however it does generate a hinted file for installation, containing file offsets for indexed playback purposes.(Patrick and Moccio, 1998)

As most servers at least offer indexed playback functionality, it is essential that the installed MPEG-1 bitstreams conform to the specification so that the server can decode and extract the required information to support these playback modes.

2.3 Design of Video on Demand Systems

As mentioned previously, there are numerous different VoD products available commercially, however developing a VoD system that will service users over a large area such as a nation state or the entire globe, is not as simple as building a single VoD server using available software. In this section I look beyond the concept of a VoD server to the ideas behind a VoD system. This is equivalent to looking at the design and implementation of a large network like the Internet as opposed to a local Intranet. When looking at the design of such a system, it is important to consider how it might scale to service a large number of users over a large area. It is also important to consider the commercial viability of implementing such a system and who might cover the installation costs.

Three implementation concepts are examined in this section:

- Single Server
- Distributed Server
- Multi-Party Distributed Server

2.3.1 Single Server Design

If an organisation is interested in providing a VoD service over the Internet, the first and most obvious solution would be to purchase a powerful workstation equipped with a large disk array and high quality video streaming software. The next step would be to connect this server to the Internet using a high bandwidth connection. As long as the customers intended to receive the streaming video are able to receive the required QOS between themselves and the video server, this system will provide the streaming service adequately. This system design is represented in Figure 2-2, with a single large video server attempting to provide entertainment quality video over a nationwide network. The problems with this system design are fourfold: High Bandwidth Cost; High Upgrade Costs; a Single Point of Failure; and the Ultimate Scalability of the System Design.(But and Egan, 2002a; Branch, 1996; Branch and Tonkin, 1997)

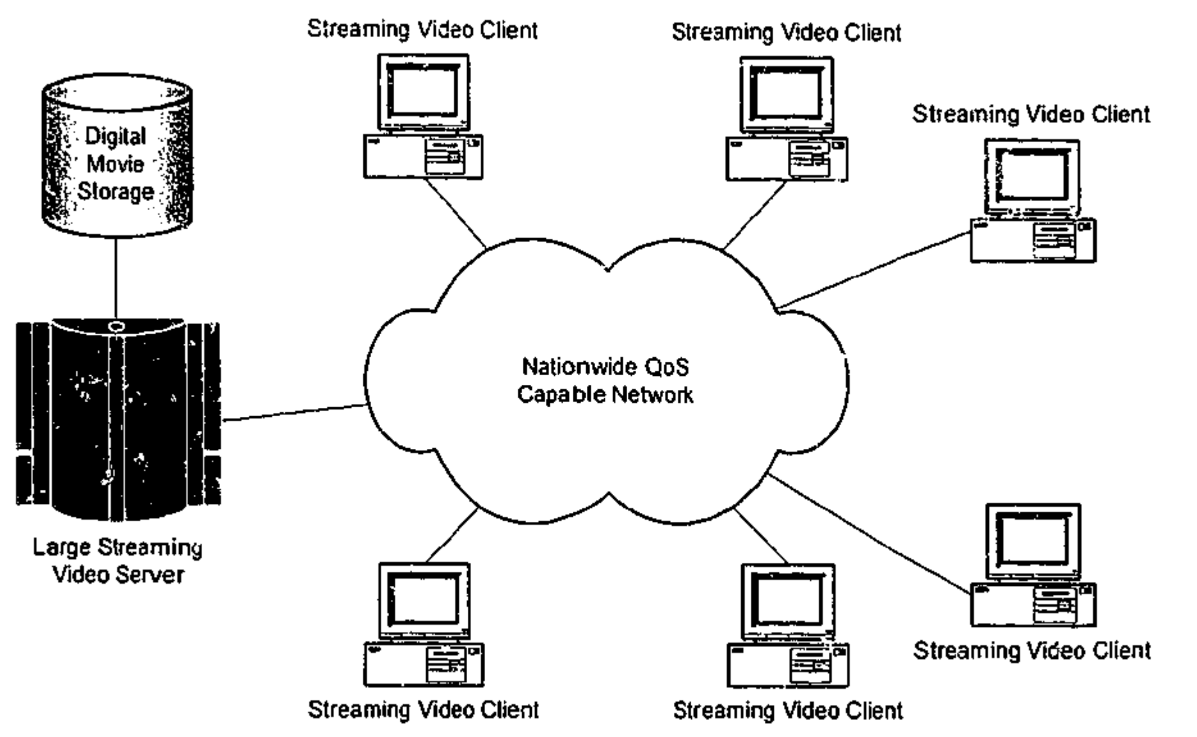


Figure 2-2: Monolithic Single Server VoD System Design

The current major limitation of this system is the high cost of network bandwidth, this involves not only the connection of the server to the Internet, but also the available bandwidth between the server and the customers. Whether the customer or the service provider directly pays the cost, eventually the cost will be passed on to the customer. In the network environment today, high costs of bandwidth limit the size of the customer base. In the Cinemedia Digital Media Library trial, an 8Mb/s broadband connection was purchased between the University site and the Cinemedia central offices. Despite the high cost of this link, only three concurrent video streams could be streamed to the Cinemedia site from the University site. In order to extend the system to cover a larger customer base including a range of secondary schools statewide, the overall cost of supporting the necessary broadband links was prohibitive for a complete service implementation. Bandwidth costs have been steadily decreasing and there is no reason to believe that this trend will not continue. The gradual introduction of the Next Generation Internet will provide increased bandwidth, as well as better management of that bandwidth. This should drive costs cheaper still. As costs decrease, the economic feasibility of introducing a VoD service will become more likely.(Cornall et al., 1999; Cornall, 1998)

The second problem inherent in a single server design are the upgrade costs involved when the customer base exceeds the capabilities of the server. In this case, the single server will need to be replaced with a larger server to service the increased requirements of a larger customer base. These costs could involve more hardware (faster computer, hard disk space), more licenses for the video streaming software, and a more expensive, faster, connection to the Internet. These increased costs are unlikely to scale linearly with the number of customers, thus as the service becomes more popular, the service provider faces increased costs per customer to provide the service. Even multi-processor based systems eventually reach a limit where further expansion becomes expensive.

The third problem with a single server solution is that the system has a single point of failure. Should the single, large video server have a system failure, all customers would face a loss of service, potentially at a large cost to the service provider.

The final, and most severe, problem occurs when considering a system to provide a service to a widespread customer base, such as a nationwide or global service. In this instance, we have a substantial increase in cost of providing the service, which is due to the existence of a single server at a central location. The service provider, and therefore the customer, must pay for a high bandwidth, QOS enabled link between the server and each customer. The cost of both providing and managing a QOS enabled link over a large-scale network is extremely high. As such, it is economically unfeasible to site a central server in a capital city and expect to provide a video streaming service to the entire nation. The high cost of streaming a single movie to a remote location is prohibitive, let alone streaming to multiple customers at this location.(Branch and Tonkin, 1997)

A single server solution is economically unfeasible when considering streaming entertainment quality video to either a large or widespread user base. This effectively limits entrance to VoD service provision to large corporations with a lot of money, smaller film distributors could not afford to provide streaming video to a large user base Given the inherent problems with scaling such a service, a single large VoD server is unlikely to be the design implemented in a real-world VoD implementation. Indeed, many existing lower quality streaming Internet Video developers (such as Real Networks, Microsoft Media Services, etc.) are already developing distributed server and caching systems to overcome these problems inherent in a single server design. These initiatives will undoubtedly be utilised in a real VoD service provision solution.

2.3.2 Distributed Server Design

A distributed VoD server design overcomes many of the limitations inherent in a monolithic single server design. In this configuration, multiple smaller servers are configured at remote locations to service the customer base in the immediate local area, whilst requested videos are transmitted at the available bandwidth without QOS guarantees between streaming video servers on the network. This system design is shown in Figure 2-3, where two large companies are operating competing nationwide VoD services.(Branch et al., 1996; Jung et al., 2000; Ramarao and Ramamoorthy, 1991; Wu et al., 2001)

Since the separate servers only stream video to the local area, the number of customers that they service is lower and therefore cheaper and less powerful hardware can be utilised. Similarly, costs in providing a QOS guaranteed network connection are lower between a client and a local server as compared to a client and a remote server. An increase in the user base within a local area can be handled by either increasing the capacity of the distributed server servicing that area, or by installing a second distributed server to service the same area. Also, increasing the area of coverage by the system is a simple matter of installing a new distributed server in the new remote location. The only major drawback remaining in this design is the large costs involved with a single company implementing a large enough system to cover a wide area. This means that smaller video distributors are locked out of the networked video service industry.

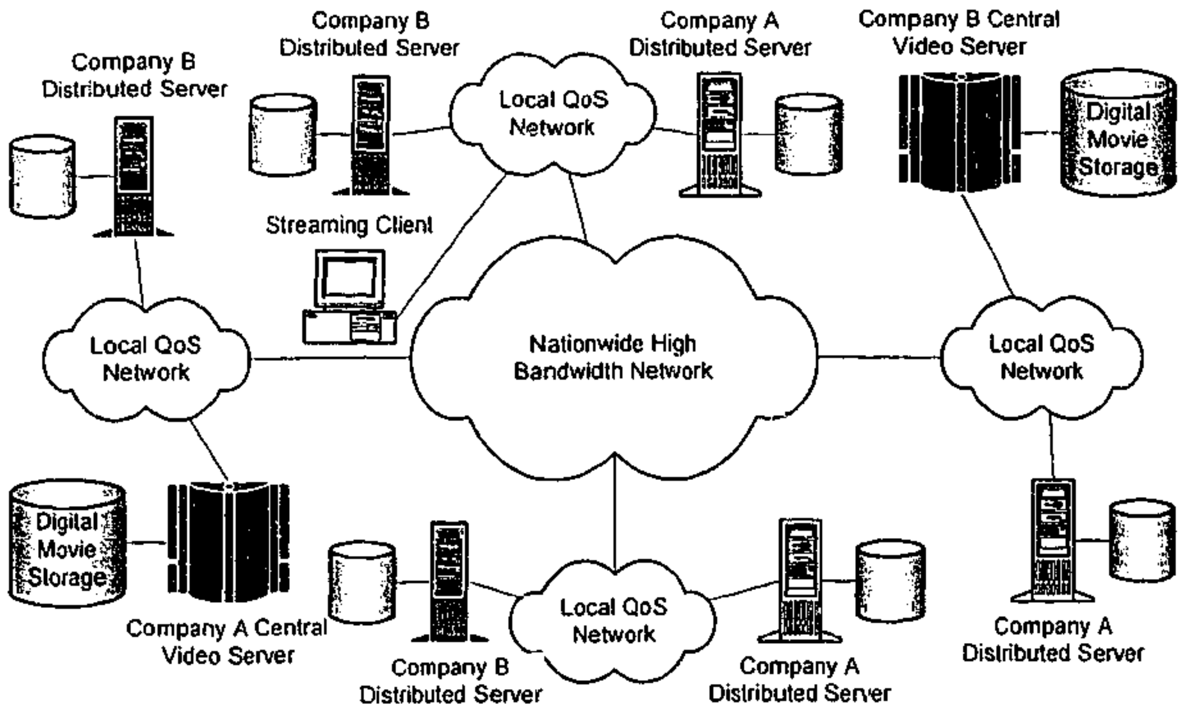


Figure 2-3: Distributed Server VoD System Design

The distributed server design was also trialled in the Cinemedia Digital Media Library project, whereby two smaller servers were installed at remote locations. Assets were now either transferred overnight over slower, cheaper, network connections, or transferred to CD and physically transferred to the remote server. The asset was then installed on the remote server, which was then used to service multiple clients at the site without the need for an expensive broadband connection back to the central server. Indeed, a remote server with a low speed connection back to the central server was able to serve more concurrent streams at a lower cost than the central server could over the broadband connection. This demonstrated the viability of the distributed server design, and that overall costs were lowered. However, even though the video streaming servers were of the same brand, there were still interoperability issues between the servers to manage, as well as increased complexity in asset management and transfer between distributed servers on the network.(Cornall et al., 1999; Egan, 1998; Cornall and Lipton, 1997)

The operating principles of a distributed server solution lead to a better allocation of resources, even if a cheap QOS capable nationwide network becomes available. The main reason for better resource usage lies in the network requirements for video streaming in that each customer requires QOS guarantees between themselves and the streaming server. In a single server design, this guarantee must be provided from the central server site, to each client currently streaming video, no matter their location. Not only does this potentially require more data being transferred over greater distances (for multiple concurrent streams), but that each stream only utilise its required bandwidth for viewing the video. In a distributed server arrangement, video assets are copied between servers at the current available bandwidth (which could be faster than the required streaming bandwidth), and only streamed using QOS guaranteed connections from the end-point server to the customer. As a result, bandwidth management now becomes simpler as it is only required between the distributed streaming server and the client, and nationwide data transfer drops as a single transfer to one remote distributed server will be able to service all potential clients in that area.

When video assets are transferred between distributed servers, it need not be streamed – pre-existing data transfer protocols can be used to perform the transfer. File transfer can be performed at the available bandwidth, if the nationwide backbone has capacity for an 8Mb/s file transfer, then a 2Mb/s encoded video can be transferred and installed on the remote server at four times the speed than if the asset was being streamed. The remote server can immediately commence streaming as it is receiving and installing the video asset. Other clients within the same remote area can also access the video on their local server without a second transmission from the central server. In a situation where there is a popular movie, such as new releases, the asset can even be pre-delivered to the remote distributed servers during off-peak time to take advantage of cheaper network premiums.

The biggest advantage, however, of a distributed server design versus a single server design lies in its scalability. The fact that the system can readily scale to a large size, be it national or global. There are many reasons why a distributed server configuration is more scalable, these are:

- **Lower Network Infrastructure Costs or Requirements** – With a distributed server design, a nationwide fully QOS capable network is not required since streaming will not take place from a remote location. A high bandwidth backbone will be required for content transfer between distributed servers. The design requires only a QOS guaranteed service between the client and the closest streaming server, network costs are minimised due to lower QOS requirements.

- **Server Infrastructure Costs** – The service provider no longer needs to purchase an extremely large system to service a large number of clients. A file server with a large disk complement can be utilised as a central store and smaller, cheaper video servers can be distributed around the network. Also, in the event of a system failure of a distributed server, the load can be shifted to another nearby distributed server without loss of service to other customers. Replacement of a failed server is also simplified.

- **Growth of Customer Base** – If the customer base grows, the system can be scaled to support these customers by either the addition of further distributed servers or upgrading an existing distributed server to a larger model.

- **Extending Range of Service** – In a single server design, extending the range of service provided involved ensuring that a QOS capable network existed between the central server and any new customers. In a distributed server arrangement, the solution requires a local QOS capable network in the new remote area, the addition of a single distributed server to this area, and adequate average bandwidth from the central server to the new distributed server.

There are still a number of problems with the distributed server design. These involve the management of such a system, and the overall implementation costs. When looking at a distributed server design as opposed to a single server design, we have increased the complexity of the implementation, and therefore the management of such a large system. In a single server design, asset management involved solely keeping track of the assets installed on the server, in a distributed server design, we must keep track of assets installed on each server as well as the location and status of each distributed server. Whilst this added complexity will certainly make a distributed server design more difficult to implement, it is an obstacle that must be overcome in order to provide a scalable VoD service.(Bridie, 1997; Bridie and Branch, 1998) Indeed, these solutions are already being integrated into existing lower quality streaming video server products (Apple QuickTime, Windows Media Services) as these products are moving from a single server design to a distributed server design. Even so, these solutions involve a single brand of video streaming software that communicates using proprietary protocols and therefore forces system operators to select a single brand name to minimise interoperability issues.(Apple, 2002a; Apple, 2002b; Microsoft, 2002)

The other problem is the overall cost required in setting up a nationwide VoD service using the distributed server model. Whilst the costs involved are still lower than those required to run a working single server solution, they are still prohibitive for all but the largest companies. Also, if two competing companies set up such a system, there would be a large duplication of hardware outlay in order to for both companies to provide a similar service to all users nationwide.

A distributed server design will readily scale to provide not only a nationwide, but also a global service if a capable Internet backbone is available. This service could easily reach all customers and can handle an influx of new users. However the cost of providing this service is still prohibitive for smaller companies to consider and effectively limits access to the industry to large companies. Competing VoD services would lead to duplication of equipment within the network and possibly some areas being serviced by only one company. So a distributed server design has removed the scalability issues inherent in a single server design, but hasn't addressed the issue of costs and the economic feasibility of implementing such a system.

2.3.3 Multi-Party Distributed Server Design

As discussed in the previous section, a distributed VoD system design is one that is inherently scalable and can be used to provide a wide area service to a large number or customers. The failings of such a design however, are the high costs involved in implementing such a system, as well as the duplication of costs with competing services. The obvious question to ask is if the distributed server design can be somehow modified to lower the implementation cost to parties interested in setting up a VoD service.

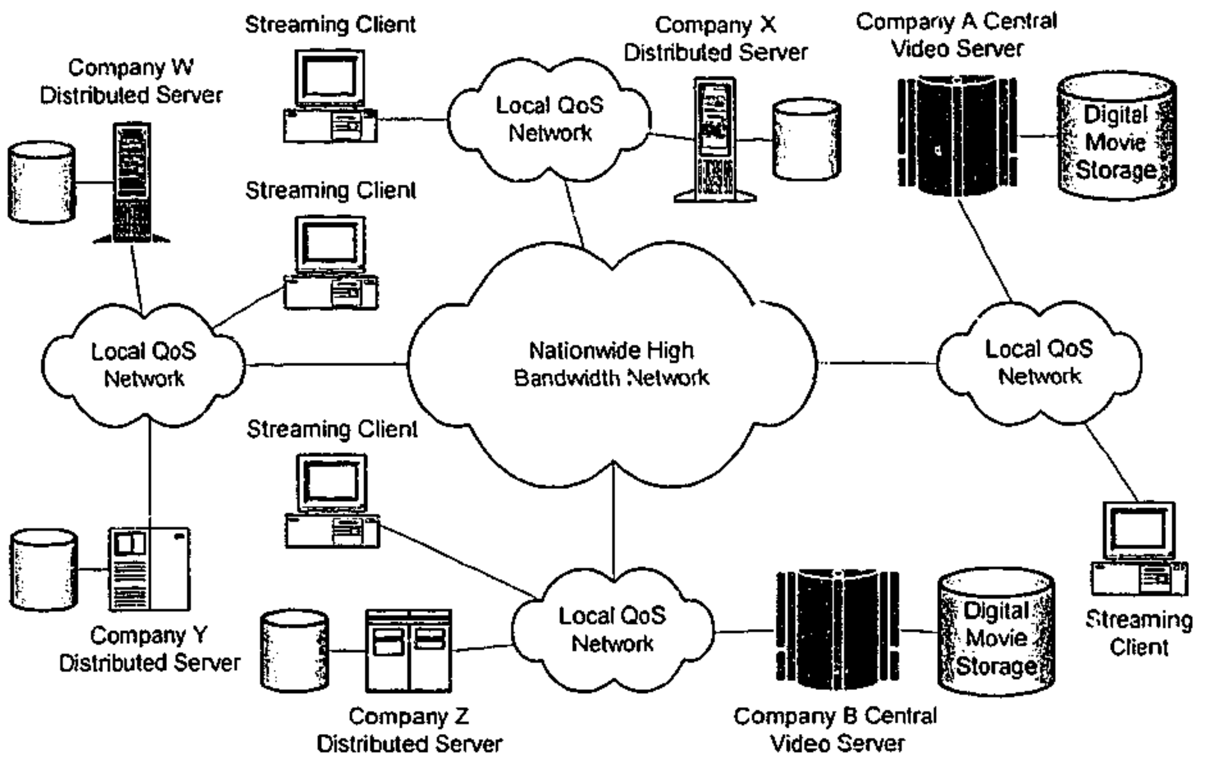


Figure 2-4: Multi-Party Distributed Server VoD System

This can be achieved by taking the concept of the distributed server solution and sub-dividing it into smaller components, allowing individual companies to operate and manage these components separately. In this situation, the implementation cost is shared amongst a large number of players, thereby reducing the costs involved in providing a service.(But and Egan, 2002b)

The extension to the distributed VoD system design can best be described as a multi-party distributed server system, shown in Figure 2-4. In this arrangement, we allow third party, neither distributor nor customer, companies to own and operate the individual distributed video servers. These small third party operators have no direct contact with the customer, instead the customer deals directly with the video distributor, which then transmits the video to a third party operated distributed server for final streaming to the customer. The third party operator then bills the video distributor for usage of their server; who in turn bills the customer directly. These third party operators can be seen as

providing a streaming service to video distributors, offering streaming capabilities to a local area for a remote central media store.

In operation, there are three types of companies that are involved in a video streaming system, along with a central distributed video server network manager:

- **Video Distributor** – This entity would set up a central server that stores a copy of each of the assets owned by that company, this server is primarily a file server and would not be required to provide any streaming services. As for the previous model, the major cost involved is setting up the server to maintain a copy of all available assets.

- **Streaming Service Provider** – This entity is responsible for the streaming the video to the customer. This company would own and maintain a smaller video server whose content is dictated by the customers and the central servers. The customer conducts business with the distributor who then utilises the streaming service provider to help deliver the video to the customer.

- **System Manager** – This entity is required to keep tabs on all distributors and streaming service providers available on the network. This would enable distributors to get information on which streaming servers service particular areas and assist in management of the large network.

This involvement of third party companies in management of distributed servers allows small companies to enter the video streaming industry, even if they do not own any content. This design also allows competition between small streaming companies, an example of this can be demonstrated in a rural town. A small business in this town could install and operate a distributed server, streaming video managed by a major distributor in a capital city to the residents of the town. If business proves profitable, a second small business may decide to start up a similar business in competition to the first, which will invariably lead to lower streaming charges and therefore lower costs to the end user. In this situation, an end user would have no knowledge of the streaming company, nor of how many distributed video-streaming providers are servicing their area. A small business could form, register their server in the nationwide video server network, and provide a streaming service. Similarly, if the business proves not to be profitable, they could remove their server from the nationwide server network and close the business.

An obvious advantage of the multi-party distributed server design is that since the distributed servers are managed by a number of different operators, a video distributor who owns content can also become an online provider with little initial outlay. This is mainly because the distributed server network is already in place. In this situation, a video distributor need only set up a central server containing their content and register this server with the distributed server network. A remote client can now request the video, which would be delivered to an appropriate distributed server before being streamed. The advantage of this is that small video distribution companies can now be involved in offering content online, as they no longer need to finance a nationwide distribution network. Either a large video hire chain store, or a small independent film producer could be involved with digital streaming. In the example of an independent film producer, they are unlikely to have more than a handful of video assets. They could install a small server containing the content that they own. A customer could now request to view this video, the small server would then locate the closest available streaming server to the customer, and transfer the movie to this server for streaming, if the distributed server already has a copy of the video, it can be streamed immediately.

The greatest advantage of this design is that it has lost none of the advantages of the general distributed video server design, whilst removing its major disadvantage. The cost of installing the video service has been spread over a number of companies. The overall cost is reduced as the cost and effort of installing distributed servers is not duplicated by competitors. Smaller distributors can now afford to provide a nationwide video streaming service by utilising an exiting network of distributed servers. Because the cost of implementation is spread over a number of companies, this network is more economically feasible to construct, and can even be constructed in phases. As a new distributed server comes online, the range of participating video distributors is extended to cover the region serviced by the new server. Increased competition amongst managers of the distributed streaming servers will lead to lower charges to the distributors, this in turn leads to a lower cost to the customer utilising the service.

The multi-party distributed VoD service design does have some drawbacks, distributed server operators will be competing against each other and using different video streaming products. This will serve to emphasize interoperability issues and will require a standard protocol for communication between servers provided by different companies, but the extra complexity inherent in the multi-party distributed server design cannot be completely eliminated.

Another new issue to consider is the trust factor inherent in utilising third party delivery systems. Since the streaming video servers are now owned and operated by a third party to the distributor/client relationship, the fear of theft of a digital asset whilst installed on a third party server becomes real. The issue of copyright ownership and protection of digital video assets is very important and should not be taken lightly. The use of public network infrastructure already means that digital video material is subject to theft whilst in transit across the network and the potential use of public or third party streaming servers means that digital video material is also subject to theft whilst stored on these servers. As such, we have introduced a new security issue that must be dealt with to the satisfaction of copyright owners before such a service could be legally implemented.

The multi-party distributed video server solution lowers the overall implementation cost and spreads this cost over a number of players, making implementation more economically feasible. This improvement has come at the expense of increasing the complexity of managing the system as well as decreasing the security and protection of digital video assets. There is a very real requirement to develop a solution to this security problem.

2.3.4 Non-Technical Issues

The two most important non-technical issues are payment for access to streaming video, and copyright concerns. The first issue is an e-commerce issue that must protect both the customer and distributor in regards to monetary transactions, the second issue involves the security of the digital video assets and their protection against theft.

2.3.4.1 Payment for Access Privileges

If a company is considering introducing a nationwide VoD service, even using the multi-party distributed server model, the overall cost of implementing this service is extremely high. These costs can be broken into three types, the first being the initial hardware installation costs, which even shared over a number of companies, are still high. The second major cost is the ongoing maintenance and bandwidth costs. These recurring costs are required to ensure that the service is always available to customers. The final major cost is to the copyright owners of the video to be streamed, the copyright holders have an investment in a film in particular and expect a return on that film through royalties each time it is screened.(Memon and Wong, 1998; Bloom et al., 1999; Little and Venkatesh, 1994; Fist, 1994)

All companies involved are going to want to see a return on their investment, and therefore it is imperative that customers are charged for utilising the service. Similarly, due to the online nature of the service, it is highly likely that online payment will be the preferred method of billing customers. If customers are not confident in the online billing system, then the service will be under-utilised due to insecurity in money transfer leading to the fear of being over-charged. If service providers are not confident, there is the possibility of free access to video assets and therefore no return on their investment. This eCommerce issue is exactly the same for online video as for all other online money transfers and the solutions being developed for other online business enterprises will be directly applicable to the VoD industry.(Aslam, 1998)

2.3.4.2 Copyright Concerns

Copyright holders pay a large amount of money to purchase the screening rights to a particular asset and, as with all investments, they demand a return on that investment through royalties each time the material is screened. The problem with the concept of a VoD service is that the video is streamed over the Internet, a public network infrastructure, and stored on potentially insecure servers, making the video material subject to theft. If this occurs, copyright owners lose potential returns on their assets. This is less of an issue in non-digital systems, due to the deterioration in quality when copies are made. Theft of this material from an online service will result in the loss of a perfect digital copy of the asset, which is not subject to quality degradation when making multiple further copies.(Memon and Wong, 1998; Bloom et al., 1999)

As such, protection of copyright is critical for providing a VoD service. If copyright holders are not confident that their material will not be subject to digital theft, then they will not make this material available in the provision of an online service. If there is a lack of material available, the service will not be utilised as heavily by potential customers and will eventually fail due to lack of customer interest. Copyright holders must be satisfied that their material will be protected before releasing their material for online distribution.

2.4 Copyright

In the previous section I briefly discussed the issue of eCommerce and how it related to the provision of VoD service. One of the eCommerce or monetary issues of the service providers is the payment of royalties to copyright holders of the video being streamed.

Copyright ownership provides exclusive rights on a work to make and distribute copies, prepare derivative works, and to perform and display the work in public. In terms of streaming video content over a network, it entitles only the Copyright owner to make modifications to the content and to distribute it to other users of the network. Whether this content is distributed freely or at cost is up to the Copyright owner.(Memon and Wong, 1998; Abdulaziz, 2001)

Ownership of Copyright is an economic investment. Having purchased the Copyright, the owner expects a return, commonly called royalty fees, on that investment. Returns on Copyright ownership of music or audio recordings are paid by a percentage of every purchase of that recording, or payment of a fee when that recording is broadcast by a third party such as a radio station. Returns on Copyright ownership of video assets are paid in a similar way, either a percentage of a purchase or fee payment for each broadcast.

The concept of delivery of audio or video content over public access networks creates new problems in the area of Copyright. New issues involve collection of royalty fees, protection of content against digital theft, and which parties will be responsible for ensuring that these requirements are met. This challenges not the existing legal view on Copyright, but rather introduces technical challenges of how to apply existing Copyright law to the digital domain.(Bloom et al., 1999; Bao, 2000; Bao et al., 1998)

2.4.1 Digital Rights

The issue of copyright was brought to the attention of CTIE during a VoD trial project including Cinemedia(The State Film Centre of Victoria) and Silicon Graphics. In this project, a VoD system was designed to stream educational curriculum content to secondary schools within the state. The issue was discovered as Cinemedia began to negotiate with copyright owners to obtain permission to use their material for this trial. One complex issue was where the original copyright did not mention digital rights for the material in question, as such some copyright owners were unable to release their material to the trial. Whilst this issue could become a problem, especially when considering the availability of older or historically relevant material, this is only minor in comparison with the concerns of copyright holders over the security of their assets. In fact, it is this initial concern and problem that led to the research evident in this thesis project. Copyright holders pay a good deal of money in order to own the copyright on a particular video asset, and therefore expect a return on this investment. This expectation is reasonable and the return is generally paid by broadcasters (advertisers in the case of free-to-air television) or the public (in the case of video hire). In this situation, master copies of the asset are only available to trusted organisations. The copy played to the public is of lower quality and subsequent reproductions of lower quality still. As such, the possibility of theft is low, and the potential return for theft of low quality copies is insignificant.(Bridie and Branch, 1998; Branch, 1996; Branch and Tonkin, 1997)

With digital streaming video, we have a high quality digital copy being streamed to the customer and any reproductions will be of the same high quality. Transmission over a public network can easily lead to third party theft of a high quality digital copy of the video asset. Given that the issue of online monetary transactions has not been solved, copyright owners are concerned about the uncertainty of return on their investment as well as the theft of high quality digital copies of their video assets. While the act of stealing a digital copy from the network is not as easy as many copyright owners perceive, it can be done and therefore the concerns over theft are valid concerns.(Hsing et al., 1993)

Copyright holders are generally not experts in Internet technologies. They often perceive it to be a medium whereby any material placed online is freely available to all. Theft of material from the Internet, while not simple, is possible. Insecure transmission of digital data over a third party network is always open to theft by an authorised administrator of that network. The only way to reassure copyright owners that their asset will be protected is if it is encrypted while in transit over public networks and only decrypted directly before being played back.

Protection of digital material against theft is extremely important. Copyright owners need to be assured of two things, firstly that their material will be safe from theft by a third party, and secondly the flow of money from the consumer is both well defined and secure. Until these issues are addressed, copyright owners will withhold content, thereby ensuring the commercial failure of any video streaming service.

2.4.2 Digital Theft

Having discussed the importance to copyright holders of protection of digital video material from theft, it is imperative to look at the vulnerable points of VoD system design to see where this theft might occur. Only then can we design a set of security requirements that will protect the digital material.

2.4.2.1 Theft from the Central Server

One of the potential weak links in distributed server system designs is the security present on the central server. In both cases, this server is owned and managed by the distributor of the digital asset. The task of the server is to accept requests for video from the customer and then to transfer and install the video on a streaming server local to the customer. This system is vulnerable to theft of its assets by a hacker gaining unauthorised access to the server and to the digital assets installed on the server. If the server is not secure, gaining access may be relatively easy, at this point it would be a simple matter of digitally retrieving the video file from the server.

There are two techniques that could be utilised to correct this potential problem; the first of these is to improve the security of the server itself. The reconfiguration of the server to ensure against unauthorised access could be done using existing security measures such as terminating unnecessary services on the server (e.g. telnet) and/or utilising better security measures of the operating system. The second solution to the problem is to store the videos on the central server in an encrypted form. This does not secure against theft of the encrypted asset but, if the encryption is secure, ensures that the digital asset is secure since the hacker will be unable to retrieve the unencrypted format of the video file. The unencrypted files could be stored on another server protected by a firewall that is responsible for encrypting and placing the files on the public server. Either of these two solutions, or both used together, would alleviate the major concern of theft of a digital video asset from a server maintained by a trusted party.

2.4.2.2 Theft from Streaming Servers

If we consider the role of the streaming server in the distributed video server designs we can concentrate on the issue of how a video is streamed to a paying customer. The video asset is installed to remote distributed servers for local streaming to the customer. In the case of a simple distributed server design, it is again imperative that these servers be protected against attack and that the video be installed in encrypted form in case an attack is successful.

In the case of the multi-party distributed server design, copyright protected content is installed on a distributed server – owned and operated by a third party to the transaction – for streaming purposes. In a true multi-party network, these operators may not be trustworthy. Similarly, these servers may not be secure. The only solution is to ensure that digital assets installed on a third party server are in an encrypted form. Since a multi-party distributed server design may involve a range of different streaming server products, any encryption technique must function with a large range of video servers.

2.4.2.3 Theft in Transit

Since the Internet is a public network infrastructure, many parties have access to the data as it is transmitted across the network. Because the network is not private and the digital data cannot be physically protected from other interested parties, the data is open to theft whilst it is within the network.

Theft of digital video whilst in transit can occur by listening to network packets as they traverse the network and storing those of interest. The data within the packets can then be reassembled into the original digital file. In the case of a VoD service, the data can be stolen in transit in one of two ways, the first of these is to listen into and record the file transfer from the central server to the local

streaming server. The second is to listen into a record the network video stream from the local streaming server to the customer.

This form of theft is not as simple as it seems, especially when capturing the stream from the local streaming server to the customer. This is because the video data is usually streamed using a proprietary protocol, once a hacker has obtained and stored the packets from the network, he still must decode the streaming protocol to reconstruct the original digital asset. The solution to this problem is again one of encryption, and that is to ensure that every time the digital asset is transferred across the network, it is in an encrypted form. In this way, even if the stream were intercepted and the encrypted file retrieved, the digital asset is secure since the hacker will be unable to retrieve the unencrypted format of the video file.

2.4.2.4 Client Theft

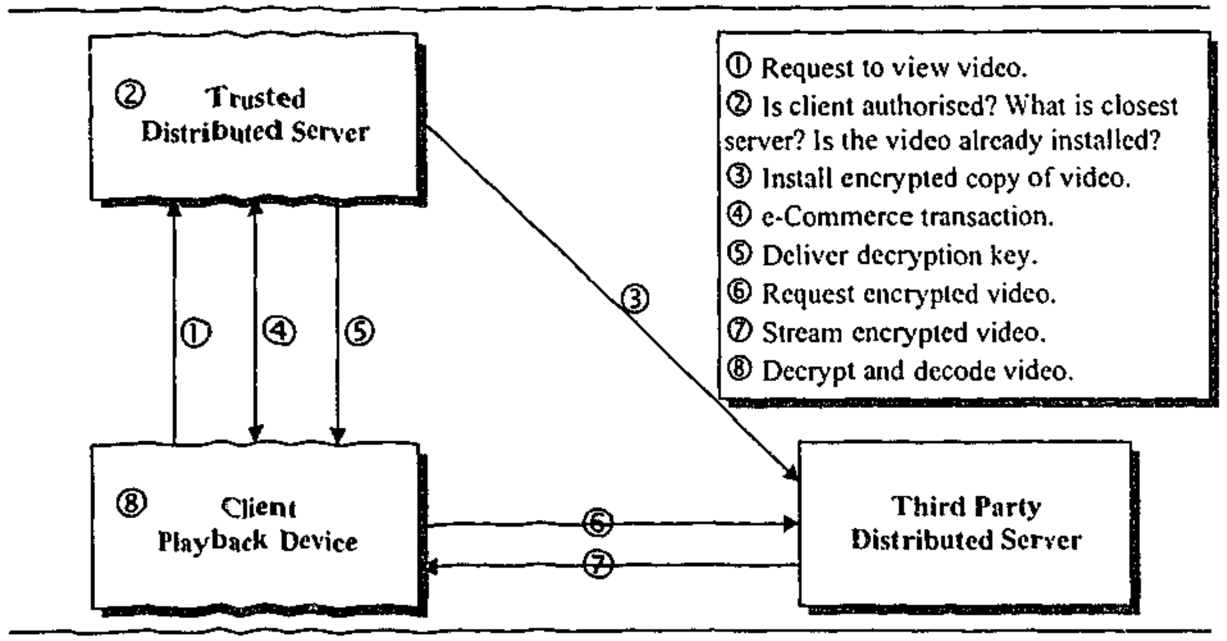
Finally, digital video material can be stolen at the client end. This can occur by paying for legal delivery of a video stream, and then making a digital copy of the stream as it is retrieved for playback. As the content is streamed in a digital form, it is a relatively simple matter for a programmer to capture and save to disk the video stream before it is decoded and displayed. This copy could then be used for illegal gain. The solution in this case also involves encryption. If the video is streamed to the client in an encrypted form, then the process of decryption and decoding for viewing purposes can be tightly bound into a video playback application. This ensures that the client can see either the encrypted video stream or the completely decrypted and decoded video stream only.

2.5 Video Encryption Requirements

Given the likelihood that a commercial video streaming service will involve either a simple distributed server design, or multi-party distributed server design, and taking into account the copyright and digital rights issues mentioned in the previous section, it is obvious that a form of encryption would need to be developed that would function within these system designs. The security provided by this protection scheme must protect the asset from theft during transit over the network, whilst stored on a secured server, whilst stored on a third party streaming server and whilst being decoded and displayed at the consumer end. From a users perspective, we must also ensure that none of the conveniences of digital video are lost. This means that the functionality of indexed playback, high-speed playback, and clear still frame must all remain intact and function. Also, the encryption must be completely unobtrusive to the customer, the security must be available without providing a public face to the potential customer. In order for a new technology to be accepted by the customer base, it must not only integrate seamlessly with existing technology, but also require no extra effort by the customer to use it. In this section I list the requirements of the ideal digital media encryption scheme, beginning with those imposed by copyright owners needs, followed by those imposed by the third party distributed server design and existing streaming server products, and finally the requirements imposed by the clients.

2.5.1 Copyright Owner Requirements

The copyright owner providing the digital video will impose a set of requirements on a video streaming service. If these requirements are met, they will be satisfied that their investment is properly protected. The first requirement imposed is that of customer authentication and payment, this ensures the proper flow of monies back to the copyright owner. This problem could be solved using an appropriate e-commerce model. While the provision of customer authentication, including appropriate charging and key management, for video playback are absolutely essential in the provision of a secure networked video streaming system, it can be regarded as an adjunct to the encryption algorithm itself. This application would be solely responsible for securely delivering the correct decryption key to an authorised customer, which would then utilise this key to decrypt the encrypted video stream. As noted in many encryption technology texts, the issue of key management using Public Key Encryption Schemes is a complex issue in itself.(Denning, 1983; Menezes et al., 1997; Schneier, 1996a) Similarly, the issue of the video encryption algorithm is also complex. This thesis examines the encryption problem, leaving the key management/e-commerce issues aside. For reference, the basic Key Management procedure is outlined in Figure 2-5.



① Request to view video.
② Is client authorised? What is closest server? Is the video already installed?
③ Install encrypted copy of video.
④ e-Commerce transaction.
⑤ Deliver decryption key.
⑥ Request encrypted video.
⑦ Stream encrypted video.
⑧ Decrypt and decode video.

Figure 2-5: Customer Authentication Procedure

The second requirement imposed by copyright owners will be that of active protection of their asset via encryption. They will demand that the digital video be encrypted at all times whilst it is in transit over a public network. They will also require that the video be in an encrypted form whilst stored in any non-secure or non-trusted device on the network. This requirement imposes a restriction that all the streaming servers must be able to stream encrypted video. Finally, the copyright owners will require that the video is only decrypted at the client device and then done such that the client has no access to the decrypted video stream.(Bridie and Branch, 1998)

Copyright owner concerns impose one restriction on the video streaming system and one restriction on the design of the video encryption algorithm. The entire online video service must

provide proper customer authentication and payment facilities, this protects both the copyright owner and the client from monetary theft. Also, it is required that the video be in an encrypted state from the time it leaves the trusted central server till the time the video stream is decrypted and decoded at the client playback device.

2.5.2 Distributed Server Arrangement Requirements

The potential presence of non-trused third party streaming servers within the network imposes a unique restrivtion on the encryption scheme, the digital video must be encrypted whilst it is stored on third party servers and then delivered from these third party servers in encrypted form to the customer. In this case, the customer must communicate with the trusted server owned by the distributor in order to obtain authorisation and the key required to decrypt and view the video whilst the video is delivered more efficiently from a local third party server. The operators of the third party servers now have encrypted video installed on their servers that cannot be decoded or viewed. If the copy of the video is compromised whilst stored on the third party server, the thief will not be able to make use of it since the video is stored in encrypted form. Unprotected digital copies exist only on a secure distributor server and in a temporary form whilst being decrypted at the customer end.

The potential for a wide range of servers – including different hardware, operating system and streaming server software – means that the encrypted digital video file must be compatible with all existing and future video server systems. While it is impossible to predict exactly how future video server systems will be built, it is possible to look at existing systems and extrapolate. An interesting point is that while some systems will stream any binary file, many require a file that fits the digital video specifications. As such, if the chosen encryption scheme produces an encrypted digital video file that has the same format as an unencrypted file, we can be reasonably certain that the encrypted file can be installed on a server which has no knowledge of the video encryption scheme. If, however, the chosen encryption scheme produces an encrypted digital video file that requires the video server to have specific knowledge of the encryption scheme itself, then we are enforcing restrictions on third party video server owners and developers of video server technologies. A system without these restrictions ensures that all existing video server software that serves known video formats can be used to serve the encrypted file. In order to minimise inconvenience to server operators, the encrypted video file must install on all video server platforms.

Use of a multi-party distributed server architecture imposes three restrictions on the design of the video encryption algorithm. These restrictions enforce that the video must be installed on all servers, both central servers and third party operated streaming servers, in an encrypted form. Also, it is essential that the video encryption algorithm used make no special requirements on the construction of streaming servers. Acceptance of a new system is more widespread if there are minimal requirements made on existing products. This requirement also benefits implementers of a simple distributed server architecture, allowing more freedom in choosing streaming server platforms and not tying them to a single platform.

2.5.3 Streaming Server Requirements

Video streaming servers often support playback modes such as seek, fast forward and fast rewind. Compatibility with these functions creates its own requirements. When looking at the seek function, the video server is provided with a time index by the client playback application, the video server must then stop streaming the video and recommence streaming at the supplied time index. This implies that the video server must be able to index into the compressed video file. When dealing with unencrypted video, this is not problem as the video bit-rate can be used to locate the approximate file position, and plentiful time stamps throughout the file allow for fine tune indexing. If the same server is to support the same seek functionality on an encrypted file, then the encrypted file must contain the same time stamps within its binary stream to allow the server to locate particular time indexes. As a result the decryption process must be able to be resynchronised to support time indexing.(Fist, 1994; Gemmell et al., 1995)

High-speed playback modes, such as fast forward and fast rewind, also need consideration. In order to minimise network bandwidth usage, existing video streaming servers skip frames of a video stream rather than serve all of the frames at a faster rate. This technique saves bandwidth as only some of the compressed video data is transmitted and the audio stream is omitted entirely. This is usually done by streaming only the I-frames within the compressed video file as these frames stand alone and do not require subsequent frames to be present before decoding. If the same functionality is to be supported with encrypted video, then the video server must be able to locate the individual frames within the encrypted file, and to determine what type of frames they are. In this way, the server can still locate individual encrypted frames and stream them to the client. Similarly, the decryption process must be able to resynchronise to decrypt individual frames.(Anderson, 1996; Shanableh and Ghanbari, 2001; Wu et al., 2001)

The construction of video streaming servers place particular demands on the video encryption algorithm. Indexed and high speed playback force the encrypted video file to adhere to certain parts of the video compression standard, allowing it to be both recognised as a valid compressed video file by the server, and to allow the server to provide features such as time indexing and frame skipping. These issues will be explored in more depth in Chapter 4.

2.5.4 Client Requirements

The client viewing the video and the platform he/she is using to playback a networked video asset also imposes some limitations on the video encryption scheme chosen. The first issue that must be resolved is that of processing power. Processing power at the customer end is limited and a certain amount of CPU power is required merely to decode the compressed digital video stream for playback. There may be little CPU power remaining to decrypt the encrypted video stream. As such, the first requirement imposed on the chosen encryption algorithm is that decryption at the client end require minimal CPU cycles.

As with streaming servers, the encryption scheme must be compatible with all existing and future video decoders. As will be shown in Chapter 3, some existing video encryption techniques require specialised decoders that decode and decrypt in a single step. This automatically precludes the use of these algorithms with third party decoders that do not provide this function. By selecting an encryption algorithm that will both use minimal CPU load and is able to integrate with existing decoders, we ensure that clients are not restricted in using a particular video decoder, nor will we be obliged to produce and maintain a specialised video decoder. This allows customers to use expensive, specialised video decoder hardware, or preferred software decoders, in combination with a secure decryption module to stream and playback an encrypted video stream.

Aside from the technical issues of playing back an encrypted video stream, it is also necessary to ensure that client user requirements are met. Since clients will be paying for a streaming video service, they will expect certain digital video playback features to be available, and these include pause, seek, slow motion playback and high-speed playback in both directions. These requirements have already been discussed as an issue required by the streaming video server. In the case of the client it imposes a requirement that the decryption/decoder application must handle these features in an encrypted scheme. The encryption algorithm must not be reliant on playback speed to function correctly. It also requires resynchronisation and re-keying capability to support seek and pause functionality. Finally, and most importantly, the entire security procedure must be completely invisible to the client. Once the client has completed the monetary transaction to obtain authorisation to view a video, there should be nothing further required of the customer to view that video. If the decryption process is intrusive, there will be no client support for the application.(Anderson, 1996; Shanableh and Ghanbari, 2001; Wu et al., 2001)

Client platform and user expectations impose four restrictions on the design of the video encryption algorithm. Client platform restrictions enforce that the decryption process requires minimal CPU cycles and that it be compatible with all existing decoders. User imposed restrictions enforce that the decryption process requires support of all expected digital video playback functionality and that it not be intrusive to the user.

2.6 Conclusion

While entertainment quality video streaming is not possible over the Internet today, it will be in the near future and any true large-scale video streaming service on the Internet must follow the Third Party Distributed Server model. This is the only model that is economically feasible to build, prevents duplication of expensive hardware, and allows small companies to provide a service to a large range of customers. The technical issues in providing such a service are not the only hurdles to overcome when considering its implementation, we must also consider the copyright owners and protection of video assets against theft. In fact, unless this protection is ensured, no copyright owner will make content available for an online streaming service, leading to the ultimate failure of the service.

Copyright protection can only be guaranteed using some form of encryption. The requirements identified in this chapter for any proposed encryption scheme are:

- Video must be encrypted prior to being placed online.
- Video must be distributed to streaming servers in encrypted form.
- Video must be installed onto streaming servers in encrypted form.
- Video must be streamed to the client in encrypted form.
- All streaming server functionality, such as implementation of indexed and high-speed playback, must be maintained when streaming encrypted video. The format of the encrypted file must ensure that this functionality is provided on a wide range of existing and future streaming server products.
- All client playback functionality, such as indexed and high-speed playback, must be maintained when receiving and decoding the streamed encrypted video. The format of the encrypted stream must ensure that correct decryption can occur regardless of the different streaming modes.
- All content must be protected. No segment of video or audio should be retrievable given the encrypted video asset.

Chapter 3

Existing MPEG-1 Encryption techniques

In Chapter 2, I defined the requirements for a video cipher algorithm that would function within the distributed streaming server model. One of the primary requirements was in ensuring that the encrypted video file would install and stream from existing video streaming servers. Streaming servers have some knowledge of the video file format that allows them to provide digital video services to the end user, therefore any encrypted video must appear to be a valid MPEG-1 stream for installation purposes. Also required when considering existing video ciphers is the level of security they provide and their speed of execution. To date, there are no existing cipher schemes that meet all of the requirements listed in Chapter 2. These requirements are quite specific and require that the video cipher algorithm will:

- Place no extra demands on streaming server developers.
- Place minimal demands on customer equipment.
- Satisfy all of the concerns of the copyright owners.

In this Chapter I provide a survey and analysis of existing MPEG-1 encryption methods against the identified requirements. These existing algorithms can be categorised into one of three groups depending on their mechanism of operation:

- **Network and Transport Layer** – Includes IPSec and SSL, where the protection of content is performed at a layer underneath the streaming video application.
- **Full encryption** – The entire contents of the bitstream to be delivered is protected.
- **Partial Encryption** – Some of the stream is encrypted for protection while other portions of the bitstream are left as plaintext.

3.1 Network and Transport Layer Encryption

One of the most obvious approaches is to use existing protocols available at lower network layers – below the application layer – such as IPSec or SSL to encrypt the video stream. This places minimal requirements on system designers as the Copyright protection would be handled external to the existing infrastructure. Unfortunately, there are many drawbacks associated with these two approaches as discussed below.

3.1.1 IPSec Encryption

IPSec is a Network Layer protocol (IETF, 1998a; IETF, 1998b; IETF, 1998c) that provides for secure communications between any two IP enabled workstations. The standard is usually used to provide Virtual Private Networks (VPNs) for secure communications between two sites over a public network infrastructure. In a video streaming solution, IPSec would be installed either on the video-streaming server or the gateway router at the site of the company running the server. This would ensure that all traffic was encrypted as it left the site and not decrypted until it reached the IP stack at the client computer. IPSec is an extension of the IP Protocol and IPSec datagrams can be routed by any IP Routers on the network. On the surface, IPSec appears to provide a simple solution to the concept of streaming protected video as it is implemented in the Operating System as part of the IP Stack. Since this provides a single code base which can be used by a number of application, errors and bugs are minimised. This approach also guarantees compatibility with all existing streaming video products.(Bozoki, 1999; IETF, 1998c)

While IPSec would provide protection of the steaming video against interception of the stream by a network eavesdropper, it is unsuitable for use in a streaming video application:

- **Speed** – IPSec is a slow protocol on older host platforms as it primarily uses Block Based ciphers to encrypt IP data.(NIST, 1993a)
- **Scalability** – The server platform – or gateway – would be responsible for applying the cipher to all streams emanating from the server(IETF, 1998a; IETF, 1998c). This would severely limit the maximum number of concurrent streams that can be supported by an individual server or site.(Qiao and Nahrstedt, 1996)
- **Security** – The Streaming Server application has access to the original video as plaintext data which is sent to the IPSec layer for encryption prior to delivery. As such, anyone with administrator privileges on the server platform will have access to the plaintext bitstreams. In a third-party distributed server environment, there will not always be the required level of trust in the streaming server operators who will have access to the digital video content. Also, there is a level of trust required that the streaming server operator have adequately secured their system against outside attack and subsequent theft of content stored on the server.(IETF, 1998a; IETF, 1998b; IETF, 1998c)
- **Copy Protection** – The Client Playback application will also have access to the original video as the encrypted IP datagrams are decrypted prior to being passed to the application for decoding and playback. The owner of a playback PC could easily commit theft of a digital video stream by legally streaming the video to the playback computer, capture all the decrypted datagrams after they leave the IP Stack, then reassemble the IP data into the plaintext video bitstream.(IETF, 1998a; IETF, 1998b; IETF, 1998c)

These issues are normally not a problem with IPSec since the protocol is designed for secure communications between two trusted parties, however a streaming video service would require distribution from one trusted party (the content owner central server) to an untrusted party (the client), potentially via a second untrusted party (the streaming server operator).

3.1.2 SSL Encryption

SSL (Secure Sockets Layer) is a Transport Layer protocol (Group, 1996) that provides for secure communications between any two applications running on IP enabled workstations. It is an extension of the TCP Protocol and as such, SSL datagrams can be routed by any IP Routers on the network and can be understood by any network equipment that processes TCP packets on the network. Like IPSec, SSL appears to provide a simple solution to the concept of streaming protected video, indeed it is often used to provide secure web services such as access to purchased digital data and to personal information such as banking details. However while it may be suitable for these applications, it is not suitable for the concept of streaming video:

When used to provide secure access to purchased digital content, the content is often purchased for the permanent use by the consumer who can then use that data as they please. The concept of streaming video is that the customer purchases the right to view the content but not to store and re-use the information. Also, when used to provide secure access to personal information – like banking – the data being accessed already belongs to the customer and the protocol attempts to ensure that other users cannot also access that information.

While IPSec is usually implemented within the communications Protocol Stack, SSL is usually implemented as part of the application performing secure communications, therefore this means that existing streaming server platforms and client playback applications would have to be modified to support SSL, breaking the requirement that the chosen cipher system would not require any modifications to streaming server platforms. Related to this issue is that SSL provides secure TCP-like communications, many streaming servers use UDP – or related protocols such RTP – as a Transport Layer protocol which is not protected by SSL at all. This means that existing streaming server platforms would not only need to be modified to support SSL, but also to support a TCP based video stream rather than UDP based streaming.(Group, 1996; Bozoki, 1999)

Finally, with SSL, the same major problems with IPSec are repeated – the video bitstream is stored in plaintext at the streaming server and again at the playback application prior to decoding and display, the streaming server must be secured against outside attacks, and the streaming server has high processor requirements due to the encryption of multiple concurrent streams. This leaves an SSL based system open to easy theft by an operative working inside the system, as well as increasing the complexity due to minimising the number of concurrent streams each streaming server can support.(Qiao and Nahrstedt, 1996; Schneier, 1996a; RSA, 1996)

3.2 Full Encryption

A second approach to Copyright protection of streaming video is the encryption of the entire video stream(Qiao and Nahrstedt, 1996). The technique can be applied in one of two ways. The first approach is similar in scope to IPSec and SSL based systems, with much the same problems and vulnerabilities as an SSL based system. In this case, the video would be stored as plaintext on each streaming server platform, the streaming server would then encrypt the entire bitstream before delivery to the client player for decryption. The drawback to this approach is again that the video must be stored as plaintext on each streaming server, requiring trust in each streaming server operator not to engage in theft and to secure their own systems against attack. This approach also has the drawback that the streaming server must encrypt each active stream on the fly, severely limiting its ability to handle multiple streams due to the processing load required to encrypt multiple streams. Finally, this approach requires many modifications to be made to existing streaming server platforms to ensure that the video is streamed in encrypted form.(Qiao and Nahrstedt, 1996; Schneier, 1996a; Stinson, 1995; Menezes et al., 1997)

The second approach would be to encrypt the bitstream at the central server and then distribute the encrypted bitstream to the third-party streaming servers for delivery to the customer. This approach reduces any security flaws due to the administration of the streaming servers, as well as reducing the processing requirements by these streaming servers – the bitstream is already encrypted and no further load is required to encrypt the streams prior to transmission. Unfortunately, this approach means that many existing streaming server products cannot be utilised in the system as the encrypted bitstream will not be in a format that can be understood and used by these platforms. The streaming server platforms would have to be specifically developed to stream completely encrypted bitstreams. Even so, these platforms would not be able to provide advanced digital playback functionality such as indexed or high-speed playback unless further modifications were made to store metadata about specific index points within the encrypted bitstream. This is so because some decoding of the installed bitstream must be performed by the streaming server in order to provide this functionality.(Anderson, 1996; Chen et al., 1995; Frimout et al., 1995; Gemmell et al., 1995; Jayanta et al., 1994; Leditschke and Johnson, 1995; Lin et al., 2001; Shanableh and Ghanbari, 2001)

Complete encryption of the bitstream is not a viable solution when streaming video. A system where the cipher is applied at a central server rather than at each streaming server obviously minimises system complexity and streaming server processor requirements, as well as improve the overall security of the system as trust in operators of the streaming server platforms is no longer required. This idea also fails when considering the provision of advanced digital playback modes.

3.3 Partial Encryption

Partial Encryption of an MPEG-1 bitstream involves the protection of segments of the original bitstream while other portions are left as plaintext. Initially, partial MPEG encryption was

considered due to the processing requirements of encrypting the entire bitstream. CPU processing power is more abundant today and this issue is less of a concern. The modern approach to partial encryption of the MPEG-1 bitstream considers other aspects – indexing into an encrypted file and/or real-time decryption and playback. Previously, no consideration has been placed on the suitability of partial encryption to video streaming. In this section I analyse a series of MPEG-1 partial encryption ciphers and examine their applicability to streaming from a server platform that has no concept of the underlying cipher algorithm. I also explore the suitability of the cipher to support real-time decryption and playback at the client in both indexed and high-speed playback modes.

3.3.1 SECMPEG

The SECMPEG MPEG-1 cipher was designed by Meyer and Gadegast (Meyer and Gadegast, 1995). This cipher applies one of four partial selection algorithms to the original MPEG-1 bitstream and protects the selected data using either the DES or RSA ciphers. Once encryption has taken place, the headers are modified to include extra information so that the protected stream can be properly decrypted at a later stage. The four different algorithms applied by SECMPEG are:

- **Level 1** – Encrypt all of the headers in the MPEG-1 Video Stream.
- **Level 2** – Encrypt all of the headers plus the DC co-efficients of the I-Blocks.
- **Level 3** – Encrypt all I-Frames and I-Blocks in P and B Frames.
- **Level 4** – Encrypt the entire bitstream.

SECMPEG is not suitable for streaming video. Level 1 & 2 encryption leaves portions of the bitstream actually representing Video Content intact while protecting the metadata contained in the headers, this metadata consists of a small range of possible values and can be guessed relatively easily, leaving the encrypted bitstream unprotected. Level 3 encryption protects the contents and is suitable for public storage of video assets but cannot be streamed unless the Streaming Server is aware of SECMPEG protected streams and the decoder is able to decrypt this stream at the remote end. Level 4 encryption is similar in scope to Full Encryption described in the previous section and is therefore also not suitable for streaming purposes. Finally, for all partial selection algorithms, the changes made to the MPEG-1 headers make them non-compliant to the MPEG-1 bitstream format and as such they cannot be streamed by any existing Streaming Server products.(Meyer and Gadegast, 1995)

By changing the header format, the Video Stream cannot be streamed by existing server products. Also, the playback application would either require a decoder capable of parsing the modified headers or a decryption module to reconstruct the original headers prior to playback. A further problem with SECMPEG is the inability to index into the bitstream due to both changes in the header and non-resynchronisation of the cipher employed. This means that SECMPEG encrypted video can only be played back from beginning to end at normal playback speed, precluding implementation of indexed or high-speed playback modes.(Shanableh and Ghanbari, 2001; Leditschke and Johnson, 1995; Lin et al., 2001)

3.3.2 Zig-Zag Permutation Algorithm

When encoding Macroblocks within the MPEG-1 Video Stream, each block of 8x8 pixels is processed using a Discrete Cosine Transform and encoded in a zig-zag pattern. The processing order of the zig-zag pattern is fixed and the same for each Macroblock. Tang (Tang, 1996) has proposed an MPEG-1 Video Cipher which functions by using a random permutation list to map the 8x8 block rather than the fixed zig-zag pattern. As well as re-arranging the order of DCT co-efficients, the algorithm proposes splitting the DC co-efficient to hide its relatively large value amongst the smaller AC co-efficients. The same generated random permutation list is applied to all Macroblocks being encoded or decoded – the cipher key is the random permutation list.

Further modifications are suggested to the basic algorithm, one involves generating two random permutation lists and selecting which one to apply using a pseudo-random coin flipping algorithm, while the second groups blocks of 8 DC co-efficients and applies the DES encryption algorithm. Both approaches increase the key length, the first requires a key that includes both permutation lists and the seed to the coin flipping algorithm, the second requires an extra 56-bit value to use as the DES key.(Tang, 1996)

The Zig-Zag Permutation algorithm is particularly vulnerable to attack. Given some known plaintext, the co-efficients of the encrypted bitstream can be compared against those of the plaintext. A single known frame of video will contain a large number of Macroblocks, enough to determine the random permutation pattern. Once this pattern is obtained, it can be re-applied to the remainder of the sequence to retrieve all frames. No extra security is offered by using two permutation patterns – the same procedure can be used to retrieve both patterns. Once both patterns are available, we can apply both patterns to each Macroblock and select the most likely (high DC co-efficient and low-order AC co-efficients gathered in the upper-left corner of the block) of the two generated Macroblocks. The Zig-Zag Permutation Cipher is also vulnerable to a Ciphertext only attack as described by Qiao.(Qiao and Nahrstedt, 1996; Qiao et al., 1997)

While the security afforded by this cipher is questionable, it is interesting to consider how suitable it is when streaming video. Because the cipher only modifies the contents of the Macroblocks, the Video Stream headers are primarily left intact. While the contents of these headers may need to be modified, they will still contain the correct information indicating frame numbers and timestamps within the bitstream. The encrypted bitstream will appear to be a valid MPEG-1 sequence and as such can be installed onto existing Streaming Server platforms. These platforms will then be able to extract individual frames from the encrypted bitstream for delivery over the network in either indexed or high-speed playback modes.

At the client playback application, because each frame is encrypted using the same random permutation list, resynchronisation of the cipher in each playback mode is not required, ensuring that the received bitstream could be decrypted and played back correctly regardless of the selected playback mode.

The nature of the Zig-Zag Permutation Cipher means that the cipher is most efficiently applied during the encoding and decoding stages of video playback (Tang, 1996). During encryption, it is possible, but more complex, to apply the cipher to an already encoded bitstream, this would involve the following steps:

- Decode each Macroblock
- Retrieve each co-efficient through de-compressing the Huffman encoded stream.
- Apply the random permutation pattern to the retrieved co-efficients.
- Compress the randomly sorted list using the Huffman compression algorithm.
- Reconstruct the MPEG-1 Video Stream entirely – this last step is necessary since each Macroblock will now be a different size. The different order of co-efficients will mean that the Huffman algorithm will not be as efficient and lead to larger Macroblock sizes. This may have an effect on the contents of the headers of the remainder of the stream. The increased Video Stream length will require the stream to be re-multiplexed into the MPEG-1 System Stream.

During playback, it is imperative that the decryption process be built into the decoder being used. A simple decryption process is the same as the encryption of a pre-existing MPEG-1 bitstream, the computational effort involved in deconstructing the bitstream, retrieving the co-efficients, re-ordering the co-efficients, and finally reconstructing the bitstream, is expected to be near that required to decode a plaintext MPEG-1 Video Stream. This should be true since the procedure involved is practically the entire decoding process. Tang (Tang, 1996) proposes a decoder with a built-in decryption module. This allows the random permutation list to be applied after the co-efficients have been extracted from the bitstream, but before they are decoded back into individual pixel values. In this instance, the cipher is extremely efficient with CPU resources as the process of re-organising the DCT co-efficients is short and processor friendly. This implies that efficient decryption and playback is only possible with a specially written decoder, this precludes allowing users to choose their own decoder platform or using a hardware based decoder to playback an encrypted bitstream.

The Zig-Zag Permutation Cipher is not suitable for streaming video. While the encrypted file can be successfully installed and streamed from existing Streaming Server platforms in a variety of different playback modes, there other problems.

- Correct client playback can only occur with a specially written decoder that combines an MPEG-1 Video Decoder with the cipher module, this requires continuous maintenance of this software to incorporate improvements as well as not allowing the use of third-party software or hardware based decoders.
- The algorithm itself leaves the encrypted bitstream vulnerable to attack. The entire bitstream can be reconstructed using a known-plaintext attack while portions of the video sequence can be obtained using a ciphertext-only attack.
- The encoded audio stream is not protected.

3.3.3 Video Encryption Algorithm

Qiao and Nahrstedt (Qiao and Nahrstedt, 1997) propose a Video Encryption Algorithm (VEA). This cipher functions on chunks of data within an individual frame to be protected – All data at the Picture Layer within the MPEG-1 Video Stream is selected for encryption. The data encoded within the Picture Layer is then encrypted using the following algorithm:

- Sub-divide the bitstream to be encrypted into blocks of an even number of bytes – the authors suggest 128.
- Process this 128 byte block into two 64 byte blocks ($a_1a_2a_3...a_{64}$ and $b_1b_2b_3...b_{64}$) using a key to select which bytes to select into List 1 and which to select into List 2.
- XOR the two 64 byte blocks to form a third 64 byte block ($c_1c_2c_3...c_{64}$).
- The original 128 byte block is replaced with the third 64 byte block ($c_1c_2c_3...c_{64}$) and the 2nd 64 byte block ($b_1b_2b_3...b_{64}$) encrypted using a cipher such as DES.

This cipher is further secured by varying the key used to select bytes into List 1 and 2 such that the same pattern is not used repeatedly. Continuously varying this selection property involves the use of a range of different keys. Rather than incorporate all of this information into a single key, the authors have chosen to encode some of the keys within the encrypted bitstream.

In order to be able to decrypt the bitstream correctly, the authors have chosen to modify the contents of the original bitstream from the Picture Layer down. The proposal allows removal of ail Picture and Slice Start Codes within the Picture to be encrypted followed by the insertion of a new header block at the start of the Picture that contains information about the number and length of Slices encoded within the Picture. Also included are details on the key to use to decrypt the given frame. As explained in (Qiao and Nahrstedt, 1997), this approach does not lengthen the bitstream but rather shortens it.

Unfortunately, the authors do not discuss how to encrypt the final block in the Picture Layer which could potentially be shorter than 128 bytes. It is not suggested whether this final block should be extended to the required block size nor what values to insert to perform this function.

The security provided by VEA is excellent. Statistical analysis of an MPEG-1 bitstream coupled with the chosen block length of 128 bytes can be shown to prove that List 2 should be a unique bitstream which can be treated as a one-time pad to encrypt List 1. The encrypted bitstream consists of the ciphertext and an encrypted copy of the one-time pad. Given that further randomness is introduced by regularly changing the List selection scheme, the cipher should be as secure as the cipher used to protect List 2.(Qiao and Nahrstedt, 1997)

The suitability of the VEA for video streaming is a different proposition. While the contents of the upper layers of the encrypted bitstream still contain valid MPEG-1 information, all content from the Picture Layer down is changed to some degree. The contents of the Picture Header

remain intact but have been moved to a different section of the original stream and are no longer preceeded by the Picture Start Code. This ensures that the Picture Header cannot be located by the Streaming Server and that the high-speed playback modes cannot be implemented – as these modes require the extraction of individual I-Frames within the bitstream. For the same reasons, it is possible that some Streaming Servers will refuse to install the encrypted bitstream.

Furthermore, real-time decryption is also problematic, primarily because the bitstream will be longer in length following decryption. This implies regeneration of the plaintext Video Stream prior to passing it to a decoder for final processing. Extra processor and memory requirements complicate implementation of client playback software.

The VEA Cipher is not suitable for streaming video. While the encrypted file can be played back in an indexed playback mode, the cipher design leads to other problems that are not addressed:.

- The cipher modifies the entire Picture Layer, destroying any metadata stored at this point. While no indexing information is contained within this layer, the Picture Header does indicate the format of the frame and whether it is an I-Frame or not. This information is used by many streaming servers to implement high-speed playback modes. As such, unless the server is designed specifically to stream VEA encrypted content, these playback modes cannot be supported.
- The aforementioned changes mean that the encrypted bitstream is no longer a valid MPEG-1 Video Stream. Some streaming server products may refuse to install the encrypted video for this purpose.
- The encoded audio stream is not protected.

3.3.4 Video Encryption Algorithm – Number 2

Shi and Bhargava (Shi and Bhargava, 1998c) propose a different algorithm, also called Video Encryption Algorithm (VEA). In its initial incarnation (Shi and Bhargava, 1998c), this algorithm requires that the sign bits of all AC and DC co-efficients within the Video Stream be encrypted. The approach suggested is to use a binary key where each bit of the key is XORed with the sign bit selected for encryption. Once the key length is exhausted, encryption continues again by re-using the key. The authors also suggest regular resynchronisation at the beginning of each Group Of Pictures (GOP) by re-starting encryption from the beginning of the key.

The authors later modified their algorithm (Shi and Bhargava, 1998a; Shi and Bhargava, 1998b) to also include the encryption of the sign bits of the motion vectors. In both algorithms Shi and Bhargava directly use the key for XOR purposes, although it is possible to use the key to seed a random bit generator for increased security. Unfortunately, regardless of the cryptographic value of the random bit generator used, the system does not properly secure the video stream and is susceptible to a known plaintext attack. A plaintext attack is achieved using the following steps:

- Use the VEA approach to determine which bits are sign bits and are therefore selected for encryption.

- Compare a known sequence of frames with their encrypted counterparts to determine which sign bits have been changed and which have been left unaltered – this will result in the pseudo-random bit sequence used to encrypt those frames.

- Since the same bit sequence is reused for each GOP, it can be used to decrypt the entire MPEG-1 bitstream – note that this is true even if the bitstream is truly random, a One-Time Pad, since the random stream is used repeatedly rather than just once.

As for the Zig-Zag Permutation Cipher (Tang, 1996), it is imperative that the decryption of a VEA encrypted bitstream is performed within the MPEG-1 decoder. This process involves:

- Decoding the MPEG-1 ciphertext bitstream down to the MacroBlock layer.

- Using the Huffman codes to retrieve the AC and DC co-efficient values.

- Modifying the sign bits of the co-efficients.

- Continue decoding the bitstream.

While decoding the bitstream to locate the MacroBlock contents is a simple procedure, the decoding process begins properly upon reconstructing the MacroBlocks. This cipher is applied as part of that procedure, decryption prior to decoding will be time consuming as it requires the co-efficients to be decoded, corrected and finally re-encoded again. If the cipher is incorporated within the decoder, CPU utilisation is extremely efficient as decryption involves only a simple XOR for each co-efficient. The efficiency of the algorithm is decreased when considering separate decryption prior to decoding as some stages of the decoder must be performed multiple times. Like for the Zig-Zag Permutation Cipher, this precludes allowing users to choose their own decoder platform or using a hardware based decoder to playback an encrypted bitstream.

A VEA encrypted bitstream can be successfully installed onto a Streaming Server. The only modifications to the plaintext bitstream occur within portions of the MacroBlock. The information that the Streaming Server requires in order to implement different playback functionality such as indexed and high-speed playback modes is left as plaintext – the server will successfully stream a VEA encrypted bitstream.

The alternative VEA Cipher is not suitable for streaming video. While the encrypted file can be successfully installed and streamed from existing Streaming Server platforms in a variety of different playback modes, there other problems.

- Correct client playback requires a specially written decoder that combines an MPEG-1 Video Decoder with the cipher module. This requires continuous maintenance of this software to incorporate improvements as well as not allowing the use of third-party software or hardware based decoders.

- The cipher is not secure and is vulnerable to a known plaintext attack where the entire bitstream can be reconstructed. Given that most commercial video sequences would use some known plaintext – such as a company logo – at the beginning of most bitstreams, this attack would be easy to perform..

- The encoded audio stream is not protected.

3.3.5 Frequency Domain Scrambling Algorithm

Like many of the previously presented ciphers, the Frequency Domain Scrambling Cipher proposed by Zeng and Lei (Zeng et al., 2002; Zeng and Lei, 1999) operates on information encoded within the Macroblock layer, in particular the co-efficient values stored within the Macroblocks. At its basic level, this cipher is similar to the VEA cipher proposed by Shi and Bhargava, where sign bits of co-efficients are encrypted. The cipher further strengthens the approach by also considering the following measures:

- **Encrypting *refinement* bits within a co-efficient** – An AC or DC co-efficient can be divided into two parts. The *significance* part signifies the approximate magnitude of the co-efficient and consists of the most significant 1 bit and any preceeding 0 bits. The *refinement* part consists of the remaining bits. The choice is made as the *significance* part contains the main information and compresses well while the *refinement* part has a relatively even distribution and can be encrypted without impacting greatly on the compression rate.

- **Block Shuffling** – The bitstream is divided into a series of blocks which are shuffled using a changing shuffling table (determined by a key). Since the actual contents of the stream are unchanged, compression remains high. Instead, the positions of Macroblocks within the stream have been moved.

- **Block Rotation** – A Macroblock is rotated pseudo-randomly to further protect the original image. Again the pixel values are unchanged and therefore compression ratio is not affected.

The cipher as discussed by the authors is very secure and would be adequate for the protection of Copyright. Similarly, as all modifications to the plaintext bitstream are performed on data encoded within a Macroblock or Slice, header information used by Streaming Server products to provide advanced playback features such as indexed or high-speed playback remains unaffected. This means that the cipher is compatible with existing streaming server products.

The Frequency Domain Scrambling Cipher is more reliant on being implemented as part of the decoder than any previously presented ciphers. The complexity of operations mean that CPU efficiency is only realised when decryption is performed as part of the decoding cycle. As for other ciphers with this problem, this precludes the use of third-party and hardware based decoders in system implementation.

The Frequency Domain Scrambling Cipher is not suitable for streaming video. While the encrypted file can be successfully installed and streamed from existing Streaming Server platforms in a variety of different playback modes, other problems are:

- System requires a specially written decoder combining an MPEG-1 Video Decoder with the cipher module.
- The encoded audio stream is not protected.

3.3.6 A Unique Cipher

Griwodz et all propose a unique algorithm (Griwodz et al., 1998) and approach to protection of distributed video. First a Poisson process is used to select bytes from the original plaintext stream at pseudo-random intervals. These bytes are then extracted from the bitstream to form a new bitstream. The corresponding bytes from the original bitstream are then corrupted, using the values of nearby bytes to calculate a value that is statistically similar to the original value. Finally the corrupted plaintext is freely distributed. Playback is affected through purchase of the new bitstream containing the un-corrupted bytes, which can then be inserted back into the corrupted bitstream prior to playback. The new bitstream is delivered in encrypted form.

This novel approach is unique, experimentation by the authors show that only 1% of the original bitstream need be corrupted to render the file unplayble. Since this approach is applied to the MPEG-1 System Stream, the effect also ensures that audio is protected. The authors envisage a use where the corrupted bitstream is made freely available on local servers and caches while the smaller, encrypted bitstream is delivered from a central server. While the system functions well for a download now and play later system, it will not function in a streaming video implementation.

There is no telling which bytes will be corrupted by the system and there is therefore the potential that the corrupted bitstream will not install or be successfully streamed from existing streaming server products. There is also the issue of providing indexed and high-speed playback. Bitstream position information, while readily available when decoding from a file, is usually not available when being streamed, and these playback modes ensure that this value could change constantly. Being unable to keep track of the current byte position in the original stream will complicate the implementation of the decryption module in locating the corrupted bytes.

This cipher, while tackling the issue of video encryption from a completely new angle, will not function with existing streaming server products and is therefore not suitable for use in streaming video.

3.3.7 Multi-Layer Encryption

Tosun and Feng (Tosun and Feng, 2000; Tosun and Feng, 2001) propose a modification on the VEA cipher developed by Qiao and Narhstedt. The new proposal looks at the 64 co-efficients produced by the DCT transform and breaks them into three separate layers. The first layer consists of the lowest frequency (most significant) co-efficients and is called the Base Layer. The Middle Layer consists of the mid-range frequency components, while the Enhancement Layer is formed by the remaining highest frequency co-efficients.

The proposed cipher assumes separate transmission of each of the three layers using different transport characteristics, ideally guaranteed delivery of the Base Layer high probability of delivery of the Middle Layer while the Enhancement Layer gets the lowest priority. The three individual streams are recombined at the client prior to decoding and display.

The approach proposed by Tosun and Feng is to apply the VEA Cipher developed by Qiao and Narhstedt to the Base and Middle Layer only, the Enhancement Layer – containing minimal information on the actual content – is delivered as plaintext. The idea is to enable secure delivery of content over a network that potentially cannot cope with the required throughput. In this way, even if only the Base Layer is transmitted, it can be decrypted and displayed independently of the remaining layers, resulting in poorer quality video rather than a discontinuity in playback. Layered approaches to Streaming Video have been developed to counter networks without Quality of Service provisions, this paper attempts to merge video cipher techniques with Layered delivery.

This Multi-Layered Cipher is also not suitable for streaming video. It necessary suffers from the same issues as Qiao and Narhstedt's original algorithm. Also, not all existing streaming server products offer Layered Streaming and those that do will not necessarily use the same approach to do so. The proposed cipher lacks compatibility with the wide range of products necessary to enable multi-platform streaming server implementations.

3.3.8 Selective Macroblock Encryption

Alattar, Al-Regib and Al-Semari (Alattar et al., 1999; Alattar and Al-Regib, 1999) propose a set of four ciphers which operate on the Macroblocks encoded within the Video Stream:

- **Method 0** – Encrypt all I-Macroblocks and the Macroblock headers for all predicted Macroblocks.
- **Method 1** – Encrypt every *n*th I-Macroblock.
- **Method 2** – Encrypt every *n*th I-Macroblock and the Macroblock headers for all predicted Macroblocks.
- **Method 3** – Encrypt every *n*th I-Macroblock and every *n*th Macroblock header for all predicted Macroblocks.

In each method, all data is encrypted using DES. The authors do not specify how to encrypt data when the length of the Macroblock is not a multiple of 64 bits (the DES block-size), an assumption is that the Macroblock is padded with 0 bits prior to encryption. The algorithm also recommends resetting the count for every nth block at the start of each slice and periodically changing the DES key. The first option ensures correct selection of Macroblocks within a slice, important if data is lost. A dropped Macroblock results in incorrect decryption until the count is reset, in this case at the start of the next slice. The second recommendation makes attacking the cipher more complex as the DES key is constantly changed throughout the bitstream.

Experimentation by the authors show that the encrypted video content is not viewable. Given that the DES Cipher is provably secure against all but a Brute Force Attack (Schneier, 1996a), coupled with regular changing of the DES key, makes attacking the security of any of the proposed methods computationally infeasible.

The Cipher does not modify any contents of MPEG-1 Video Headers and therefore the resultant stream should install on existing streaming server products and provision of indexed and high-speed playback modes should not be impaired. However, the use of DES to encrypt Macroblocks and Macroblock Headers mean that the resultant MPEG-1 Video Stream will not be of the same length as the plaintext bitstream, since DES can only encrypt data in blocks of 64-bits. The result is that during both the encryption and decryption process, the Video Stream must be de-multiplexed from the System Stream before processing. Finally, the System Stream must be reconstructed prior to decoding. This is not a major issue during encryption as the task is only performed once, but is a potential problem during playback as reconstruction of a System Stream for decoding is potentially time-consuming.

The Selective Macroblock Cipher is not suitable for encryption of streaming video, it suffers from the following problems:

- Decryption during indexed and high-speed playback modes. While the server could certainly stream the encrypted bitstream in these modes. Successful decryption will ensue only if the correct DES key is known for the frame currently being decrypted. The design mentions frequent changing of the DES key, but not how frequent. The changes must be coupled with individual GOP boundaries and to the playback timestamp in order for the decryption module to determine which DES key to use for the current playback mode.

- Reconstruction of the System Stream during Decryption. If the decoder being used at the client workstation requires input of a correctly formatted MPEG-1 System Stream (such as a hardware based decoder), then the System Stream must be completely reconstructed after decryption. This means regenerating the headers to allow for the now shorter plaintext Video Stream. This approach is potentially time consuming.

- Different Length of Plaintext and Ciphertext. If the plaintext and ciphertext bitstreams are of the same length, the decryption process can replace the encrypted data with its plaintext during processing. Since they are not, all unencrypted data from the encrypted Video Stream must be copied to generate a new Video Stream. This bulk data copying can consume valuable CPU resources.

- The encoded audio stream is not protected.

3.3.9 AEGIS Algorithm

Spanos and Maples (Spanos and Maples, 1996) propose an algorithm in which the entire contents of the I-Frame and the Video Sequence Headers are encrypted. This algorithm employs major changes to the format of the bitstream as extra information is inserted to locate start and end points. The resultant bitstream cannot be streamed from existing Streaming Server products due to the non-conformance of the bitstream to the MPEG-1 bitstream format. Similarly, while the authors suggest the encryption of I-Frames only will secure the entire video, others (Qiao and Nahrstedt, 1996) have shown that it is also necessary to consider protection of the content of P and B-Frames. As such, this algorithm is not suitable for streaming video – it is not compatible with existing streaming server products and does not totally protect the encoded content.

3.4 Conclusion

Existing video encryption algorithms fail to meet all the requirements as proposed for a distributed video server solution, exhibiting one or more of the following faults:

- **The encrypted stream could only be decoded efficiently by specialised decoders:** this precludes the use of standard existing decoders – or hardware based decoder modules – at the client end.

- **Non existence of digital playback features:** whilst the encrypted stream could be played back relatively easily, some schemes preclude indexing which means that seek and variable speed playback is not supported.

- **Encrypted stream could only be installed on specialised servers:** this precludes the use of existing streaming server products or requires modifications to these systems.

- **Audio stream not encrypted:** some schemes protect the video segment of the stream but not the audio segment.

- **Video stream not fully protected:** some schemes protect most of the video segment but allow parts of frames to be recovered from the encrypted stream.

- **Encrypted video is larger than the source:** this means that more network bandwidth is required to transmit the encrypted video.

- **Excessive CPU requirements:** some schemes were designed for secure video storage rather than streaming and the CPU requirements to decrypt the video stream are too high.

Given this analysis, in this thesis I propose a new video encryption algorithm that will meet these requirements.

Chapter 4

A Novel MPEG-1 Partial Selection Scheme for the Purposes of Encryption

In this chapter, I present a novel selection scheme for the purpose of partial encryption of an MPEG-1 bitstream, where part of the bitstream will be encrypted while part remains as cleartext. The scheme developed in this chapter demonstrates the viability of partial encryption as a technique that can allow encrypted bitstreams to be installed on, and streamed from, existing video streaming server platforms. This new algorithm is shown to be fast – requiring minimal CPU resources to execute, and is compatible with a range of existing Video Server products of different brands, as well as different types of MPEG decoder systems.

By examining the format and contents of the different layers of the MPEG-1 System Stream, Video Stream and Audio Stream, I will show that no part of the System Stream need be protected via encryption and that the Video and Audio Streams can subsequently be considered separately for the purposes of encryption. The concept of in-place encryption of the Video and Audio Streams within the System Stream is developed. This has the effect of not changing the length of the encoded bitstream, as well as allowing the multiplexed streams to be encoded while not modifying any part of the System Stream. A partial selection scheme is then developed for both the Video and Audio Streams. A practical state machine to select the bytes for encryption and a simple cipher to apply to these bytes is presented, such that all requirements are met.

Finally, this system is tested to prove its viability. These tests show that the encryption process is reversible, allowing retrieval of the original plaintext bitstream, and that the processing load required to support decryption of the bitstream is minimal and does not interfere with the required processing power for video playback. They also show that a series of existing Streaming Video Server products have no problems in serving the encrypted files in a variety of different playback modes. Two separate client playback applications show that the encrypted video can be successfully decrypted and played back in real time in a variety of different modes including indexed and high-speed playback.

4.1 MPEG-1 System Stream Encryption

In this section, I explore the issue of encryption of the MPEG-1 System Stream. When looking at the System Stream, we can ignore the contents of the Video and Audio Streams as they are encoded as the payload data within packets of the System Stream(ISO, 1996a). There are three layers within the System Stream which must be examined to determine whether any of the data contained within each layer must be encrypted in order to protect the content.

4.1.1 Examination of the MPEG-1 System Stream

The topmost layer of the System Stream is the ISO 11172 Layer. At this layer, the System Stream is defined as a series of one or more packs followed by the ISO 11172 End Code (the byte aligned binary sequence 0x00 0x00 0x01 0xb9). Ignoring the contents of the Pack Sequence, the ISO 11172 Layer can be considered to be a long binary sequence of unidentified length, terminating with the four byte sequence (0x00 0x00 0x01 0xb9).(ISO, 1996a; Mitchell et al., 1996)

The Pack Layer describes the format of one Pack within the ISO 11172 Layer. The Pack bitstream consists of a Pack Header, followed by a System Header and a series of one or more Packets – where the System Header is required in the first Pack of the System Stream but is optional for any other Packs (ISO, 1996a; Mitchell et al., 1996). Information contained within both the Pack and System Header includes:

- A Clock Reference indicating a timestamp for decoding the Pack
- A measurement of rate of arrival of data at the decoder
- An upper bound on the data arrival rate
- An upper bound on the number of multiplexed Audio Streams
- A series of flags that are used to set up the Video and Audio decoders
- An upper bound on the number of multiplexed Video Streams
- Optional data to set up buffer sizes within the Video decoders

The final layer of the System Stream is the Packet Layer and it describes the format of one Packet within a Pack. The Packet bitstream consists of a Packet Header, followed by the Packet Payload – a binary series of bytes, the length of which is determined by a field within the Packet Header (ISO, 1996a; Mitchell et al., 1996). Other information conveyed in the Packet Header is:

- An identifier – which Video or Audio Stream the payload belongs to
- The required buffer size for the decoder of this stream
- Both, one of, or neither the Presentation Timestamps of the Packet to the Stream Decoder and to the user

Examing the information encoded within all of these headers, the Pack Header, the System Header and the Packet Header, I note that these fields convey no information on the actual content of the Video and Audio Streams. This information is used to set-up video and audio decoders prior to decoding and displaying data. The information can also be used to determine whether the encoded packet payload belongs to a Video or Audio Stream, as well as determining which of the potential multiple Video and Audio Streams is represented.

Some information encoded within the headers of the System Stream must be left as plaintext, particularly Packet Header information containing the payload size and stream identification. This information is important for a streaming server when streaming a source with multiple Video or Audio Streams as it allows selection of the correct stream, ensuring that other Video and Audio Streams (not viewed) are not transmitted across the network. It also allows the server to pick out and transmit only the Video Stream for high-speed playback.(Anderson, 1996; Lin et al., 2001)

The remaining information, not only in the Packet Header but in all other layers, while important in correctly configuring a decoder for eventual display of the bitstream, does not convey any information regarding the actual content of the encoded bitstream. Encrypting any of this information will not protect any vital information in determining the content of the encoded media stream, it may also cause potential problems:

- **Security** – Since many of these values are either known, or can be easily guessed from a small range of likely possibilities, it gives a cryptanalyst a piece of known plaintext. Having some known plaintext can aide in the procedure of breaking the cipher.
- **Compatibility** – Changing or encrypting some of this information could preclude installing the encrypted video with most existing MPEG capable Streaming Video Servers. Streaming Servers must parse the installed bitstream to a degree in order to provide any features beyond simple playback, such as indexed or high-speed playback. Many servers will refuse to install a file that cannot be properly parsed or will complain when streaming begins.

Encrypting any of the information encoded within either the ISO 11172 End Code, or the Pack, System and Packet Headers will not protect any vital information in determining the contents of the encoded media stream. Furthermore, encrypting any of this information may cause the encrypted bitstream not to be accepted by an existing Streaming Server product for installation. If all this information is left as plaintext, the problem can be described as encrypting the payload data of an MPEG-1 System Stream Packet. Since each packet payload forms a portion of either a Video or Audio Stream, we can consider their encryption separately and look at multiplexing the encrypted Video and Audio Streams within a plaintext System Stream.

4.1.2 Processing the MPEG-1 System Stream

Since there is no relevant data contained within the MPEG-1 System Stream that need be protected via encryption, a simple approach in processing the System Stream for encryption/decryption would be to de-multiplex the stream into its constituent Video and Audio Streams, encrypt/decrypt these streams separately, and then re-multiplex them to form a new MPEG-1 System Stream. While this could be a valid approach at the central server where content need only be encrypted once, there is a potential problem at the client end as recreating the System Stream would require the use of valuable CPU cycles. There is the option of simply de-multiplexing the encrypted Video and Audio Streams and then decrypting these prior to decoding and displaying them. This technique fails for hardware MPEG decoders that require an MPEG System Stream as input.

The solution I propose, in-situ encryption, involves encrypting and decrypting the Video and Audio Streams in-place within the MPEG-1 System Stream. This approach requires that the Video and Audio Stream ciphers do not modify the length of the Video and Audio Streams, and that they can be written to function with re-startable blocks. If this is the case, we can develop an MPEG-1 System Stream parser that passes Packet Payload data to the appropriate cipher module for in-place processing. Since the output data block is the same length, it can be re-inserted into the System Stream at the same place without making any modifications to the Packet headers. In this section I present a state machine that will parse the MPEG-1 System Stream and call separate cipher modules to process the Video and Audio Streams, this in turn will modify the System Stream in-place.

4.1.2.1 Parsing the MPEG-1 System Stream

The main steps required to complete our task are:

- Locate a Packet within an MPEG-1 System Stream
- Determine the Stream ID (which Video or Audio Stream) of the Packet
- Encrypt the Packet Payload

Given the multi-layered structure of the MPEG-1 System Stream (Mitchell et al., 1996), it is possible to bypass all data in an MPEG-1 System Stream until a valid Packet Header is encountered, indicated by the unique byte aligned sequence (0x00 0x00 0x01 *n*), where *n* signifies any byte value in the range 0xbc through 0xff. It is possible to parse the MPEG-1 System Stream by skipping through the bit stream until we encounter the 24-bit byte aligned code (0x00 0x00 0x01), which depicts the potential start of a Packet Header. If the following byte does not indicate a Packet Header, then we jump back to the start of the algorithm looking for a Packet Header. If on the other hand a Packet Header is found, we process the bytes making up the Packet Header and then pass the next *N* bytes, where *N* is the payload length, to the appropriate cipher module for processing. If the Header signifies a private or reserved stream, then the payload is skipped and ignored, otherwise the appropriate cipher is called to encrypt those bytes.

The key to this algorithm is the in-place encryption performed by the Video and Audio cipher modules. If these modules changed the format and/or length of the Packet payloads being processed, this would require subsequent modification of the System Stream Headers to reflect the new condition of the multiplexed Video and Audio Streams. In an extreme case, it may even involve regenerating the System Stream information in order to properly multiplex the encrypted Video and Audio Streams. While the CPU load required to perform this task is not important during encryption (as this process is performed only once prior to installation for streaming), it can become a problem during playback by the client, especially if the decoder being used only accepts input of an MPEG-1 System Stream. In this case, the playback process will involve the regeneration of the System Stream to fit the plaintext Video and Audio Streams, stealing valuable CPU cycles from both the decryption and decoding tasks.

By imposing two restrictions on the Video and Audio Stream cipher modules, we can perform in-place encryption of the Video and Audio Streams within the System Stream, remove the need to re-sequence the System Stream, simplify implementation, and decrease overall CPU load. The restrictions required to achieve these goals are:

- **The cipher modules must be able to process an MPEG-1 Video or Audio Stream in re-startable blocks.** A bitstream can either be processed in its entirety or as a series of shorter bitstreams of any length, and still produce the same output sequence. In this case, we do not need to extract the entire Video or Audio Stream prior to passing it to the cipher modules. The cipher module will process a block of data, remembering its state so that it can continue processing when provided with the next block of data. This means that the payload of each Packet can be encrypted individually as the packets are parsed within the System Stream.

- **The length of the ciphertext is exactly equal to the length of the associated plaintext.** If the output of the cipher module after each re-startable block is the same length as the input, we can encrypt the bitstream in-place within the System Stream. The ciphertext will replace the existing payload within the original System Stream Packet. Since the payload length is unchanged, and the payload represents the same stream type (Video or Audio), no details within the Packet Header or other headers of the System Stream need be updated, allowing the System Stream to remain completely unchanged.

Assuming that the Video and Audio Stream cipher modules meet these restrictions, a State Machine that will parse and encrypt an MPEG-1 System Stream is shown in Figure 4-1. The first sequence of four states are used to locate a valid Packet Start Code, the State Machine enters the fourth state only if the sequence (0x00 0x00 0x01) is encountered, if the next byte is in the range 0xbc-0xff, we have found Packet Header. The fourth byte, or Packet ID is remembered for later reference.

The second section of the State Machine consists of four states that process the Length of the packet, as well as any 0xff stuffing bytes and Buffer Size Information. The most significant byte of the packet length is obtained as input whilst in state 2-1, the least significant byte forms the input whilst in state 2-2. There is a special case whilst in state 2-2, if the Packet ID is equal to 0xbf, then this is the end of the Packet Header and so we move to the final state, namely 4-5. Whilst in state 2-3, we can repeatedly process any number of stuffing bytes by looping back to the same state, as well as process any existing Buffer Size Information by jumping to state 2-4 and then back to 2-3. This State Machine will also process non valid MPEG-1 System Streams – the State Machine will successfully parse a Packet Header with stuffing bytes, followed by more than one Buffer Size Information field and repeat this cycle. A solution to this problem can be found by creating a new state 2-5 into which we change following any input from state 2-4. State transitions from state 2-5 include the three possible transitions from state 2-3 to states 3-1, 4-1 and 4-5. For the situation of parsing an existing MPEG-1 System Stream for encryption purposes, it was deemed not necessary to perform this extra step based on the assumption that the original System Stream is valid. If this is the case then the original State Machine will successfully parse the header. In all states from state 2-3 onwards, the calculated Packet

Length is decremented by one so that when state 4-5 is reached, this variable will correctly represent the number of bytes in the payload.

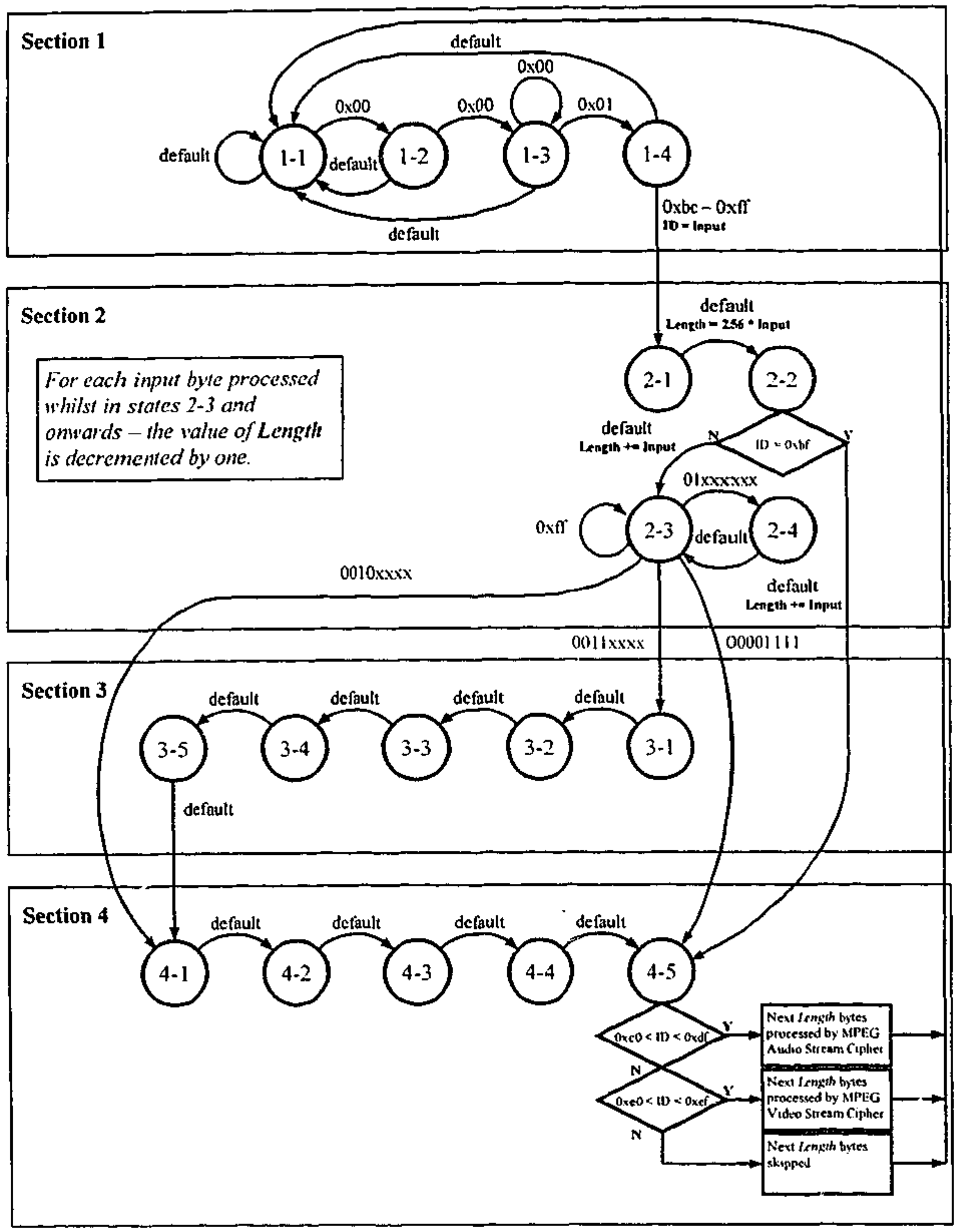


Figure 4-1: State Machine to Encrypt an MPEG-1 System Stream

We enter state 3-1 only when there are two timestamps within the Packet Header, both the PTS and the DTS. The first of these timestamps takes five bytes of input including the transition to state 3-1, this puts us into state 3-5 upon completion. We enter state 4-1 when only one timestamp remains in the Packet Header, either only the PTS or the remaining DTS if there were two timestamps.

This timestamp also takes five bytes of input including the transition to state 4-1, this puts us into state 4-5 upon completion. It is also possible to have no timestamps, in which case we can go to state 4-5 after reading the binary sequence (00001111). Once the final state is reached there is still the Packet Payload to process, if the Packet ID signifies that the payload forms part of an MPEG-1 Audio Stream, then the following *Packet Length* bytes are processed by an MPEG-1 Audio Stream Cipher, if it forms part of an MPEG-1 Video Stream, they are processed by an MPEG-1 Video Stream Cipher, otherwise they are skipped and ignored.

In general, each MPEG-1 System Stream will encapsulate one Video Stream and one Audio Stream, however it is possible for there to be up to 16 Video Streams and 32 Audio Streams multiplexed within the one System Stream (Mitchell et al., 1996). If this is the situation, each multiplexed Video and Audio Stream needs to be encrypted/decrypted separately such that any one of these streams can be extracted from the System Stream, decrypted and then played back. In order to build such a system, we must maintain a cipher module for each possible Video and Audio Stream, requiring 16 Video Cipher modules and 32 Audio Cipher modules. Each of these modules must maintain state information on their own Video or Audio Stream being processed and the System Stream parser must pass the payload data to the correct Cipher Module instance based on the Packet ID within the Packet Header. In practice, this will only be an issue during encryption – during streaming and playback, a streaming server will conserve network resources by only streaming the single Video and Audio stream that the client selects to view/listen to. This means that packets representing other Video and Audio streams are dropped. Since each multiplexed Video and Audio Stream is processed by the ciphers independently of one another, this can be easily coped with.

4.1.3 Summary of MPEG-1 System Stream Encryption

None of the actual information contained within the MPEG-1 System Stream itself needs to be protected by a cipher. The content of this information is irrelevant to the media that needs to be protected and consists of known or easily guessed information that is used to set up the decoder to facilitate playback (Mitchell et al., 1996). However, restrictions imposed by different Streaming Server designs in providing indexed and high speed playback mean that the server must be able to extract multiplexed stream information from the installed MPEG-1 file (Anderson, 1996; Lin et al., 2001). Given this, the format of the MPEG-1 System Stream must remain unchanged. In Section 4.1.2.1 I presented a State Machine that can be used to parse the MPEG-1 System Stream. When this is used with MPEG-1 Video and Audio Stream Cipher Modules that can both process the bitstream in re-startable blocks, as well as process their respective streams in-place within the System Stream, the resultant design will encrypt the Video and Audio Streams contained within the System Stream. The same State Machine can be used in any decoding software to parse and decrypt the System Stream prior to decoding by any existing MPEG-1 Playback Device.

4.2 MPEG-1 Video Stream Encryption

In this section I explore the issue of encrypting an MPEG-1 Video Stream, approaching the issue in the same way as for System Stream encryption – by exploring each layer of the bitstream from the top down, examing which information must be encrypted in order to protect the content.

4.2.1 Examination of the MPEG-1 Video Stream

Like the MPEG-1 System Stream, the MPEG-1 Video Stream is a layered bitstream (ISO, 1996b; Mitchell et al., 1996; Sikora, 1997; LeGall, 1991). The topmost layer is the Sequence Layer and consists of a Sequence Header followed by a series of Groups of Pictures (GOPs), terminated by the Sequence End Code. Further Sequence Headers may be interspersed between individual GOPs. The Sequence End Code is the 24-bit byte aligned sequence (0x00-0x00-0x01-0xb7). The fields of the Sequence Header convey the following information:

- Video Height, Width and Aspect Ratio
- Video Frame and Bit Rates
- Decoder Buffer Sizes
- Quantiser Matrices
- Extension and User Data

Each individual GOP within a Sequence is a bitstream that consists of a GOP Header followed by a series of Pictures, where each Picture represents a single frame in the Video Stream. The fields within the GOP Header are used to store the following information:

- Time Stamp of the First Frame within the GOP
- Whether the GOP refers to frames within other GOPs
- Whether the Frame order within the GOP has been modified due to editing
- Extension and User Data

Each Picture within the MPEG-1 Video Stream is a bitstream made up of a Picture Header followed by a series of Slices, where each Slice represents a horizontal stripe of the individual frame. Information stored within the Picture Header includes:

- Order of the frame within its owning GOP
- Picture Type (I, P or B Frame)
- Decoding Buffer Delay
- Forward/Backward Motion Vector Scaling Information
- Extension and User Data

Each Slice can also be considered as a bitstream made up of a Slice Header followed by a series of Macroblocks, where each Macroblock represents a 16x16 block within the Picture. Information stored within the Slice Header includes:

- Vertical Starting Position of Slice within Frame
- Quantiser Scale Factor

A Macroblock represents a 16x16 block of picture data within an individual frame and it is at this layer of the MPEG-1 Video Stream that actual data representing the Video is stored. Each Macroblock is further divided into a series of blocks, which are made up of the encoded values of the Discrete Cosine Transform (DCT) co-efficients of the pixels represented. These co-efficients are stored in compressed format using the Huffman code.

Examing the information encoded within all of these headers of the MPEG-1 Video Stream, I note that like for the headers of the System Stream, these fields convey no information on the actual content of the Video asset. The Sequence Header contains basic video information such as image size and frame rate, as well as some decoder set-up parameters. GOP and Picture Headers help to partition the stream into individual frames, and tell us how these frames are encoded, and the relationship between frames of a sequence. The Slice Header contains positioning information.

It is important to consider the provision of indexed and high-speed playback modes by a streaming server. In order to generate a high-speed playback bitstream, the server must be able to extract individual frames from the original bitstream, and determine their type (Anderson, 1996; Lin et al., 2001). By not encrypting the contents of the GOP and Picture Headers, we can ensure that the high-speed bitstream can be extracted from the encrypted bitstream. This has implications on how we encrypt the data contained within an individual frame, as the server can start streaming from any frame in indexed playback mode and deliver a series of non-consecutive frames in high-speed playback mode. As such, it will be imperative that the cipher applied to data within the Picture Payload must be able to be resynchronised at the start of each I-Frame – or first frame of a GOP – so that the transmitted bitstream can be successfully decrypted prior to decoding in each playback mode.

To enable a video server to stream the encrypted asset, it is only necessary to preserve the contents of the Headers down to the Picture Header. However, since the Slice Header contains no information relevant to the actual content, and its contents can be reasonably guessed, the partial selection scheme will choose to leave the Slice Headers as plaintext and encrypt only the contents of the MacroBlocks contained within the Slices.

As such, for the purposes of encryption of a Video Stream to protect the encoded content, it is not necessary to encrypt the contents of the Video Stream Headers. Like for the System Stream, it is important to note that many of the values within these Headers are either constant or contain values that can be easily guessed. By encrypting any of this information, the cryptanalyst has some known plaintext which can be used to help break the cipher, leaving this known information unencrypted means that it cannot be used to provide any known plaintext.

It is, however, necessary to encrypt the Slice Payload or MacroBlock Information as this portion of the MPEG-1 bitstream contains the data required to reconstruct the original video images. Some existing MPEG-1 cipher algorithms go deeper and apply encryption to the individual DCT co-efficients within the Macroblock (Qiao and Nahrstedt, 1997; Shi and Bhargava, 1998a; Shi and Bhargava, 1998b), while others encrypt the order of DCT coefficients within the Macroblock (Qiao et al., 1997; Tang, 1996). The problem with both of these approaches is that a specialised MPEG-1 Video Stream decoder is required to decrypt and playback the video. While both these approaches are efficient in the usage of CPU cycles for decryption purposes, they prohibit the use of hardware or third party software MPEG-1 decoders. Since the decryption process is intimately tied in with the decoding process, the CPU cycle cost of decryption alone is as high as the cost of decoding an unencrypted stream. When using a hardware decoder, this means that the benefit of offloading the decoding process from the CPU is lost, when using a third party software decoder, the CPU cycle requirements double. Instead, I suggest that the entire Macroblock be encrypted, thereby removing the intimate connection between the decryption and decoding cycles. A diagrammatic representation of the Video Partial Selection Scheme is shown in Figure 4-2, where the shaded blocks indicate which portions of the Video Stream is encrypted.

4.2.1.1 Restrictions on Encryption of Macroblocks

One important aspect within an MPEG-1 Video Stream is that of MPEG Start Codes. MPEG Start Codes are a byte aligned, four byte binary sequence in which the first three bytes form the pattern (0x00 0x00 0x01). The mechanics of the MPEG-1 Video Stream format means that it is impossible for this byte aligned 24 bit sequence to appear anywhere but within a valid Start Code – in practical terms, this allows us to search an MPEG file at speed by looking for particular Start Codes (ISO, 1996b; Mitchell et al., 1996). Since any presence of these Start Codes indicate a valid MPEG-1 header, a further restriction on the cipher we plan to use is that it must not introduce any false Start Codes, otherwise we would be creating an invalid MPEG-1 Video Stream. In summary, this means that the output of the cipher that is encrypting Macroblocks must not produce the 24 bit byte aligned sequence (0x00 0x00 0x01).

4.2.1.2 Analysis of Selection Criteria

The proposed MPEG-1 Video Stream cipher requires the complete encryption of the existing Macroblocks of the unencrypted Video Stream. The encryption process involves decoding an existing MPEG-1 Video Stream down to the Slice Layer, maintaining the Slice Header and encrypting all Macroblocks. The decryption process would involve decoding to the Slice Layer and decrypting the Macroblocks. This would have no effect on a Video Streaming Server as the Video Server must only be able to extract information down to the Picture Layer. Also, since decoding down to the Slice Layer is simple and requires minimal CPU cycles, the decryption process also requires minimal CPU cycles above the actual encryption algorithm chosen. Finally, since the partial encryption selection and decryption is entirely decoupled from the actual decoding phase of playback, the proposed solution is compatible with a range of existing decoder options.

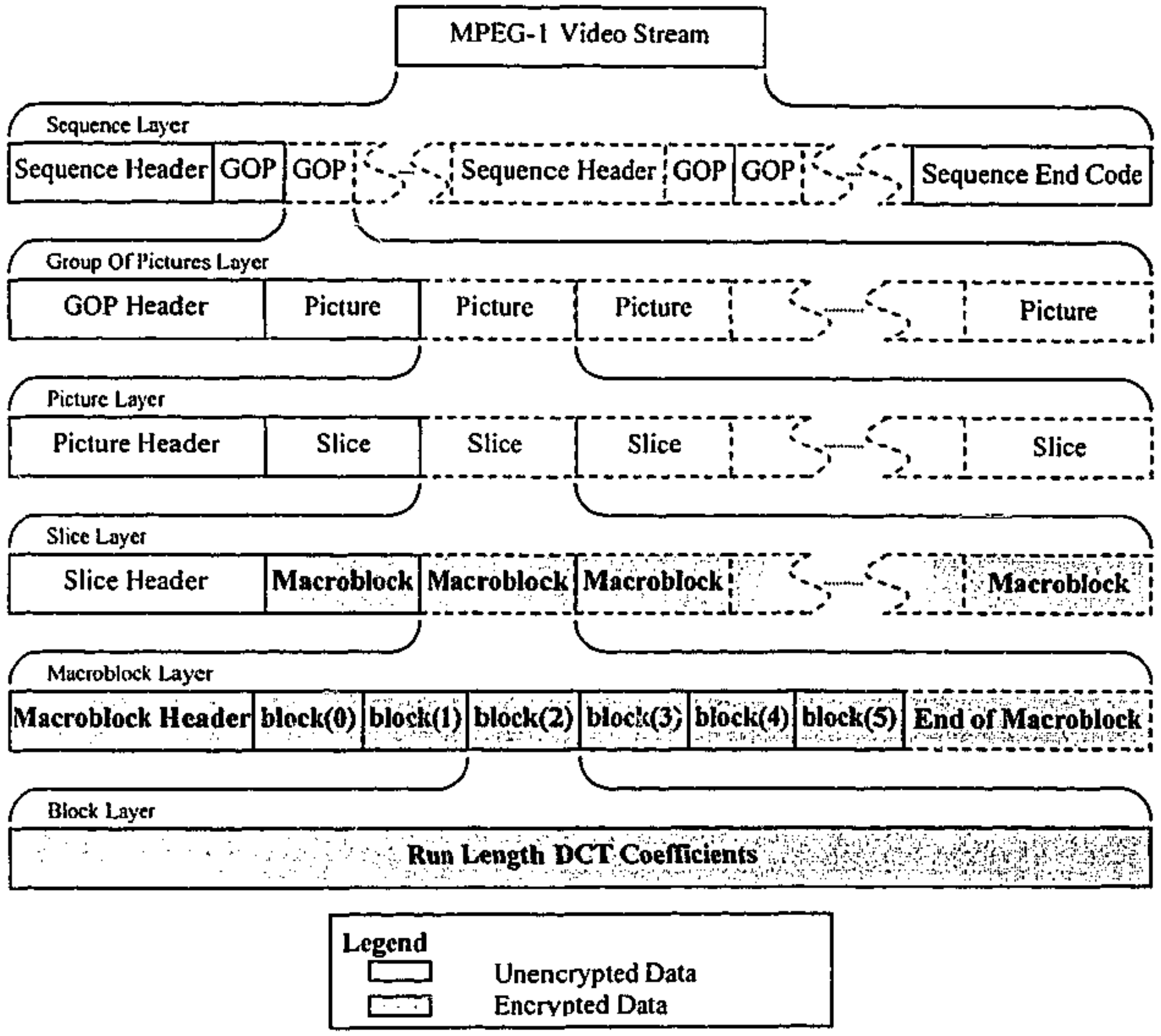


Figure 4-2: Selective Encryption of an MPEG-1 Video Stream

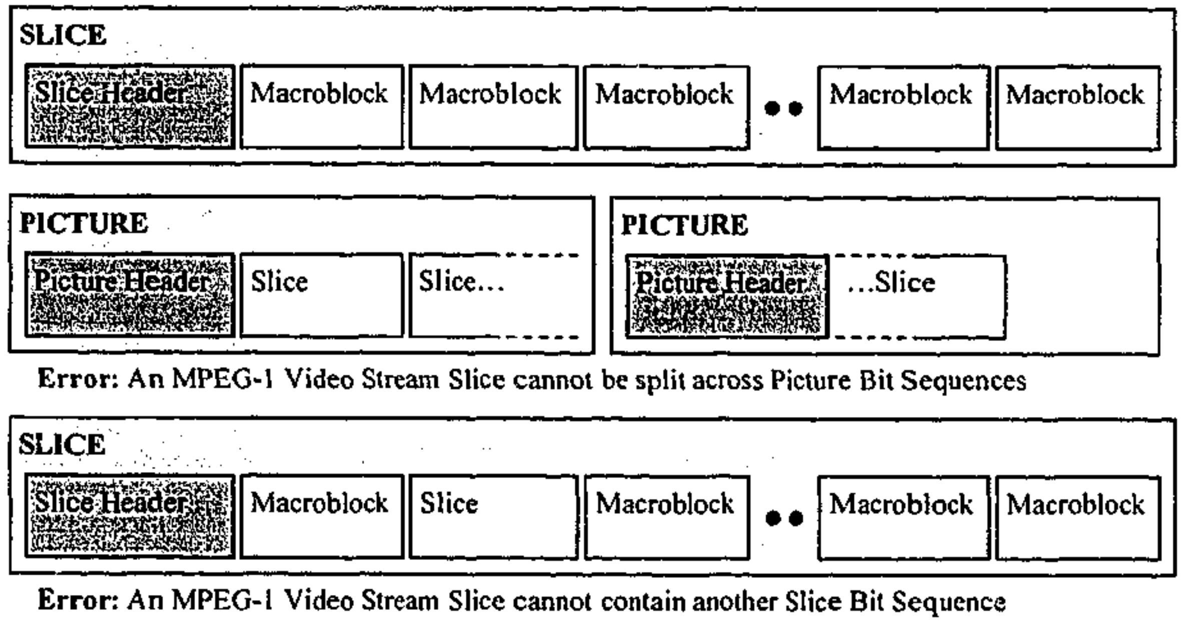
4.2.2 Processing the MPEG-1 Video Stream

In processing the MPEG-1 Video Stream, it is necessary to locate individual MacroBlocks within the bitstream for encryption purpose. This task can be restated as locating and encrypting the payload of Slices within the original bitstream. In this section I present a State Machine that will implement the proposed partial selection algorithm. I also describe a simple Cipher that can be applied to the selected bytes to generate an encrypted MPEG-1 Video Stream that conforms to all of the requirements mentioned in Chapter 2. The same system will also be capable of decrypting the Video Stream back its original state.

4.2.2.1 Designing the Partial Stream Selection State Machine

In order to implement encryption of an MPEG-1 Video Stream, it is necessary to build a state machine that can parse the existing Video Stream such that the selected bytes can be passed to a cipher for processing. Recalling that the partial selection criteria has chosen complete encryption of

Macroblock data whilst leaving all Sequence, GOP, Picture and Slice headers intact, it seems that we need to build most of an MPEG-1 Video Stream decoder in order to locate the necessary bytes for encryption. Fortunately, the multi-layered structure of an MPEG-1 Video Stream, as depicted in Figure 4-2, allows a more streamlined approach. All MPEG-1 Video Stream Sequences start with a sequence header and consist of one or more *complete* Groups of Pictures. This means that when searching for GOP blocks, it is possible to ignore Sequence Headers and look solely for GOP headers – with the GOP ending when the next Sequence Header Code, Group Start Code or Sequence End Code is encountered. The occurrence of valid Picture or Slice Start Codes do not signal the termination of a GOP since a GOP can contain more than one Picture or Slice segment, noting also that a GOP sequence does not span between two sequences nor contain another GOP sequence. (Mitchell et al., 1996)



Error: An MPEG-1 Video Stream Slice cannot be split across Picture Bit Sequences

Error: An MPEG-1 Video Stream Slice cannot contain another Slice Bit Sequence

Figure 4-3: Mechanics of the MPEG-1 Slice Bit Sequence

If this argument is continued to the Macroblock Layer, it can be shown that a Slice Sequence within the Video Stream also contains one or more *complete* Macroblocks, see Figure 4-3. A Slice Sequence consists solely of a Slice Header followed by a series of Macroblocks, does not span between two Picture Sequences, nor contain another Slice Sequence. As such, it is possible to define a Slice Sequence as beginning with a valid Slice Header code and then continuing until the next valid MPEG-1 Start Code is found, remembering that anytime the byte-aligned binary sequence (0x00 0x00 0x01) is located within the Video Bit Stream, it signifies a valid MPEG-1 Start Code. It is therefore possible to summarise the task of encrypting the Macroblocks to the algorithm proposed in Figure 4-4 where the Video Bit Stream is scanned to locate a valid Slice Header, the Header is left intact and all following bytes (making up the Macroblocks) are encrypted until the next (0x00 0x00 0x01) sequence is located, these three bytes must remain unencrypted to preserve the Start Code within the Video Stream.

```
Encrypt MPEG1 Video Stream()
{
     while (DataLeftInStream())
     {
          while (Next_3_Bytes() != [0x00, 0x00, 0x01])
          {
               Read_NextByte();          /* Do not encrypt         */
          }
          Read_3_Bytes();                /* Read 0x00, 0x00, 0x01  */
          if (NextByte IN [0x01…0xaf])
          {                              /* Found valid slice head.*/
               Read_SliceHeader();
               while (Next_3_Bytes() != [0x00, 0x00, 0x01])
               {                         /* Within Macroblock Data */
                    Encrypt_NextByte();
               }
          }
     }
}
```

Figure 4-4: Algorithm to Encrypt Macroblocks Within an MPEG-1 Video Stream

A State Machine to accomplish this task is shown in Figure 4-5, the first sequence of four states are used to locate a valid Slice Start Code, the machine enters the fourth and final state only if the sequence (0x00 0x00 0x01) is encountered, from here a valid Slice Code has the next byte in the range 0x01-0xaf, this input progresses the State Machine to the Slice Header Processing States where the Slice Extra Information bits are processed until the end of the Slice Header is encountered. The second section of the State Machine consists of eight states that locate the end of the Slice Header – each byte of extra information is encoded as nine bits, the ninth bit equal to 0 if there is more data. Each of these eight states checks the state of one of the eight input bits – if it is set then the machine cycles through the states bypassing the extra information, otherwise the end of the Slice Header has been found and the State Machine can progress to the third section, within the Slice Data. When the State Machine locates a valid Slice Header, it enters the sixth of the eight states of the second section, this is because the Slice Start Code is succeeded by a five bit quantiser scale value, the sixth bit of this byte then indicates the presence of extra information.

The third and final section of the State Machine signifies that the input stream being processed forms the Slice Data or the Macroblocks within the Slice, whilst within this section of the State Machine, all input bytes must be passed through the cipher after being processed. The final section of the State Machine is similar to the first except that we are now looking for any valid MPEG-1 Start Code. When the (0x00 0x00 0x01) Sequence is located, the State Machine jumps to the final state of the first section of the State Machine. This state will then check if the new Start Code indicates another Slice Header which passes control to section 2, or another MPEG-1 Start Code, therefore transferring control back to the initial state. Finally, when the State Machine traverses from section 3 back into section 1, the previous three bytes must be returned to their initial values of (0x00 0x00 0x01), thereby preserving the original Start Code for the decryption process. Another important issue is that the cipher itself never produces the byte-aligned (0x00 0x00 0x01) sequence as this would introduce a false Start Code – in the decryption phase this would mean an incorrect location of the end of the Slice and corruption of the original Video Stream.

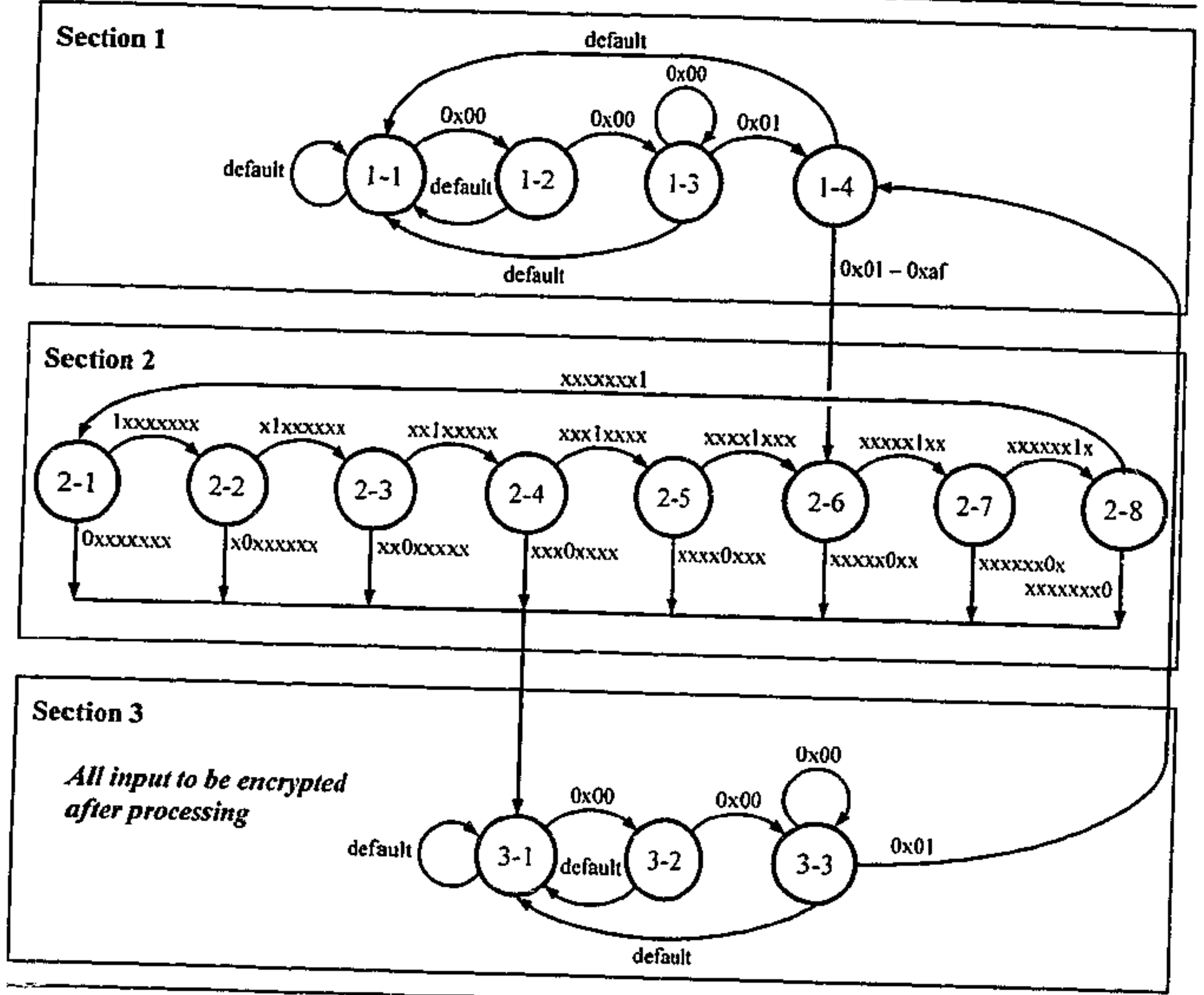


Figure 4-5: State Machine to Encrypt an MPEG-1 Video Stream

4.2.2.2 Designing the Prototype Cipher

The design of the Video Stream Partial Selection State Machine requires a cipher that can encrypt an *n*-byte, byte-aligned sequence; the simplest technique available to produce this result is to encrypt the existing data in single byte blocks. Since the data to be encrypted is byte aligned, encrypting byte-sized blocks of data allows us to produce an efficient software solution as bit level operations are generally inefficient in software. Also, by not encrypting in larger block sizes (16, 32 or 64 bits) we avoid potential alignment problems at the beginning and end of the sequence to be encrypted. We are now faced with the problem of designing a cipher that encrypts a series of consecutive bytes, yet does not produce the output sequence (0x00 0x00 0x01). The generalised output function of a cipher that meets these preconditions is shown in Figure 4-6.

$$f(x): \quad \begin{array}{ll} \text{byte} & \Rightarrow \text{byte} \\ 0\text{x}00 & \rightarrow 0\text{x}00 \\ 0\text{x}01 & \rightarrow 0\text{x}01 \\ [0\text{x}02 \ldots 0\text{xff}] & \rightarrow [0\text{x}02 \ldots 0\text{xff}] \end{array}$$

Figure 4-6: Generalised Cipher Function for MPEG-1 Video Stream Encryption

The functionality of this cipher is as follows:

- **If the input byte is 0x00 or 0x01, then the cipher outputs the same byte** – this ensures that if the cipher encounters a valid MPEG-1 Start Code then that code will remain in the output stream untouched.

- **If the input byte is in the range 0x02-0xff, then the cipher must output a byte within the range 0x02-0xff** – the security of the cipher rests in how random this transformation is, the restriction ensures that we do not produce any 0x00 or 0x01 bytes within the stream and therefore any false Start Codes.

A simple cipher that meets these restrictions is proposed in Figure 4-7. This particular cipher uses an 8-bit key (*k*). While such a short key offers minimal protection against a brute force attack, with only 128 (2^7) keys needing to be tried on average, this cipher will suffice to demonstrate that the encrypted stream can be installed to and streamed from an existing server, as well as being played back using existing decoder technology. The functionality of this cipher is simple, the cipher functions by XORing the input byte with *k*. The problem with this approach is that if the input byte is either equal to *k* or (*k* ⊕ **0x01**), then the output bytes will respectively be **0x00** and **0x01** – also, if the input byte is equal to **0x00** or **0x01**, then the output bytes will respectively be *k* and (*k* ⊕ **0x01**). This problem is easily solved by not encrypting the bytes **0x00**, **0x01**, *k* and (*k* ⊕ **0x01**). Finally, the proposed cipher is a Private Key Cipher and the same function can be used to decrypt the data, with **0x00**, **0x01**, *k* and (*k* ⊕ **0x01**) bytes still retaining their values, and encrypted bytes being XORed with *k* to retrieve their original values.

$$f(x,k): \quad \begin{array}{ll} \text{byte} & \Rightarrow \text{byte} \\ 0\text{x}00 & \rightarrow 0\text{x}00 \\ 0\text{x}01 & \rightarrow 0\text{x}01 \\ k & \rightarrow k \\ k \oplus 0\text{x}01 & \rightarrow k \oplus 0\text{x}01 \\ x & \rightarrow x \oplus k \end{array}$$

Figure 4-7: Simple Cipher for use in MPEG-1 Video Stream Encryption

4.2.2.3 Support for Indexed and High-Speed Playback Modes

It is important to also consider decryption of the Video Stream during indexed and high-speed playback. As discussed in Chapter 2, indexed playback involves starting playback of the Video Stream from any GOP within the complete bitstream, while high-speed playback involves playing back a newly constructed Video Stream generated from the I-Frames within the original bitstream. This implies that the Cipher Module must be able to resynchronise itself for each of these situations, resynchronisation must occur at the start of each GOP to allow for indexed playback as well as for each individual I-Frame to support high-speed playback.

Due to the simplicity of the proposed Cipher, resynchronisation becomes a mute issue. As the Video Stream structure is maintained during these special playback modes, the partial selection scheme ensures that the same bytes are always selected for decryption. Since all bytes are encrypted in the same way, there is no issue of resynchronisation. It will however be important to consider this issue when designing a more secure cipher.

4.2.3 Summary of MPEG-1 Video Stream Encryption

By combining the cipher from Section 4.2.2.2 with the State Machine designed in Section 4.2.2.1, we will end up with an encrypted MPEG-1 Video Stream of exactly the same size as the original stream. Since the encrypted stream size hasn't changed, this procedure can be implemented inplace within an MPEG-1 System Stream, ensuring that the requirement introduced in Section 4.1.2 is met. Also, due to the design of the cipher module, we can decrypt the MPEG-1 Video Stream using the same algorithm. As for encryption of the MPEG-1 System Stream, certain parts of the stream are not encrypted and left as plaintext, again, the content of this information is irrelevant to the media that needs to be protected and consists of known or easily guessed information that is used to set up the decoder to facilitate playback. The overriding reason for not encrypting this data rests in the design of Streaming Video Servers, since a server must be able to locate the start of a GOP for indexed playback as well as the start of all I-Frames for high-speed playback, it is essential that the basic format of the MPEG-1 Video Stream remains unchanged so that the necessary information can be accessed.(Anderson, 1996; Lin et al., 2001; Shanableh and Ghanbari, 2001)

Ease of decryption at the client end is also an important consideration. In simple playback mode this task involves repeating the encryption process, however the task becomes more complex when considering indexed or high-speed playback. The partial selection scheme ensures that the overall structure of the Video Stream is maintained when playback in these modes occur, but it is also necessary to consider re-synchronisation of the cipher module when skipping frames during playback. Due to the simplistic design of the prototype cipher module, this requirement is not an issue but must be considered when designing a secure MPEG-1 Video Stream encryption algorithm.

4.3 MPEG-1 Audio Stream Encryption

The MPEG-1 Audio Stream is formatted differently from the MPEG-1 Video Stream – there is no concept of different layers of encoding. The data is stored in a series of frames that start with a frame header of four bytes in length (Mitchell et al., 1996; Haskell et al., 1997; Pan, 1993). While the entire contents of the Audio Stream can be encrypted in theory, at least one video streaming server implementation will not install an MPEG-1 file with the Audio Stream modified in this way. As such I will present a cipher algorithm whereby the frame headers of the Audio Stream are left intact, allowing the encrypted bitstream to be installed onto a streaming server, while ensuring that the audio content is protected.

4.3.1 Examination of the MPEG-1 Audio Stream

The MPEG-1 Audio Stream format is explained in its entirety in Appendix A. Unlike the MPEG-1 Video Stream, it is not made up of a number of layers but rather a single layer of consecutive frames which are headed by a frame header. The frame header is four bytes in length and contains information such as which of the three encoding algorithms should be applied to the data, the encoded audio bit-rate, and other minor information (Haskell et al., 1997; Pan, 1995; Shlien, 1994; Noll and Pan, 1997). The interesting thing to note is that there is no timestamp or other indexing information present in the Audio Stream, however the frame headers are equally spaced throughout the stream and a single frame always contains data that decodes to a known time span of raw audio data. In the case of decoding an MPEG-1 Audio Stream in its own right, time indexing is performed by jumping to a multiple of the frame size to locate an Audio Frame header.

Since time indexing information is not encoded in the Audio Stream itself, it must be encoded within the System Stream Packet Headers. This has an implication when considering the decryption of the Audio Stream during indexed playback. We will be required to resynchronise the cipher module based on the time index during playback, information that cannot be acquired from the Audio Stream itself. This means that the cipher module must be resynchronised by some other trigger point. However, if we consider a cipher with the same complexity as that employed in the MPEG-1 Video Stream cipher, where each byte is XORed with an 8-bit key, resynchronisation is no longer as issue. When streaming video in one of the high-speed playback modes (fast forward or rewind), there is no audio data sent and the stream is watched in silence. As such, we do not need to consider decrypting an Audio Stream during high-speed playback.

It appears that none of the data within the Audio Stream is of importance to the Streaming Server and as such we can simply encrypt the entire Stream. Indeed, this was the approach taken when developing the first prototype system, but when an attempt was made to install the encrypted MPEG-1 file on the SGI MediaBase server, the installation failed and the Server reported that the file was not a valid MPEG-1 file. While the Streaming Server does not require any information from the encoded Audio Stream in order to stream the stored file, it appears that at least this system checks to see if the Audio Stream is valid. At this point the prototype was changed to leave the frame headers intact and encrypt only data encoded within the frame data.

4.3.2 Processing the MPEG-1 Audio Stream

A state machine to parse and encrypt the MPEG-1 Audio Stream is much simpler than one to parse the Video Stream. We simply need to read and parse the frame header, before encrypting each byte until the next frame header is encountered. A frame header consists of four consecutive bytes beginning with the 12-bit byte aligned sequence (1111 1111 1111). The state machine needs to parse input bytes to look for this sequence as well as read the following 20 bits so that the entire frame header is processed. While the amount of audio data following the frame header can be determined from the frame header (from the encoding algorithm and bit rate fields), this calculation can be somewhat laborious. It is much easier to continuously encrypt bytes of data until the next frame header is encountered. This can be accomplished using a similar approach to encrypting data in an MPEG-1 Video Stream where all bytes are encrypted until the next MPEG-1 Header sequence is encountered, taking care to ensure that a false Audio Frame Header is not accidentally introduced.

A method of ensuring that an Audio Frame Header is not accidentally created is to avoid producing the output byte 0xff while encrypting data: via a similar XOR encryption scheme to that used in encryption of the Video Stream. If the byte to be encrypted is equal to the binary inverse of the 8-bit key value, then the byte is left unencrypted, as the XOR would result in an encrypted byte value of 0xff. If this output byte is never produced, then the byte aligned 12-bit sequence (1111 1111 1111) can also never be produced. Similarly, an input byte of 0xff is left as is to ensure that each 8 bit value has a unique mapping. The details of this cipher are outlined in Figure 4-8.

$$
\begin{array}{lll}
f(x,k): & \text{byte} & \Rightarrow \text{byte} \\
 & 0\text{xff} & \rightarrow 0\text{xff} \\
 & k \oplus 0\text{xff} & \rightarrow k \oplus 0\text{xff} \\
 & x & \rightarrow x \oplus k
\end{array}
$$

Figure 4-8: Simple Cipher for use in MPEG-1 Audio Stream Encryption

A State Machine that will correctly parse an Audio Stream is shown in Figure 4-9. This State Machine will function with all three MPEG-1 Audio Encoding formats and uses the cipher from Figure 4-8 to encrypt the actual audio data. The first sequence of four states will read and process the MPEG-1 Audio frame header. Once the final state of this sequence is exited, the header has been processed and the audio data block follows. The second section of the State Machine consists of two states which are used to locate the start of the next frame header – while input bytes that are not equal to 0xff are processed, we are still within the audio data block and the input byte is encrypted using the cipher from Figure 4-8. If an input value of 0xff is encountered then it is a candidate for an audio frame header and we progress to the second state. This state will check whether the next four bits are set which will indicate the start of the next frame header, if this is the case we progress to State 1-3 to complete processing the header, otherwise we return to the previous state to continue processing data bytes. Data bytes processed whilst in state 2-2 are not encrypted, this avoids the problem whereby the four most significant bits of this byte may accidentally be set. As mentioned in the previous section, resynchronisation of the Audio Stream cipher is not necessary since all frames are encrypted in the same way, however this issue must be addressed when designing a more secure cipher.

4.3.3 Summary of MPEG-1 Audio Stream Encryption

As for encryption of the MPEG-1 Video Stream, when the State Machine from Figure 4-9 is combined with the cipher from Figure 4-8, we end up with an encrypted MPEG-1 Audio Stream of exactly the same size as the original stream. As the stream size is the same, the encryption can be performed in-place within the MPEG-1 System Stream. The majority of the binary stream is encrypted as the Audio data blocks are encrypted while the frame headers are left unencrypted. The stream can be easily decrypted at the client end as the same process is applied to the MPEG-1 Audio Stream after it has been de-multiplexed from the System Stream. As there is no audio playback or streaming in high-speed playback modes, this is not an important consideration, however decryption of the Audio Stream during indexed playback must be addressed. In the case of the prototype system, this issue is conveniently ignored as each Audio Frame is encrypted using the same 8-bit key and no resynchronisation is necessary. However, when considering the decryption of an Audio Stream with a more secure cipher, it is important to remember that the cipher must be resynchronised to correctly function during indexed playback.

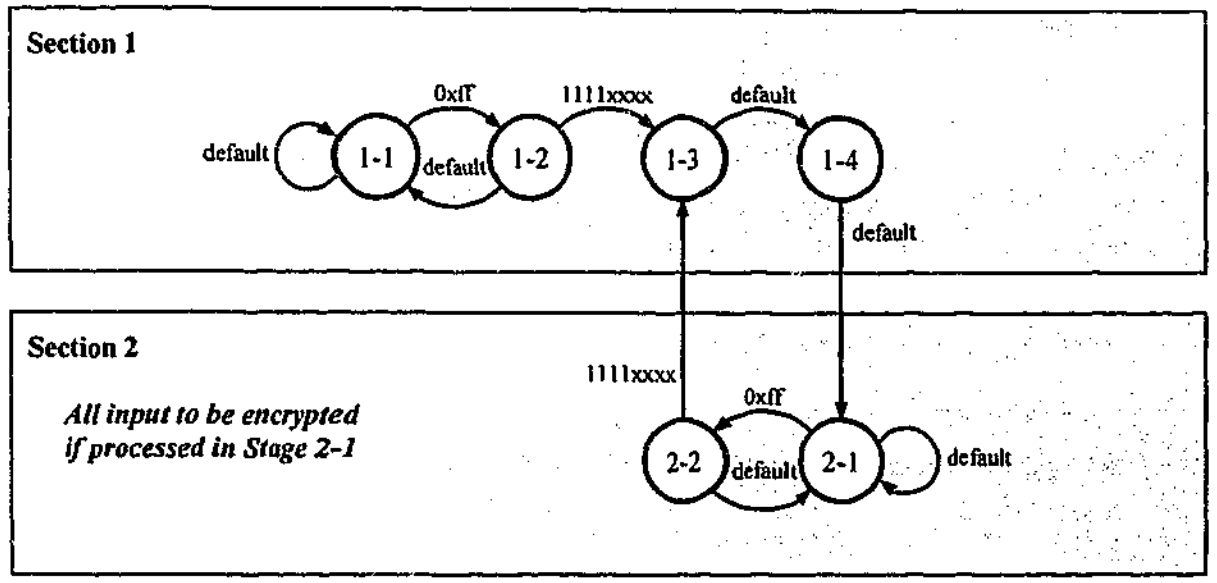


Figure 4-9: State Machine to Encrypt an MPEG-1 Audio Stream

4.4 Prototype System Testing

Before proceeding with the design of a secure MPEG-1 Cipher, it is necessary to build the prototype system and ensure that it functions as expected. It is also important to ensure that the encrypted binary stream can be successfully installed on a range of servers as well as streamed from those servers without any errors occurring. This means that the streaming server must be capable of streaming the encrypted video file in a variety of modes including normal, high-speed and indexed playback. Once this initial testing has been completed, it becomes necessary to incorporate the decryption module within a client player. Success of this trial requires:

- That there exists enough CPU cycles to decrypt and decode the video stream in real-time.
- That the mechanism for partial encryption of the MPEG-1 Video and Audio Streams is valid and that the encryption process can be reversed in a variety of playback modes, including normal, high-speed and indexed playback.

This section discusses and summarises the results of the prototype system testing, a complete set of results can be found in Appendix D.

4.4.1 Trial Conditions

This section describes hardware and software platforms used to conduct the trials, as well as list the input files used.

4.4.1.1 Input Files

Six different test MPEG-1 files were used for trialling the prototype cipher, these test bitstreams were:

- **tennis.mpg** – A standard MPEG-1 test sequence, regularly used in MPEG-1 experiments. This sequence is an MPEG-1 Video Stream of short duration (approx. 4 seconds)

- **flowg.mpg** – Another standard MPEG-1 test sequence. This sequence is also an MPEG-1 Video Stream of short duration (approx. 4 seconds).

- **us.mpg** – a slightly longer MPEG-1 Video Stream (approx. 12 seconds). This sequence is also regularly used in research, but not as common as the previous two sample files. This sequence was chosen because it is an excerpt from a motion picture and therefore represents the type of content we wish to protect. All three of these test files were downloaded from http://peipa.essex.ac.uk/ipa/src/formats/mpeg/stanford

- **Chicken.mpg** – A sample MPEG-1 System Stream encoded by Microsoft engineers and provided on the CD containing the initial release of the Microsoft NetShow Theatre Streaming Server.

- **Monash Nursing.mpg** – MPEG-1 System Stream encoded using the Siemens Eikona MPEG-1 Encoder at Monash University. This sample is encoded at a high bitrate for MPEG-1 (2.7Mb/s).

- **Diablo2_5.mpg** – MPEG-1 System Stream of a short movie. This sequence was encoded by Monash University using an Optibase Hardware Encoder as part of a VoD trial run at Monash University.

4.4.1.2 Test Applications

Several test applications were written in order to run the trials. A description of the development and usage of these applications can be found in Appendix C. These applications can be used to:

- **MPEG1 Cipher** – Encrypt or decrypt an MPEG-1 file on disk. The application can encrypt either an MPEG-1 System Stream or MPEG-1 Video Stream file. Statistics are provided on the cipher process.

- **DirectShow MPEG Cipher Filter** – A DirectShow Transform Filter to enable real-time decryption of MPEG-1 Video and Audio Streams within the Microsoft DirectShow environment. Allows further development of other applications to decrypt and playback an encrypted file on disk, or to stream, decrypt and playback an encrypted video streamed from a server with a DirectShow capable Source Filter.

- **DirectShow Stream Playback Application** – Allows playback of an encrypted MPEG-1 Stream from either the Microsoft NetShow Theatre or the SGI MediaBase 3.1 Streaming Server platforms. The DirectShow source filter for MediaBase does not support the high-speed playback modes.

- **MediaBase Playback Application** – Allows real-time streaming and decryption of an encrypted MPEG-1 Stream from the SGI MediaBase 3.1 Streaming Server. The decrypted bitstreams are saved to disk, later playback can be used to confirm functionality. This application does support the high-speed playback modes.

4.4.1.3 Test Platforms

Various test platforms were used for different purposes. Three different Streaming Server platforms were used to test functionality of the cipher during streaming:

- Microsoft NetShow Theatre

- SGI MediaBase 3.1

- Apple QuickTime or Darwin Streaming Server

Real-Time Decryption and decoding trials were executed on two different hardware platforms:

- 233 MHz Pentium II Workstation with 384 MB RAM, running Windows 2000

- 1.6 GHz Pentium 4 Workstation with 256 MB RAM running Windows 2000

4.4.2 Trial Results

A series of different trials were run to prove the viability of the prototype system, culminating in a final trial to stream, decrypt and decode and encrypted stream installed on a streaming server in real-time. The trials performed are outlined in the following sections.

4.4.2.1 Percentage of the MPEG-1 File Encrypted

The first experiment performed on each test stream encrypted file was to calculate the actual percentage of the MPEG-1 file selected for encryption. This is primarily used to indicate the required speed of the cipher implemented in a secure MPEG-1 Cipher – since the required CPU load for decryption is directly related to the amount of data that needs to be decrypted. The file encryption program as written produces these results as a set of statistics reported to the user upon completion of the file encryption task. The complete set of results is presented in Table D-2, a summary of these results is presented in Table 4-1. The important result is the high proportion of the test files that are selected for encryption, even though practically the entire MPEG-1 Video Stream format is selected as plaintext and the entire System Stream is selected as plaintext. This indicates that the information encoded as macroblocks forms a large proportion of the encoded bitstream. The conclusion to be drawn from this is that the final cipher as designed in Chapter 5 must be able to process an encrypted stream at a very high percentage (> 95%) of the bitstream playback rate while at the same time ensuring there are enough remaining CPU cycles to decode and display the bitstream to the user.

Filename	% Video Stream Selected	% Audio Stream Selected	Total % Selected
tennis.mpg	99.5%	N/A	99.5%
flowg.mpg	99.8%	N/A	99.8%
us.mpg	98.6%	N/A	98.6%
Chicken.mpg	99.2%	98.7%	98.2%
Monash Nursing.mpg	99.6%	99.2%	98.2%
Diablo2_5.mpg	99.5%	99.2%	96.3%

Table 4-1 Proportions of Test Bitstreams Selected for Encryption

4.4.2.2 Is the Encryption Process Repeatable and Reversible

The second series of tests were to confirm that an encrypted MPEG-1 file could both be consistently generated, and that it could be restored to its original state. The first determination would provide confidence that the second step would be possible. This test would therefore validate that it would be possible to decrypt a protected video stream for playback, and that the proposed encryption scheme is functional. This test involves two parts – the first is to ensure that repeated encryptions of a test MPEG-1 sequence produces the same encrypted file as output. The second test would confirm that each byte in the decrypted bitstream compares exactly with the corresponding byte in the plaintext sequence. A complete description of the sequence of operations required to verify that the encryption process is repeatable is explained in Section D.3.1 along with the results of these tests which show that the encryption process is indeed repeatable, as is to be expected when examining the design of the prototype cipher as presented in this chapter.

At this point it became apparent that the number of bytes selected for encryption compared against the number of bytes actually encrypted was less than the expected amount. Assuming an even distribution of byte values within the bytes selected for encryption, and given that there are four byte values which will not be encrypted, we would expect $^{252}/_{256}$ = 98.43% of the selected bytes encrypted. Instead we find that fewer than 95% of the selected bytes are encrypted. This anomaly can be explained by two factors:

- The first three bytes of the next header (0x00-0x00-0x01) are actually counted as selected for encryption by the cipher program.
- The proportion of 0x00 and 0x01 bytes in the source streams is higher (> 4%) than for a purely random distribution (< 1%). The remaining byte values in the bitstreams are relatively evenly distributed.

The complete results are shown in Table D-3 and Table D-3.

The second test for reversibility was to prove two precepts, one, that the original plaintext bitstream could be retrieved if the correct key was used, and two, that the original plaintext could not be retrieved if the incorrect key was used. The prototype cipher allows for 256 possible keys, 255 of which actually modify the plaintext (key 0x00 causes no change to the plaintext). Rather than perform the test using all keys (which would lead to 256^2 possible permutations), a subset of four keys was selected – requiring 16 separate tests for each of the six test bitstreams (96 separate results). The four keys chosen for test purposes were 0xff, 0x00, 0x4a and 0x42 – the first key represents inverting all bits of the bytes selected for encryption, the second performs no encryption, while the third and fourth key form the ASCII code for my initials. A complete tabulation of the results can be found in Table D-5. The results proved consistent for each of the six test files, a summary is presented in Table 4-2. From this, we can see that all files were successfully decrypted when the encryption key was reapplied – also, attempting to decrypt a file with a different key to which it was encrypted resulted in a bitstream that was not equivalent to the original file. All 96 generated files were passed through both software and hardware based MPEG-1 decoders. All correctly decrypted files were successfully played back. Files decrypted with the incorrect key were processed as follows:

- **Software Decoder** – Video decoded as a blank screen, audio consists of high-frequency "cheep" type noises.
- **Hardware Decoder** – Video decoded as random green and monochromatic square blocks, with similar audio output to the software decoder.

	Decryption Key			
Encryption Key	0xff	0x00	0x4a	0x42
0xff	☑	☒	☒	☒
0x00	☒	☑	☒	☒
0x4a	☒	☒	☑	☒
0x42	☒	☒	☒	☑

Table 4-2 Reversing the MPEG-1 System Stream Encryption

We can conclude that the encryption process is reversible and that the original plaintext can be obtained if the correct key is used.

4.4.2.3 CPU Requirements for Encryption/Decryption

The third test performed was to calculate the CPU time that was required to execute the cipher. The aim of this test is to show that the decryption process would not place an undue load on the client computer when decrypting and decoding the MPEG-1 Stream for playback. CPU requirements were measured using two different techniques, the maximum rate and the difference technique.

The maximum rate technique involves timing the encryption process while encrypting a file on disk. The measurement cannot be performed by simply timing the encryption program as the application is also performing disk I/O in reading the original file and writing the modified file – in fact, the time spent performing disk I/O forms a major portion of the execution time of the encryption program. In order to correctly calculate the execution time required, the test procedure needs to take the disk I/O time into account so that a valid measurement could be made of the actual time required to execute the cipher. Also of issue when measuring the time taken to perform encryption is the fact that the test platform is running a multi-tasking Operating System, meaning that 100% of available CPU cycles would not be given to the executing test application. It becomes necessary to also take the CPU load into account when calculating the actual time required execute the cipher only.

The procedure to calculate approximate times and CPU loads for encryption/decryption, taking into account disk I/O and CPU allocation by the OS, is described in Section 0. There are no results for the first three test files – the time taken to encrypt the file was too short to be able to make meaningful measurements of both time and CPU Load. The three longer test bitstreams show that the prototype cipher can encrypt an MPEG-System Stream at between 60-80 Mb/s on the Pentium II Test Platform, and between 260-500 Mb/s on the Pentium 4 Test Platform. Of more interest is the required free CPU Load to perform the encryption/decryption at real-time playback speeds, the results show a required load of between 2.5% and 4% on the Pentium II and about 0.5% available CPU cycles on the Pentium 4. The results are summarised in Figure 4-10.

The same figures are then obtained using a difference method. In this case, we measure CPU Load during both playback of the plaintext bitstream and during simultaneous decryption and playback of the encrypted bitstream. The difference between the two figures provides the CPU Load required for decryption only, which can then be compared with the previously calculated result. The complete results are shown in Table D-9 and Table D-10, and summarised in Figure 4-10. In the case of both the maximum rate method and difference method, less than 6% of CPU cycles are required by the prototype cipher on the Pentium II platform, while under 1.5% of CPU cycles are required on the Pentium 4 platform.

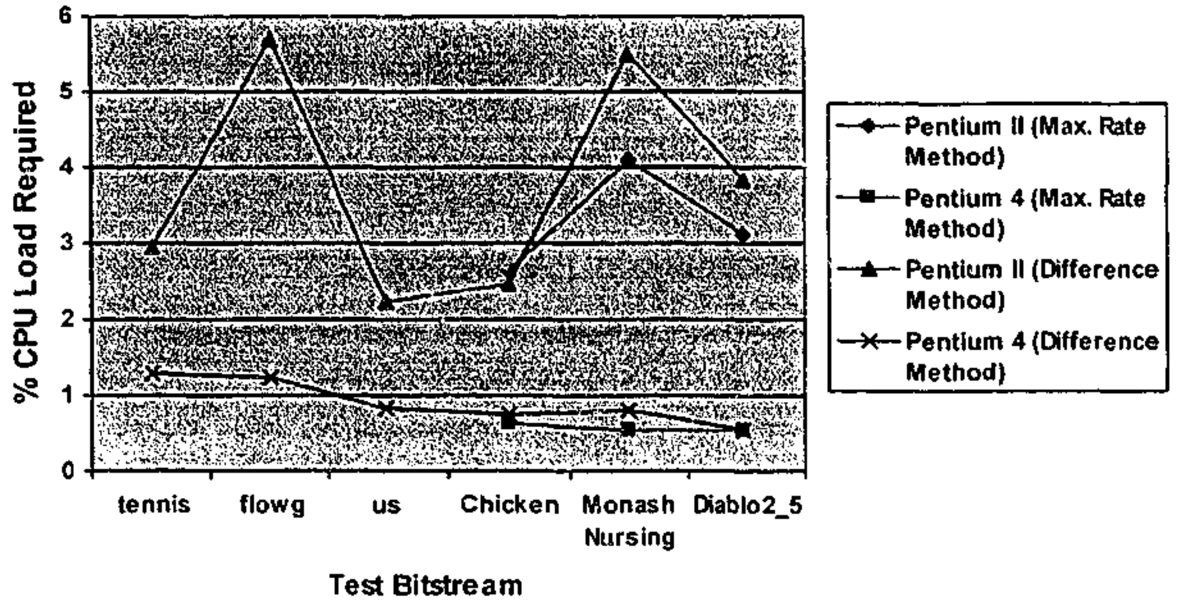


Figure 4-10: Performance Results Using the Prototype Encryption Scheme

Table 4-3 shows the CPU load required for decoding and full-screen playback for each test file on both test platforms, noting that only "Chicken.mpg" could be played full-screen on the Pentium II platform. If we compare these results with the graph in Figure 4-10, we see that the cipher requires about 10% of the CPU load required to decode and display the corresponding plaintext MPEG-1 Stream. The simplicity of the actual cipher – XOR – suggests that these results reflect the processing requirements of the stream selection parsers that determine which bytes are to be encrypted. These figures show that both processes (decryption and decoding) are able to function in parallel and that there is some leeway on available CPU cycles – especially on the faster Pentium 4 platform – to implement a more complex cipher offering greater security.

Filename	CPU Load for Plaintext Playback on Pentium II	CPU Load for Plaintext Playback on Penitum 4
tennis.mpg	47.20 %	11.93 %
flowg.mpg	59.17 %	16.80 %
us.mpg	41.30 %	8.10 %
Chicken.mpg	87.67 %	12.10 %
Monash Nursing.mpg	56.13 %	15.17 %
Diablo2_5.mpg	59.60 %	15.03 %

Table 4-3 Required CPU Load for Plaintext Playback

4.4.2.4 Verification of Functionality with Existing Streaming Video Servers

The ultimate test of functionality is to successfully decrypt and decode an encrypted video that is being streamed from a Streaming Server. This proves the viability of the approach for real-time decryption of streaming video as opposed to decryption of stored content for later playback. While impossible to test the cipher on all Streaming Servers, I did have access to a range of Servers which could be used to verify that the file would install and be successfully streamed from them. These servers were:

- A Cluster of Intel Based Windows NT Workstations running Microsoft NetShow Theater
- SGI Challenge L running MediaBase 3.1
- An Linux Workstation running Apple Darwin Streaming Server

The first set of tests involved the Microsoft NetShow Theatre product and consisted of the installation of the encrypted assets onto the server and the subsequent streaming and real-time decryption and playback in all supported playback modes. A full description of the tests and results can be found in Appendix D. Installation of MPEG-1 Video Streams is not supported by Microsoft NetShow Theatre and therefore test results are limited to the three MPEG-1 System Stream test files. The first step involved the installation of the plaintext version of each test bitstream as well as copies of the same bitstream encrypted with all three test encryption keys. All files were recognised as valid MPEG-1 bitstreams by the server and the subsequent installation proved successful.

Following installation of all test bitstreams, the assets were streamed to a client player using the *"StreamCipher.exe"* application. This application can be used to stream an MPEG-1 bitstream from a Microsoft NetShow Theatre Server using the DirectShow framework to decrypt and decode the stream while providing access to all available playback modes. The same application can be used to stream a plaintext MPEG-1 asset by selecting to use *"No Cipher"* on the graphical user interface. The tests performed using this application involved random – via user interaction with the application – selection of different playback modes and indexed playback from randomly selected timestamps. Random selection of playback modes ensured that changing between all playback modes resulted in correct decryption and playback while random jumps through the bitstream in all available playback modes ensured that indexed playback was correctly supported by the cipher. Successful decryption and subsequent playback was confirmed both visually and aurally by observing the resultant video played back on screen and through the computer sound system. When streaming from the Microsoft NetShow Theatre Server, the cipher correctly decrypted the bitstream under all test conditions.

The second set of tests involved the SGI MediaBase 3.1 product, like with the NetShow Server, the procedure began with the installation of the encrypted assets onto the server and concluded with the streaming, real-time decryption and playback of those assets in all supported playback modes. A full description of the tests and results can be found in Appendix D. In the case of MediaBase, installation of MPEG-1 Video Streams is supported by the server and thus all six test files were able to be tested for installation. All test files were recognised as valid bitstreams by the server and the subsequent installation proved successful.

Two test applications could be used to stream and playback the encrypted assets installed on the MediaBase Server. The DirectShow based application could be used to provide access to normal speed and indexed playback only as all high-speed playback modes is not supported by the MediaBase DirectShow source filter. Also, the *"StreamCipher.exe"* application was specifically written to support only MPEG-1 System Stream assets and could not be used to test playback of the three MPEG-1 Video Stream test files. However, the length of these test files mitigate against testing both high-speed and indexed playback as their duration is too short to provide any meaningful results.

The second test application, *"SGIStreamCipher.exe"* can be used to test functionality of the cipher during high-speed playback modes. This application produces a series of output files which consists of the retrieved bitstream after being passed through the cipher. Functionality can be verified via playback of these files with a standard MPEG-1 player. Again, the application was written specifically to support only MPEG-1 System Stream assets and so testing could only be performed on the three shorter test files. The same procedure as for the NetShow Theatre server was used, to employ the random selection of playback modes and indexed jumps and the same results were observed.

When using the DirectShow enabled player, the encrypted bitstream was correctly decoded through both paused playback and indexed playback. When using the API enabled player, all generated output files were correctly played back through Windows Media Player. Files created during either of the two high-speed playback modes resulted in MPEG-1 Video Streams that played back the original video at high-speed.

The final server under test was the Apple Darwin Streaming Server, and by inference, also the Apple QuickTime Streaming Server since they use the same protocols to stream video across a network. The installation procedure for the Darwin Server is somewhat different, where the test files had to first be *"Hinted"* using the Apple QuickTime application. Any movie file that can be successfully hinted can be correctly installed onto an Apple Darwin or QuickTime server. A full description of the procedure can be found in Appendix D. The test results showed that all six test files were able to be hinted and subsequently installed onto the server.

Testing the streaming and decryption functionality with the Apple product would involve the development of an MPEG-1 Cipher module similar to the DirectShow Cipher Filter but using the Apple QuickTime SDK instead. This software was not developed and therefore streaming from the Apple server was not able to be tested. However, documentation verifies that the Server can stream – using the RTP and RTSP protocols – any hinted movie file and that the resultant file can be retrieved at the client end. This implies that streaming of the encrypted bitstream and the subsequent retrieval at the client is guaranteed, as long as the encrypted bitstream can be successfully hinted. The Apple Server products do not support any of the high-speed playback modes.

The results are summarised for each server type in Table 4-4, and described in full in Section D.3.4. It was verified using the NetShow Theatre server that all playback modes were supported by the Cipher design through correct decryption and playback. It was verified using the MediaBase 3.1 server that all playback modes were supported by the cipher design – real-time decryption and playback was checked for normal speed playback modes and real-time decryption with delayed playback was checked for high-speed playback modes. Finally, installation of encrypted assets was verified on the Apple QuickTime and Darwin Streaming servers.

		Playback Mode				Indexed Playback Mode			
Server	Installation	▶	‖	▶▶	◀◀	▶	‖	▶▶	◀◀
NetShow Theatre	☑	☑	☑	☑	☑	☑	☑	☑	☑
Mediabase	☑	☑	☑	☑	☑	☑	☑	☑	☑
Quicktime	☑								

A blank entry in the table signifies that the functionality was not tested.

Table 4-4 Streaming an Encrypted MPEG-1 File from a Streaming Video Server

4.5 Summary

The prototype encryption scheme was implemented in a variety of applications. One implementation allowed for the encryption of an MPEG-1 file stored on disk. These encrypted files demonstrated the ability to install onto a range of servers. Other applications were used to successfully stream the encrypted bitstreams from these servers in a variety of different playback modes, which was then successfully decrypted and displayed on a variety of end stations. Typically, less than 6% CPU load is required on a 233 MHz Pentium II platform for byte selection and successful decryption, contributing less than 10% to the total video streaming and display load.

Having shown that the streamed video could be successfully decrypted and decoded in real time, the next issue to resolve is to upgrade the level of protection afforded by the cipher. In the next Chapter I will further develop the novel partial selection scheme presented here to increase the security of the cipher. It is important that whatever algorithm is finally employed, that it break none of the requirements for the MPEG-1 Cipher that have so far been outlined.

Chapter 5

A Novel MPEG-1 Partial Encryption Scheme

In the previous chapter I presented a new prototype MPEG-1 Encryption scheme that met the requirements outlined in Chapter 2. The aim of this process was to develop a simple scheme that would allow an encrypted file to be installed on a variety of existing video streaming servers and enable playback of the encrypted video. This was demonstrated through the installation of the encrypted video of a range of test servers and playback by client playback applications capable of decrypting the streamed video. I showed that the encrypted video could be successfully played back in a variety of playback modes including pause, indexed playback and high-speed playback.

Only one of the requirements from Chapter 2 was not met and that was that the encryption be difficult to break. In this chapter I will extend on the prototype encryption scheme developed in Chapter 4 such that the issues of security are met, while maintaining the flexibility of the prototype scheme. I begin the chapter by looking at the question of selecting a secure cipher that would be suitable for use in streaming video. This implies certain restrictions, as the choice must be compatible with the Partial Stream Selection algorithm employed in Chapter 4, as well as meeting the other issues previously outlined in the same chapter.

Following this, I explain how I incorporated the selected stream cipher (SEAL) into the prototype encryption schemes for both Video and Audio Stream encryption and how the issue of resynchronisation of the cipher is handled. I then describe my implementation and testing of this system to prove its validity. I conclude the chapter with an analysis of how the encryption scheme impacts on the provision of a Streaming Video service, especially when issues such as network problems (dropped packets, bit errors) are taken into account.

5.1 Selecting a Secure Cipher

The simple XOR Cipher presented in Chapter 4 does not provide an adequate level of security and the protected video could easily be retrieved from the encrypted stream. The final aim of the presented Cipher Scheme is to protect the content such that it becomes practically impossible to retrieve the original video, where practically impossible means that the costs involved in retrieving the video are higher than purchasing the rights to that video.

When considering a new cipher to tie together with the partial selection algorithm, there are many ciphers to choose from, including a variety of Public Key Ciphers, Private Key Block Ciphers and Private Key Stream Ciphers. There are also a number of restrictions that must be placed on these ciphers in order that they successfully interoperate with the partial selection scheme and existing Video

Streaming Servers. In this section I will begin by outlining these restrictions on the cipher before exploring and discarding the use of Public Key Ciphers and Private Key Block Ciphers. Having decided on the use of a Private Key Stream Cipher, I will then explore the suitability of Feedback Shift Register ciphers in general, as well as two popular software based Stream Ciphers – RC4 and SEAL. I will conclude with the selection of the SEAL Stream Cipher to use as a base for development of a cipher that meets the restrictions previously outlined.

5.1.1 Restrictions on the Cipher

The restrictions on the selected cipher which are outlined below come primarily from those described in Chapter 2 along with extra restrictions imposed by the design of the partial selection criteria as explained in more detail in Chapter 4. When considering all of the limitations with respect to selecting a base cipher for use in encryption, the list can be shortened to four basic requirements:

- **In-place encryption of data** – The cipher does not change the length of the plaintext.
- **Minimal CPU Load** – There must be enough CPU time to both decrypt the streaming video and decode the compressed video stream. Experimental results from Chapter 4 show that if the cipher can process a bitstream at speeds approaching 60 Mb/s on the Pentium II test platform, then real-time decryption and decoding will be possible.
- **The cipher must support resynchronisation** – This has implications for streaming of encrypted video, as the decryption module must be able to start its decryption cycle at any of the key points spreviously specified in order to support special playback modes such as indexed or high-speed playback. The cipher must be able to be restarted at will without repeating output ciphertext sequences.
- **Prevent the accidental creation of false MPEG-1 headers** – Maintaining the integrity of the MPEG-1 bit-stream as understood by Streaming Video Servers

5.1.2 Public Key Ciphers

Public Key Ciphers are generally not suitable for use when encrypting streaming video, primarily due to two reasons:

- **Public Key Ciphers are too** slow – Execution speeds do not allow for processing of the bitstream at rates required by video in real-time situations.
- **Public Key Ciphers do not allow in-place encryption** – If the data block to be encrypted is not a multiple of the cipher block size, then the data block must be extended, thereby increasing the the size of the output block.

There is however the outstanding issue of Key Management, safely delivering the Private encryption key to the client application for playback of an encrypted stream. In this scenario, Public Key Ciphers provide the basic tools to implement a safe key exchange over the public Internet. While this thesis will not discuss the Key Management issue, it does note that a Public key Cipher will likely

form an important part of an overall streaming video solution in the implementation of a decryption key delivery system.(Aslam, 1998; Denning, 1983; Menezes et al., 1997; Rivest et al., 1978; RSA, 1996; Schneier, 1996a)

5.1.3 Private Key Ciphers – Block Ciphers

Private Key Block Ciphers (Schneier, 1996a; Schneier, 1998; NIST, 1993a; Preneel et al., 1998) are also generally not suitable for the encryption of streaming video. While Block Ciphers are certainly fast enough to decrypt encrypted streaming video in real-time, and there are a large number of pre-existing secure Block Ciphers from which to choose, other properties of Block Ciphers in general means that they are not suitable for the encryption of streaming video. These properties are:

- **Output is a random bit-stream of the same length as the input block** – a 64-bit Block Cipher will randomly translate one of 2^{64} possible inputs to one of 2^{64} possible outputs. It is extremely difficult to ensure that the Block Cipher does not create any false MPEG-1 headers, such as the byte aligned 24-bit value (0x00-0x00-0x01).
- **Data is only encrypted in multiples of the block size** – meaning that if the size of the plaintext data block is not a multiple of the block size it must be extended to enable the use of Block Ciphers. This rules out an implementation that uses in-place encryption of Video and Audio Streams such as the one presented in Chapter 4.
- **Running the Block Cipher in CFB (Cipher FeedBack) or CBC (Cipher Block Chaining) mode is too slow** – running the Cipher in one of these modes can be used to reduce the effective block size to 8 bits, enabling in-place encryption of the Video and Audio Streams, in a 64-bit Block Cipher, this would reduce the speed of the cipher by a factor of eight, making the cipher too slow.

5.1.4 Private Key Ciphers – Stream Ciphers

This leaves Private Key Stream Ciphers as the final alternative, which, unlike Block Ciphers or Public Key Ciphers, seem to meet the four basic requirements as previously outlined. A Stream Cipher functions by XORing the plaintext with a pseudo-random string of the same length in order to produce the ciphertext. Traditional hardware based Stream Ciphers actually use a pseudo-random bit generator. For software implementations, it is often more convenient to consider a larger minimum block size of 8 bits. If the pseudorandom generator outputs larger blocks, they can be broken down into multiple blocks of the required minimum block length (e.g. 8 bits), ensuring that the ciphertext and plaintext length remains equal. (Schneier, 1996a)

Since the MPEG-1 partial selection scheme processes and encrypts the data stream at byte level, any cipher that encrypts data in block sizes of 8 bits will allow in-place encryption of the MPEG-1 stream. Given that all Stream Ciphers can be made to encrypt data with a block size of 8 bits, this provides a guarantee that the length of the ciphertext is exactly the same size as the plaintext, therefore ensuring that in-place encryption of the MPEG-1 System Stream can take place.

Stream Ciphers also meet the requirements regarding the speed or rate at which data can be encrypted, since the XOR operation is not time consuming, the cipher encryption rate is dependant on the speed of the random number generator. While some Stream Cipher designs are engineered towards hardware solutions, popular software based Stream Ciphers are among the quickest ciphers available. Indeed, the RC4 cipher can encrypt data at the rate of approximately 9 machine instructions per byte while the SEAL cipher can potentially require only 5 machine instructions per byte of data encrypted. (Schneier, 1996a; Rogaway and Coppersmith, 1998; RSA, 1996)

Stream Ciphers are also more amenable to re-synchronisation than other ciphers, primarily because a Stream Cipher can be considered to be a state machine where the current state determines the next output. Re-synchronising the cipher is usually a simple matter of resetting the internal state to a known value. Since the random number generator is a closed system, generating a random stream without external input, we can easily reset the internal state. Some Stream Ciphers may have a large state variable to reset while other may allow quicker re-synchronisation. (Schneier, 1996a)

The final point of concern involves securing against the production of false MPEG-1 headers, and again Stream Ciphers prove to be the most amenable to solving this potential problem. Without going into too much detail here, a similar approach to what was used in Section 4.2.2.2 protects against the creation of false headers while not overly compromising the security of the Stream Cipher. Again, this advantage is primarily due to the operation of the Stream Cipher where the pseudorandom byte stream is XORed with the plaintext to produce the ciphertext.

In the next sub-sections, I will discuss a range of Stream Ciphers, discussing their unique suitability to the task of MPEG-1 encryption. The ciphers that will be explored for suitability include the entire range of standard Feedback Shift Register Stream Ciphers, and the two most popular software based Stream Ciphers, RC4 and SEAL. Following this I will outline my reasons for choosing the SEAL cipher as a base for which to modify before inclusion into the prototype encryption scheme proposed in the previous chapter.

5.1.4.1 Feedback Shift Registers

Linear Feedback Shift Registers (LFSR) and their descendants, Feedback with Carry Shift Registers (FCSR) and Non-linear Feedback Shift Registers (NLFSR) have been used in hardware stream encryption systems by military organisations for a long time (Schneier, 1996a). Their simplicity in design, see Figure 5-1, leads to extremely simple and fast hardware implementations that can encrypt a serial bit-stream at extremely high rates – due to the generator producing a single output bit on each clock cycle. The translation of these ciphers into software implementations often lead to cumbersome and slow ciphers, primarily due to the weakness of bit-level operations on modern computers. While the potential speed of an FSR based cipher in a software implementation is compromised, it should still operate at the speeds required for encryption of streaming video.

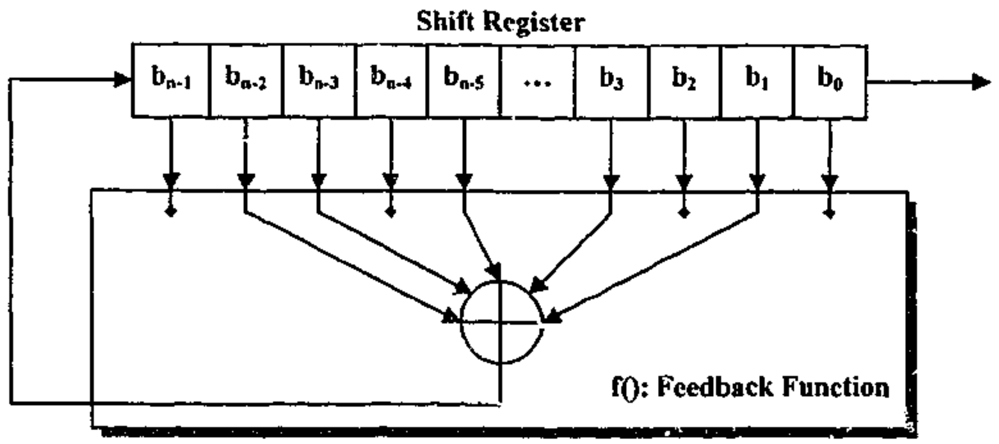


Figure 5-1: Linear Feedback Shift Register Random Stream Generator

What is of more concern however is the difficulty in designing a secure FSR based stream cipher. FSR systems have considerable mathematical theory behind them to prove their effectiveness in random number generation, however this theory also provides a good starting point when attacking the cipher for any weaknesses. Even given that most Stream Ciphers based on LFSRs are built by combining multiple LFSR systems, guaranteeing the security is difficult, and each extra LFSR slows down any software implementation. (Schneier, 1996a)

FCSR based Stream Ciphers are relatively new compared to the simpler LFSR based ciphers and little cryptanalytic work has been done in this field. However, like LFSR based systems, they are slow to implement in software and therefore unlikely to be useful for encrypting the MPEG-1 System Stream. NLFSR generators are similar to LFSR generators except that the feedback function can be more complex than a simple XOR of the tapped bits. An interesting side point of NLFSR based ciphers is that while the complexity of the non-linear feedback means that the cipher output is more difficult to analyse and break, it also means that the randomness of the cipher is more difficult to analyse. (Schneier, 1996a)

LFSR based Stream Ciphers are not suitable for use in encryption of an MPEG-1 System Stream. The unique suitability of these ciphers towards a hardware implementation, and the relative complexity of a software implementation make a compelling case for their non-use.

5.1.4.2 RC4

RC4 is a fast and simple software based Stream Cipher developed in 1987 by Ron Rivest of RSA Data Security (Schneier, 1996a). RC4 is a proprietary algorithm whose details are only made available after signing a non-disclosure agreement. In 1994 however, an algorithm claiming to be the RC4 algorithm was published online, while RSA has not confirmed that the algorithm is correct, licensed users of RC4 have confirmed that the random key sequence generated by the published algorithm for a given key is equal to that produced by RC4 for the same key. As such, the published RC4 algorithm is often referred to as Alleged RC4 (Schneier, 1996a; Fluhrer and McGrew, 2000; Golic, 1997). For future reference in this thesis, the Alleged RC4 algorithm will be referred to as RC4.

The RC4 algorithm is extremely simple to implement and is one of the primary reasons why it is such a fast cipher. It consists primarily of a slowly permutating S-Box – 256 8-bit values – which is used to select one of 256 possible output values to generate the key sequence. The current state of the RC4 cipher is determined by the contents of the S-Box and two 8-bit values, *i* and *j*. This allows for approximately 2^{1700} (= 256! × 256^2) possible states, however, there is no guarantee that the cipher will cycle through all possible states for a given secret key or for any current state. The initial state of the S-Box is based on the value of the secret key, and is initialised using the algorithm shown in Figure 5-2, while the random key sequence – which is XORed with the plaintext – is generated using the algorithm from Figure 5-3.

Key = K = $K_0K_1K_2 \ldots K_{n-1}$

```
for (i=0 to 255)
{
     SBox[i] = i;
     Init[i] = K(i mod n);
}
j = 0;
for (i=0 to 255)
{
     j = (j + SBox[i] + Init[i]) mod 256;
     Swap(SBox[i], SBox[j]);
}
i = j = 0;
```

Length of the Secret Key (K) can be any multiple of 8 bits between 8 and 2048 bits (1 and 256 bytes)

Execution Times:
- 1st Loop – 512 Machine Operations
- 2nd Loop – 2560 Machine Operations

Figure 5-2: RC4 S-Box Initialisation

There was little cryptanalytic analysis performed on RC4 prior to 1994 due to its status as a non-published algorithm. However, since its accidental release, it has been analysed by many in a search for a cryptanalytic attack. Thus far, the avenues of attack on RC4 are twofold – statistical analysis and tracking analysis -- both of which have concluded that RC4 remains a secure cipher. The statistical analysis approach involves using digraph probabilities, or the probability of two consecutive pseudo-random output bytes occurring in the generated key sequence (Fluhrer and McGrew, 2000; Golic, 1997). For versions of RC4 with smaller output words (2-5 bits), this approach involves the following steps:

- For each given state – *i*, *j* and the S-Box values, calculate the next two output words
- Given all these digraphs, determine the probabilities of each digraph.

For a truly random sequence, these probabilities would be equal, however it has been found that certain digraphs are more probable than others. This algorithm is of order 2^{8n}, where *n* is the output word size in bits. Executing this algorithm with a word size of 8 bits – the full implementation

of RC4 – is both memory intensive and too slow. For larger sizes of *n*, RC4 is used with random keys to produce a long output sequence, digraph probabilities are then calculated on this output sequence. When *n* is equal to 8, it has been found that $2^{30.6}$ output words are required to distinguish the output sequence from a purely random distribution. The statistical analysis approach has been used to show that the RC4 output sequence is not purely random, however no attack has been formulated to use this knowledge so that the pseudo-random stream can be reproduced.(Fluhrer and McGrew, 2000; Golic, 1997; Knudsen et al., 1999)

```
i = (i + 1) mod 256;
j = (j + SBox[i]) mod 256;
Swap(SBox[i], SBox[j]);
return SBox[(SBox[i] + SBox[j]) mod 256];
```

Execution Time: Between 8 and 16 Machine Operations

Figure 5-3: RC4 Pseudo-Random Sequence Generation

The second attack of tracking analysis (Mister and Tavares, 1998a; Mister and Tavares, 1998b) involves knowledge of some portion of the output sequence – obtained by XORing the ciphertext with some known plaintext – and then attempting to reconstruct the values of the S-Box, therefore obtaining the current state of the RC4 cipher. Once the current cipher state is known, it can be used to generate more output bits. This approach uses a recursive algorithm to try all possible values of S-Box entries until a contradiction in assignment occurs, the algorithm then backtracks and tries a different value for the S-Box. The algorithm involves assigning all possible values of the S-Box and therefore is of order $2^n!$, as well as increased memory requirements to store the S-Box state at each level of recursion. While supplying some initial values of the S-Box can decrease the execution time when *n* is equal to 8, about half of the S-Box values are required to reproduce the current S-Box state and a cryptanalyst is unlikely to have this information.

Tracking analysis provides an interesting method of attack on RC4. As computing power increases, this approach can eventually be used to break RC4 encrypted material. However, all current evidence indicates that RC4 will be secure for some time to come. Even if RC4 does become insecure, its potential application in streaming video protection would involve frequent (approximately twice per second) resynchronisation of the cipher. This means that the same attack would have to be repeated twice for each second of encrypted video. As such, this attack would be rendered impractical due to the requirements of the streaming video cipher.

Based on current computing capability, RC4 provides a secure level of protection against attack, however, it is important to see how well RC4 meets the four requirements previously outlined. In the first two instances, RC4 has no problem, like all Stream Ciphers it can perform in-place encryption of the MPEG-1 Stream due to its ability to process plaintext in block sizes of 8-bits. Also, the simplicity of the key sequence generation algorithm means that each pseudo-random byte can be calculated using between 8-16 machine operations, ensuring that RC4 is able to encrypt data at a high rate. The third issue, resynchronisation of the cipher, is also solvable – this involves resetting the current state of the RC4 cipher and can be done by resetting the S-Box, *i* and *j* to known values using a different key. Finally, a similar approach as was taken in the previous chapter could be used to modify the XOR operation of the RC4 cipher to guard against the accidental creation of false MPEG-1 headers.

Given that all the requirements are met, RC4 seems to be a potential candidate for use in the MPEG-1 encryption scheme. A key issue to overcome however is the complexity in re-synchronisation of the RC4 cipher since this involves the generation of a new key for each re-synchronisation point and the subsequent regeneration of the S-Box. While the processing load is not a key problem given computing power currently available, this extra complexity could potentially cause some problems, especially in older and slower machines.

5.1.4.3 SEAL

SEAL is a very fast software based cipher developed in 1993 by Phil Rogaway and Don Coppersmith of IBM (Rogaway and Coppersmith, 1993). The design has been optimised for implementation on 32 bit processors. While a license is required for commercial usage of the SEAL algorithm, the algorithm has been openly published for peer review (Schneier, 1996a). SEAL differs from most Stream Ciphers in that it is a member of the pseudo-random function family – after a lengthy set-up procedure using a 160 bit secret key, SEAL can be used to produce one of 2^{32} different pseudo-random outputs, which are XORed with the plaintext to produce the ciphertext. SEAL does not strictly conform to the description of Stream Ciphers since it outputs a random stream of 32 bit values, however, each random 32 bit value can be treated as four random 8 bit values or as 32 independent random bits if desired.

The SEAL Cipher can be broken up into two stages, the first being a setup stage whereby the 160 bit secret key is provided as input and approximately 3kB of lookup tables are calculated and stored. This is a time consuming process that ideally should only occur once. The second stage of the algorithm is the generation of a random string. The pre-calculated tables for SEAL can be used to produce up to 2^{32} individual random strings of 32 bit words. The SEAL function is called with a 32 bit sequence number (*n*) which is used to choose one of the random sequences. The length of the random sequence is dependant on the size of the third table used by SEAL. (Rogaway and Coppersmith, 1993; Rogaway and Coppersmith, 1998)

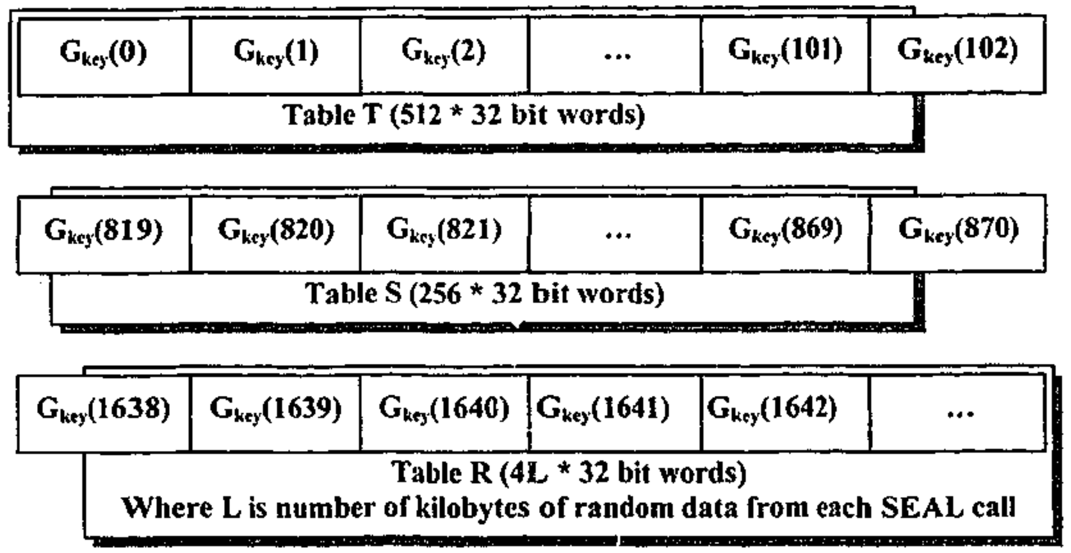
While the SEAL specifications call for the algorithm to return the entire random string, the algorithm can be broken down into individual rounds where four 32 bit values are calculated at each round. If a finer granularity than 128 bits is required, some extra code is required to separate this 128 bit value into smaller values. As for all Stream Ciphers, the random string provided by the SEAL algorithm is be XORed with the plaintext to produce the ciphertext. A feature provided by SEAL is the ability to resynchronise the cipher by selecting a different random sequence through selection of a different sequence number (n).

```
input:
     key:                                          /* 160 bit secret key     */
     n:                                            /*  32 bit selector       */

output:
     result:                                       /* 160 bit random hash    */

variables:
     h0, h1, h2, h3, h4                            /* 32 bit variables       */
     X[80]                                         /* 80 * 32 bit array      */
     A, B, C, D, E, temp                           /* 32 bit variables       */

functions:
     fun$_0$(X, Y, Z): (X ∩ Y) ∪ (¬(X) ∩ Z)
     fun$_1$(X, Y, Z): X ⊕ Y ⊕ Z
     fun$_2$(X, Y, Z): (X ∩ Y) ∪ (X ∩ Z) ∪ (Y ∩ Z)
     fun$_3$(X, Y, Z): fun1(X, Y, Z)

constants:
     C$_0$ = 0x5a827999
     C$_1$ = 0x6ed9eba1
     C$_2$ = 0x8f1bbcdc
     C$_3$ = 0xca62c1d6

algorithm:
     A = h0 = key$_{(0..31)}$; B = h1 = key$_{[32..63]}$; C = h2 = key$_{[64..95]}$;
     D = h3 = key$_{[96..127]}$; E = h4 = key$_{[128..159]}$;

     X[0] = n;
     for (j = 1⇒79)
     {
          case j
          {
               1⇒15:    X[j] = 0;
               16⇒79:  X[j] =
                         (X[j-3] ⊕ X[j-8] ⊕ X[j-14] ⊕ X[j-16])$\circlearrowleft_{31}$;
          } end case
     }

     for(j = 0⇒79)
     {
          temp = A$\circlearrowleft_{27}$ + fun$_{j/20}$(B, C, D) + E + X[j] + C$_{j/20}$;
          E = D; D = C; C = B$\circlearrowleft_{2}$; B = A; A = temp;
     }

     result$_{(0..31)}$ = (h0 + A); result$_{[32..63]}$ = (h1 + B);
     result$_{[64..95]}$ = (h2 + C); result$_{[96..127]}$ = (h3 + D);
     result$_{[128..159]}$ = (h4 + E);
```

Figure 5-4: SEAL G_{key}(n) function

The table setup phase of the SEAL Cipher utilises a function called **G_{key}(n)**, the implementation of this function is described in Figure 5-4. This function is based on the standardised SHA-1 160 bit Hash function (NIST, 1993b) which is designed to produce a 160 bit representation of a longer document. This algorithm is modified to produce the hash value of a 32 bit variable which is extended to the required 512 bits with the addition of zeroes. The other modification concerns the initial values of the temporary variables – breaking up the 160 bit key into five 32 bit values. In effect, **G_{key}(n)** maps a 32 bit value to a 160 bit value via an irreversible process where the procedure is based on the secret key. For more information on the functionality of the SHA-1 algorithms, see (NIST, 1993b; Schneier, 1996a).

The tables used by SEAL are labelled **T**, **S** and **R**. Table **T** is a table of 512 32-bit words and is used by SEAL as a 9 bit * 32 bit S-Box. The entries of the S-Box are set to be the first 102 outputs of the **G_{key}(n)** function. Only the first 64 bits of **G_{key}(102)** are required to fill up the entirety of the table. Table **S** is a table of 256 32-bit words and is used by SEAL as values to either ADD or XOR to internal loop variables to produce a final output. Each entry in the **S** table is used only once through all iterations of the inner loop. As table **S** requires 1kB of storage to hold the table, all iterations of the SEAL inner loop produce exactly 1kB of random data. The **S** table is also populated with consecutive **G_{key}(n)** calculations, starting with **G_{key}(819)** and offset 32 bits into this value. Table **R** is sized depending on how long a random string is required from the SEAL algorithm. Each execution of the outer loop of SEAL provides 1kB of random data and this loop is executed **L** times to produce **L**kB of random data. Table **R** contains (4 * **L**) 32 bit words that are used to generate starting values for internal variables at the start of each outer loop. As for the other tables, we use consecutive **G_{key}(n)** values starting from the 65th bit of **G_{key}(1638)** through to however many calculations are required to fill the table. The table sizes and entries can be seen in Figure 5-5.



G_{key}(0)	G_{key}(1)	G_{key}(2)	...	G_{key}(101)	G_{key}(102)

Table T (512 * 32 bit words)

G_{key}(819)	G_{key}(820)	G_{key}(821)	...	G_{key}(869)	G_{key}(870)

Table S (256 * 32 bit words)

G_{key}(1638)	G_{key}(1639)	G_{key}(1640)	G_{key}(1641)	G_{key}(1642)	...

Table R (4L * 32 bit words)
Where L is number of kilobytes of random data from each SEAL call

Figure 5-5: SEAL Table Generation

Finally we come to the SEAL algorithm to generate a random string, which can really be seen as a function that maps a 32 bit value into an **L**kB random string, as shown in Figure 5-6. For each pass though the outer loop, we reinitialise our internal loop variables by combining the 32 bit sequence selector **n** with the next four entries in the **R** table using XOR. We then go through a short initialisati n phase where a sequence of instructions update the **A**, **B**, **C** and **D** variables, these instru ..ons are executed three times with the values copied into the **n1**, **n2**, **n3** and **n4** variables after the second loop. At this stage we are ready to execute the inner loop.

```
input:
     n:                                        /*  32 bit selector        */
output:
     randstring:                               /* L kB random string      */
variables:
     loop, count                               /* Loop variables          */
     A, B, C, D, n1, n2, n3, n4                /* 32 bit variables        */
functions:
     setadd(X, Y, Z): X = (Y ∩ 0x7fc); Z = Z + T[X↻2]; Y↻9;
     setxor(X, Y, Z): X = (Y ∩ 0x7fc); Z = Z ⊕ T[X↻2]; Y↻9;
     updadd(X, Y, Z): X = ((X + Y) ∩ 0x7fc); Z = Z + T[X↻2]; Y↻9;
     updxor(X, Y, Z): X = ((X + Y) ∩ 0x7fc); Z = Z ⊕ T[X↻2]; Y↻9;
algorithm:
     for (loop = 0⇒(L - 1))
     {
          A = n ⊕ R[4 * loop];
          B = (n↻8) ⊕ R[(4 * loop) + 1];
          C = (n↻16) ⊕ R[(4 * loop) + 2];
          D = (n↻24) ⊕ R[(4 * loop) + 3];
          for (count = 0⇒2)
          {
               setadd(pos1, A, B);
               setadd(pos1, B, C);
               setadd(pos1, C, D);
               setadd(pos1, D, A);
               if (count = 1)
                    n1 = D; n2 = B; n3 = A; n4 = C;
          }
          for (count = 0⇒63)
          {
               setadd(pos1, A, B); B = B ⊕ A;
               setxor(pos2, B, C); C = C + B;
               updadd(pos1, C, D); D = D ⊕ C;
               updxor(pos2, D, A); A = A + D;
               updxor(pos1, A, B);
               updadd(pos2, B, C);
               updxor(pos1, C, D);
               updadd(pos2, D, A);
               append(randstring, B + S[4 * count]);
               append(randstring, C ⊕ S[(4 * count) + 1];
               append(randstring, D + S[(4 * count) + 2];
               append(randstring, A ⊕ S[(4 * count) + 3];
               if (odd(count))
                    A = A + n1; B = B + n2; C = C ⊕ n1; D = D ⊕ n2;
               else
                    A = A + n3; B = B + n4; C = C ⊕ n3; D = D ⊕ n4;
          }
     }
```

Figure 5-6: SEAL Random String Generation

The inner loop of the SEAL algorithm generates 1kB of random data. For each cycle through the loop, it executes a sequence of eight functions to update the internal variables. These functions use 9 bits from one variable to select a value from table **T**, which is then either ADDed or XORed with a different variable. The eight functions perform this operation on different combinations of the internal variables. Finally, the four internal variables are then combined with the next four values from the **S** table using either ADD or XOR to produce the next 128 bits of the random string.

Before recommencing the inner loop, all four variables are modified by either adding or xoring them to either **n1** and **n2**, or **n3** and **n4**, depending on whether the loop counter is odd or even. Obviously, both the inner and outer loop counters could be maintained as states to produce an implementation that returns a 128 bit value rather than an Lkb string.

The SEAL Cipher claims its security from a number of different angles:

- It uses a large, secret S-Box that is key dependant.
- Using the SHA-1 Hash function ensures that the entries to all tables are effectively random.
- A random value from table **R** is used to initialise the internal variables and a random value is used from table S to modify the final values before output.
- The inner loop interchanges the ADD and XOR operations, which are non-commutative.
- The internal variables are updated differently by a random variable at the end of the inner loop based on the least significant bit of the loop counter.

The security offered by the SEAL Stream Cipher has thus far been strong, with no successful attacks on the algorithm in the nine years since its publication. Indeed, the only weakness found with the cipher has been through the use of the Chi-Squared attack (Handschuh and Gilbert, 1997). This attack involves the use of the χ^2 statistic, which can be used to test the randomness of a sequence of numbers. The implementation of the attack uses the fact that there is some correlation between certain internal variables of the SEAL algorithm and focuses on the four least significant bits of every fourth output word of SEAL. The overall result of this attack on SEAL is that given approximately 2^{30} consecutive output words from SEAL, it is possible to determine that the sequence is not a random sequence. This result however, does not provide the original secret key, details of the values of the **R**, **S** and **T** tables, nor the beginnings of an approach to determine these values, and therefore not providing a significant threat to the SEAL cipher.

A similar attack based on the same correlation has been made on a modified version of SEAL where the summation operations are replaced by XOR operations. This attack has been used to yield portions of the **T** table. It is important to note however that the cipher was designed with both summation and XOR operations specifically because these operations do not commute – it is therefore unlikely that a similar attack could be used on the full version of SEAL.(Handschuh and Gilbert, 1997)

In response to the issues observed using the χ^2 statistic, Rogers and Coppersmith released a modification to SEAL in 1998 (Rogaway and Coppersmith, 1998) named SEAL 3.0 – this is the version presented in this thesis. In this paper, they explained that SEAL 3.0 was their original design which they later modified to increase the speed of implementation. Cryptanalysis on the SEAL 3.0 algorithm shows that the χ^2 statistic no longer reveals any potential weaknesses and no attack on SEAL 3.0 has been discovered. SEAL has thus far proven to be a very secure Stream Cipher, however it is a relatively new cipher and some attacks may be discovered in the future. Even so, its current record

suggests that security is not a problem and its ability to output one of many pseudo-ransom values lend to an efficient resynchronisation ability.

As regards to the four requirements previously outlined, SEAL either directly meets or can easily be modified to meet them. When considering in-place encryption of data, this can be performed with a simple modification, SEAL XORs a 32-bit pseudo-random word with the plaintext at each stage, this 32-bit word can be broken up into four 8-bit pseudo-ransom words – therefore enabling in-place encryption of data. It is important to specify however, whether the 32-bit word will be broken up using little-endian or big-endian byte order, so that the same approach will work on both types of hardware platforms. The second requirement on the speed of the cipher is also easily met with SEAL being one of the fastest software based ciphers with a minimum of 5 machine operations required to encrypt each byte of plaintext.

A minor problem with the SEAL cipher is that the maximum length of the pseudo-random output for a given value of n is 64kB (Rogaway and Coppersmith, 1998), therefore the cipher must be resynchronised before more than 64kB of data is encrypted. A video sequence encoded at 2Mb/s requires 256kB for 1 second of video, a similarly encoded sequence at 1.5Mb/s requires 192kB while a 2.5Mb/s video requires 320kB. However, the encoding rate is not constant across individual frames and is dependent on the encoding used for each frame (I, P or B-Frame). Since the size of the I-Frames within an encoded video sequence can often approach 64kB, it becomes necessary for resynchronisation to occur before and after each I-Frame. While resynchronisation is only required at each GOP header to ensure that all the different playback modes are supported, the small length of the maximum pseudo-random output of SEAL requires that resynchronisation occur for each frame of video rather than for each Group-Of-Pictures.

It should be noted that the length of this pseudo-random string is determined by the size of the **R**-Table used by the cipher. It appears that it would be possible to further increase the size of this table to allow generation of longer pseudo-random strings. Indeed the algorithm used to fill the **R**-Tables does not restrict continuous calculation of values for a larger table. There has been no published work analysing the effect of increasing the **R**-Table size and therefore the pseudo-random string output length but it appears on initial examination to not affect the security of the cipher. The $\mathbf{G_{key}}$ function fills the tables with random values and each entry in the **R**-Table is used as part of a formula to generate part of the pseudo-random string. Since the $\mathbf{G_{key}}$ function is based on the thoroughly tested SHA-1 Hash Algorithm, increasing the **R**-Table length would appear to only increase the potential length of output without impacting on the security of the cipher as a whole. However, confirming this hypothesis would take serious examination and is outside the scope of this thesis.

The SEAL cipher can also easily be resynchronised, but this procedure differs from the approach that would be used with a different cipher such as RC4. As SEAL allows us to produce one of 2^{32} different pseudo-random bit streams, resynchronisation is a simple matter of selecting a different random stream from the one currently being used. This operation would be performed at each resynchronisation point, even allowing for 30 resynchronisation points per second of encoded video, this allows over 39,000 hours of video to be encrypted using the same key without re-using a pseudo-random sequence. Due to the implementation of SEAL, this procedure involves little or no processing power and is performed by setting the value of *n* to the newly selected random stream and restarting the SEAL generation algorithm from Figure 5-6. Finally, like RC4, SEAL is amenable to modification to ensure against the production of false MPEG-1 headers.

SEAL meets all of the requirements for selection as a potential base cipher in the implementation of a secure MPEG-1 cipher.

5.1.5 Conclusion

In conclusion, I recommend using the SEAL Stream Cipher as the base system for use in encryption of an MPEG-1 System Stream. The primary reasons for this choice are as follows:

- **Security** – following a great deal of work, the SEAL cipher has been shown to be secure such that a brute-force attack is the only viable attack on a SEAL encrypted stream. Despite mathematical tests being able to prove that the random stream generated by the original SEAL algorithm is not purely random, this approach has failed to produce an avenue of attack against SEAL. Further, the SEAL 3.0 algorithm has proven to be resistant against the same analysis.

- **Speed** – as SEAL is highly optimised for software implementation, it can generate pseudo-random values at a high rate, potentially requiring only 5 machine instructions per random byte. This ensures that most machines will be capable of performing both decryption and decoding of an MPEG-1 stream in real-time.

- **Re-synchronisation** – as a random number generator SEAL operates using two keys, one is used to calculate the initial table entries and is considered to be the key. The second key is used to select one of 2^{32} possible output random streams. This allows for simple re-synchronisation by selecting a different output stream at each synchronisation point, a range of 2^{32} points is many more than is required for even a 2-hour video which would only require approximately 216,000 or 2^{17} synchronisation points.

- **Ease of modification** – SEAL is easily amenable to modification such that it can encrypt plaintext in block size of 8-bits, ensuring the in-place encryption of the MPEG-1 System Stream can occur. Also in common with all Stream Ciphers, because the random stream is XORed with the plaintext, it is possible to ensure that false MPEG-1 Headers are not created.

5.2 MPEG-1 Video Stream Encryption

In this section I will examine the issues involved with integrating the SEAL cipher into the MPEG-1 Video Stream partial selection scheme for encryption that I developed in Chapter 4. Previously, I briefly discussed that SEAL could be modified to ensure against the creation of false MPEG-1 headers, as well as show that SEAL was open to easy cipher resynchronisation due to its

inherent nature, there is still the issue as to how these features should be exploited when used with the MPEG-1 Video Stream partial selection algorithm. Issues in particular that must be explored include:

- Modification of SEAL for 8-bit keystream generation rather than 32-bit.

- How to modify SEAL such that false MPEG-1 headers are not created, and how does this modification affect the security of the SEAL cipher.

- How to key the SEAL resynchronisation sequence such that indexed and high-speed playback remain supported at the client end.

I will begin by explaining how SEAL should be modified so that it can be incorporated within the existing prototype encryption system. This includes modifying the SEAL output so that the keystream is generated in block sizes of 8 bits as well as ensuring that false MPEG-1 headers are not created. Finally I will look at the issue of resynchronisation in more detail, explaining how I plan to key the resynchronisation of the cipher to ensure that the special playback modes are supported.

5.2.1 Incorporation of SEAL Stream Cipher into Existing System

As previously described, the SEAL cipher can be thought of as a random number generator that outputs one of 2^{32} pseudo-random sequences based on the 160-bit secret key, where each random sequence is made up of a series of 32-bit values (Rogaway and Coppersmith, 1998; Schneier, 1996a). However, one of the restrictions on the cipher as laid out in Chapter 4 require that the ciphertext length be equal to the plaintext length, and that the cipher be able to process input in block sizes of 8-bits. On the first count, SEAL has no problem, being a Stream Cipher – the ciphertext length is always equal to the plaintext length. There is however a slight problem on the second count, as the standard implementation of SEAL processes data in 32-bit blocks. Therefore, SEAL must be modified such that it can process data in 8-bit blocks.

Looking more closely at how the cipher functions, note that SEAL produces a series of psuedo-random 32-bit values, each of which is XORed with 32 bits of plaintext to produce the ciphertext. If SEAL were modified to break the 32-bit pseudo-random value into four 8-bit values then these four separate values could be sequentially XORed with four consecutive 8-bit blocks of plaintext to produce the same ciphertext sequence. The primary issue is to define how the 32-bit values will be separated to form the four 8-bit values, since the effect will be different on Little and Big Endian machines. I have chosen to adopt the Little Endian format – compatible for Intel implementations, where a 32-bit pseudo-random value will be divided into four consecutive 8-bit pseudo-random values with the least significant byte first.

This modification allows SEAL to process plaintext data in 8 bit blocks. Every fourth call to the modified SEAL pseudo-random number generator will cause a single call to the 32-bit SEAL generator – the returned value will be divided and used for this and the subsequent three calls to the 8-bit SEAL generator. This modification does not effect the security of the SEAL cipher in any way as the same sequence of pseudo-random bits are being utilised in the XOR process.

The second issue that must be examined is to prevent the creation of false MPEG-1 headers. MPEG-1 headers can be defined as a 32-bit byte aligned sequence beginning with the 24-bit value (0x00-0x00-0x01). Because all Stream Ciphers use an XOR operation as the final step in order to produce the ciphertext, it becomes trivial to modify the cipher such that it doesn't produce this 24-bit output sequence. Indeed the simple cipher used in the prototype system and presented in Figure 4-7 can be considered to be a very simple Stream Cipher – where the psuedo-random sequence is a repeated 8-bit sequence – which has been modified to ensure that the output sequence (0x00-0x00-0x01) is never produced unless it is already in the plaintext.

The same approach can be taken when modifying the SEAL cipher, which (now) consists of an 8-bit pseudo-random number generator and an XOR function. The pseudo-random number generator is left untouched, however the generated pseudo-random byte and the plaintext byte are inspected before performing the XOR operation. As for the cipher in Figure 4-7, if the seven most significant bits of the plaintext byte are either 0 or equal to the seven most significant bits of the random byte, then the XOR operation is not performed. This cipher is shown in Figure 5-7 where k_i represents the 8-bit output of the SEAL pseudo-random generator and x_i represents the plaintext input to be encrypted.

$$
\begin{array}{lll}
\mathrm{f}(x_i,k_i): & \text{byte} & \Rightarrow \text{byte} \\
 & 0\text{x}00 & \rightarrow 0\text{x}00 \\
 & 0\text{x}01 & \rightarrow 0\text{x}01 \\
 & k_i & \rightarrow k_i \\
 & k_i \oplus 0\text{x}01 & \rightarrow k_i \oplus 0\text{x}01 \\
 & x_i & \rightarrow x_i \oplus k_i
\end{array}
$$

Figure 5-7: Modification of SEAL XOR Function for use in Video Stream Encryption

5.2.1.1 Security Provided by the Modified SEAL Cipher

The question that must now be examined is whether the modification that ensures against the generation of false MPEG-1 headers reduces the security provided by the SEAL cipher. Before looking at this, it is worth considering the results of recent cryptanalytic work on SEAL, remembering particularly that there are no known weaknesses or potential attacks on the SEAL 3.0 algorithm. We can also consider the cipher from Figure 5-7 differently by saying that if any of the four conditions for not performing the XOR are met, then the pseudo-random byte is replaced with the value 0x00 and the XOR is still performed. This increases the probability of the value 0x00 in the pseudo-random sequence. Obviously a lower number of key sequence values are now required to prove that the sequence is not random – but does this make the cipher less secure?

Given that plaintext values of 0x00 and 0x01 remain unchanged, an ideal cipher would map one of the remaining 254 plaintext values to a pseudo-random ciphertext value between 0x02 and 0xff with a probability of 0.00394=$^1/_{254}$ (5.1).

$$
\Pr\left(c_i = \alpha \middle| x_i \notin \{0x00, 0x01\} \middle| \alpha \in \{0x02 - 0xff\}\right) = \frac{1}{254} \tag{5.1}
$$

When the plaintext byte is in the range (0x02-0xff), there are three conditions where the output ciphertext byte of the cipher presented in Figure 5-7 remains unchanged – when the pseudo-random byte is equal to one of 0x00, the plaintext byte (x_i) or the plaintext byte x_i XORed with 0x01. Given that the value of the pseudo-random byte is equi-probable with probability $^1/_{256}$, this means that the value of the ciphertext byte c_i will be equal to the value of the plaintext byte x_i with probability 0.01172=$^3/_{256}$(5.2) and will be equal to one of the remaining 253 ciphertext values with probability 0.00391=$^1/_{256}$(5.3).

$$
\begin{aligned}
\Pr(c_i = x_i) &= \Pr(k_i = 0) + \Pr(k_i = x_i) + \Pr(k_i = x_i \wedge 0x01) \\
&= \frac{1}{256} + \frac{1}{256} + \frac{1}{256} = \frac{3}{256}
\end{aligned} \tag{5.2}
$$

$$
\Pr(c_i = \alpha | \alpha \neq x_i) = \Pr(k_i) = \frac{1}{256} \tag{5.3}
$$

The probability that the ciphertext output byte remains unchanged is higher than the ideal value ($^1/_{254}$). Even so this probability still remains less than 2% and is less than three times the value of the ideal probability. Assuming that all plaintext byte values are equally likely, all ciphertext byte values remain equally likely with a probability of 0.00391=$^1/_{256}$ (5.4).

$$
\begin{aligned}
\Pr(c_i = \alpha) &= \sum_{\beta=0x02}^{0xff}\left(\Pr(c_i = \alpha | x_i = \beta) \cdot \Pr(x_i = \beta)\right) \\
&= \frac{1}{256} \cdot \left(\sum_{\beta=0x02}^{0xff} \Pr(c_i = \alpha | x_i = \beta)\right) \\
&= \frac{1}{256} \cdot \left(\left(\sum_{\beta=0x02}^{\alpha-1} \Pr(c_i = \alpha | \alpha \neq \beta)\right) + \Pr(c_i = \beta) + \left(\sum_{\beta=\alpha+1}^{0xff} \Pr(c_i = \alpha | \alpha \neq \beta)\right)\right) \\
&= \frac{1}{256} \cdot \left(\left(\sum_{\beta=0x02}^{\alpha-1} \frac{1}{256}\right) + \frac{3}{256} + \left(\sum_{\beta=\alpha+1}^{0xff} \frac{1}{256}\right)\right) \\
&= \frac{1}{256} \cdot \frac{1}{256} \cdot \left(\left(\sum_{\beta=2}^{\alpha-1} 1\right) + 3 + \left(\sum_{\beta=\alpha+1}^{255} 1\right)\right) \\
&= \frac{1}{256} \cdot \frac{1}{256} \cdot ((\alpha - 2) + 3 + (255 - \alpha)) \\
&= \frac{1}{256}
\end{aligned} \tag{5.4}
$$

Similarly, the probability that a ciphertext byte value is equal to 0x00 or 0x01 is equal to the probability that these values are present in the plaintext sequence. Again given that all plaintext byte values are equally likely, this probability is also equal to 0.00391=$^1/_{256}$ (5.5) and (5.6).

$$\Pr(c_i = 0x00) = \Pr(x_i = 0x00) = \frac{1}{256} \tag{5.5}$$

$$\Pr(c_i = 0x01) = \Pr(x_i = 0x01) = \frac{1}{256} \tag{5.6}$$

The modified SEAL cipher is no less secure overall, if a cryptanalyst uncovers a 0x00 or 0x01 value in the encrypted bitstream, he/she can be certain that the plaintext value of this byte remains 0x00 or 0x01 respectively. Similarly, if a cryptanalyst uncovers another byte value in the encrypted bitstream, he/she can only be certain that the plaintext value of this byte is not 0x00 or 0x01 and that the probability that the corresponding plaintext byte is unchanged is three times more likely (the generated pseudo-random value was actually 0x00, was equal to the original plaintext byte x_i, or was equal to x_i XOR 0x01) than for any other value (5.7).

$$\Pr(x_i = c_i) = \Pr(c_i = x_i) = \frac{3}{256} \tag{5.7}$$

The difference in the probable value of a single plaintext byte is not entirely relevant. In order to decode the entire plaintext sequence, we require the original pseudo-random sequence. When attempting to determine this sequence, the triple probability for the plaintext byte being equal to the ciphertext byte devolves into three possibilities for the corresponding byte value in the pseudo-random sequence – 0x00, x_i or x_i XOR 0x01 – each with a probability of $^1/_3$, thereby decreasing the probability of individual random values back to $^1/_{256}$ (5.8). Even when the ciphertext byte is equal to 0x00 or 0x01 and therefore the plaintext byte is fully known, the pseudo-random byte value for this point in the sequence could have any value with equal probability of $^1/_{256}$ (5.9) and (5.10)

$$\text{Given } (x_i = c_i): \quad \begin{aligned} \Pr(k_i = 0x00) &= \frac{1}{3}\cdot\frac{3}{256} = \frac{1}{256} \\ \Pr(k_i = x_i) &= \frac{1}{3}\cdot\frac{3}{256} = \frac{1}{256} \\ \Pr(k_i = (x_i \wedge 0x01)) &= \frac{1}{3}\cdot\frac{3}{256} = \frac{1}{256} \end{aligned} \tag{5.8}$$

$$\text{Given } (c_i = 0x00): \quad \Pr(k_i) = \frac{1}{256} \tag{5.9}$$

$$\text{Given } (c_i = 0x01): \quad \Pr(k_i) = \frac{1}{256} \tag{5.10}$$

In conclusion, the modifications to the cipher still ensure that the individual values of the pseudo-random sequence remain equally likely for any given ciphertext sequence. If an attempt were made to retrieve the plaintext through breaking the SEAL cipher, the cryptanalyst must first retrieve a part of the pseudo-random sequence using some known plaintext. Where a ciphertext was 0x00 or 0x01, the pseudo-random byte is completely unknown – since the value of k_i is random and equi-probable. Where the ciphertext and plaintext byte values are equal, the pseudo-random byte value can be one of three possible values (0x00, x_i and x_i XOR 0x01) with equal likelihood. This actually increases the difficulty in determining the actual pseudo-random stream and therefore mounting an attack on SEAL itself. Even though the modified Stream Cipher has an increased probability of outputting the 0x00 byte as input to the XOR function, this does not degrade the difficulty in determining the original SEAL pseudo-random sequence. While the probability that an individual byte is unchanged is slightly greater than desired, over a sequence of consecutive bytes the effect of this statistical abnormality is reduced.

5.2.2 Resynchronisation of Cipher at each Picture

It is important to be able to resynchronise the cipher at the start of each Group-Of-Pictures (GOP), allowing us to provide indexed and high-speed playback modes. Indexed playback is performed by streaming the MPEG-1 System Stream from the start of the closest GOP. Since the first frame in a GOP is always an I-Frame, and other frames in a GOP rely on this I-Frame being decoded first, it is impossible to commence playback in the middle of a GOP as some frames cannot be reconstructed. Indexed playback always commences at the start of a GOP, implying that the cipher must be able to be resynchronised at the start of each GOP within the MPEG-1 Video Stream. During indexed playback, the first frame will be correctly decrypted as the cipher will be correctly resynchronised by the GOP header immediately preceding the frame to be decoded, subsequent frames will be decrypted as for normal playback. (Anderson, 1996; Lin et al., 2001; Jayanta et al., 1994)

High-speed playback is performed by streaming a modified MPEG-1 Video Stream – containing only I-Frames – across the network. The modified Video Stream is constructed by deleting all but the first I-Frame Picture from each GOP in the original Video Stream. Sending the GOPs in reverse order allows playback in fast-rewind mode, this is properly decoded and displayed as each GOP only contains one frame. As for indexed playback, as long as the cipher is resynchronised at the start of each GOP, we will be able to successfully decrypt the stream in the high-speed playback modes. The high-speed playback modes can be considered to be a form of indexed playback where each jump involves playback of a single frame only, if indexed playback is properly supported, then so is high-speed playback with the cipher being resynchronised at the start of each frame. (Shanableh and Ghanbari, 2001; Leditschke and Johnson, 1995; Frimout et al., 1995)

While it is only essential to resynchronise the cipher at each GOP header, because the SEAL cipher can only produce a 64kB pseudo-random sequence (Rogaway and Coppersmith, 1998) for each value of n (the length of an I-Frame can easily approach 64kB), we must instead resynchronise the cipher at each Picture Header. The SEAL cipher can easily be resynchronised by selecting a different pseudo-random stream – by changing the value of n – but the encrypted video stream must also contain information as to which of the 2^{32} pseudo-random streams to select. This cannot be encoded into the Video Stream as extra information as in-place encryption of the MPEG-1 Video Stream will no longer be possible. The information as to which value of n to use must be extracted from existing information contained within the GOP and Picture Headers.

Following the GOP Header marker – the 32-bit byte aligned sequence (0x00 0x00 0x01 0xb8) – is a 25-bit field that contains a time-stamp value for the GOP being decoded. This value is unique within the MPEG-1 Video Stream and can be used as an input to calculate the value of *n* to resynchronise the SEAL cipher – every time a GOP header is processed, this value is updated for future resynchronisation of the SEAL cipher while processing each Picture header.

Following the Picture Header Marker – the 32-bit byte aligned sequence (0x00 0x00 0x01 0x00) – is a 10-bit field containing the picture order counter for the GOP. This value is unique within the outlying GOP and can be used as a secondary input to calculate the value of *n* to resynchronise the SEAL cipher. Every time a Picture header is processed, the picture count value and the outlying GOP time-stamp is used to resynchronise the SEAL cipher to decrypt the frame of video represented by this Picture.

The question arises as how to combine the 25-bit time-stamp value with the 10-bit picture count value to form a single unique 32-bit value – the value of *n* must be unique to avoid re-use of the same pseudo-random sequence. The format of the time-stamp is shown in Figure 5-8, note that five bits have been allocated to encode the value of hours, allowing representation of up to 31 hours of video playback. Also note that the first bit of the time-stamp is a drop-frame marker bit, indicating whether the video is encoded at 29.97 frames per second or not. I propose to reduce the time-stamp value to a length of 22-bits by ignoring the drop-frame marker and the two most significant bits of the hours field, dropping these fields still provides a unique 22-bit value as long as the video length does not exceed 8 hours – a good assumption for almost all video sequences. The 10-bit picture count field can then be appended to the end of the 22-bit modified time-stamp value to form a unique 32-bit value for each frame of video as long as the video sequence is shorter than 8 hours.

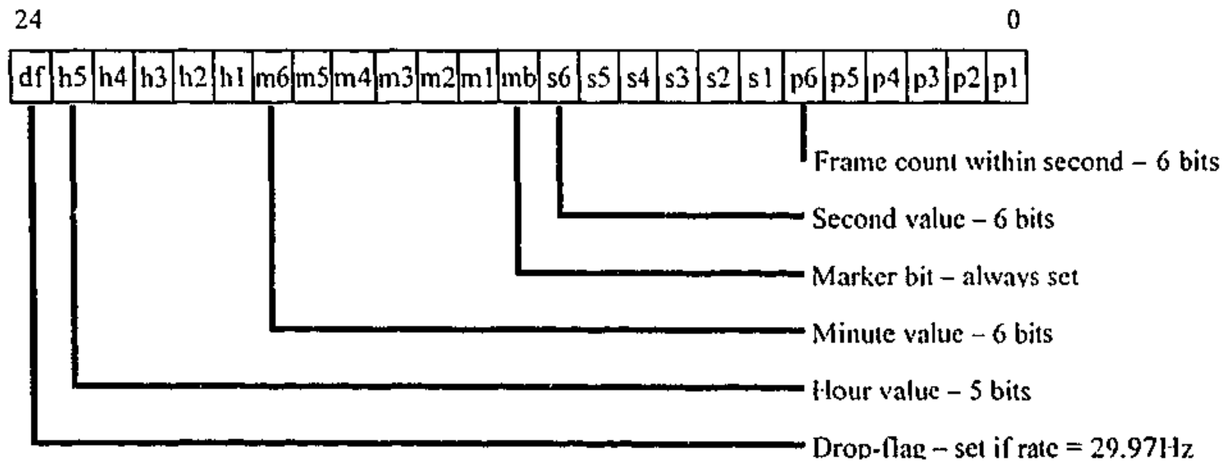

24
0
df h5 h4 h3 h2 h1 m6 m5 m4 m3 m2 m1 mb s6 s5 s4 s3 s2 s1 p6 p5 p4 p3 p2 p1
Frame count within second – 6 bits
Second value – 6 bits
Marker bit – always set
Minute value – 6 bits
Hour value – 5 bits
Drop-flag – set if rate = 29.97Hz

Figure 5-8: Format of 25-bit Time-Stamp within GOP Header

If this approach is used, the state machine proposed in Figure 4-5 must be modified. The existing state machine processes the MPEG-1 Video Stream for Slice Headers in order to encrypt the Slice Payload. The new state machine must now also search for GOP Headers to extract and store the modified 22-bit time-stamp value in an internal register, as well as search for Picture Headers to extract the picture count value and combine it with the stored time-stamp value to resynchronise the cipher. This modified state machine is presented in Figure 5-9. Note that all the modifications occur in the first section of the state machine. Once the fourth state of the first section is reached, the MPEG-1 unique header identifier (0x00 0x00 0x01) has been processed, at this stage one of four things can occur:

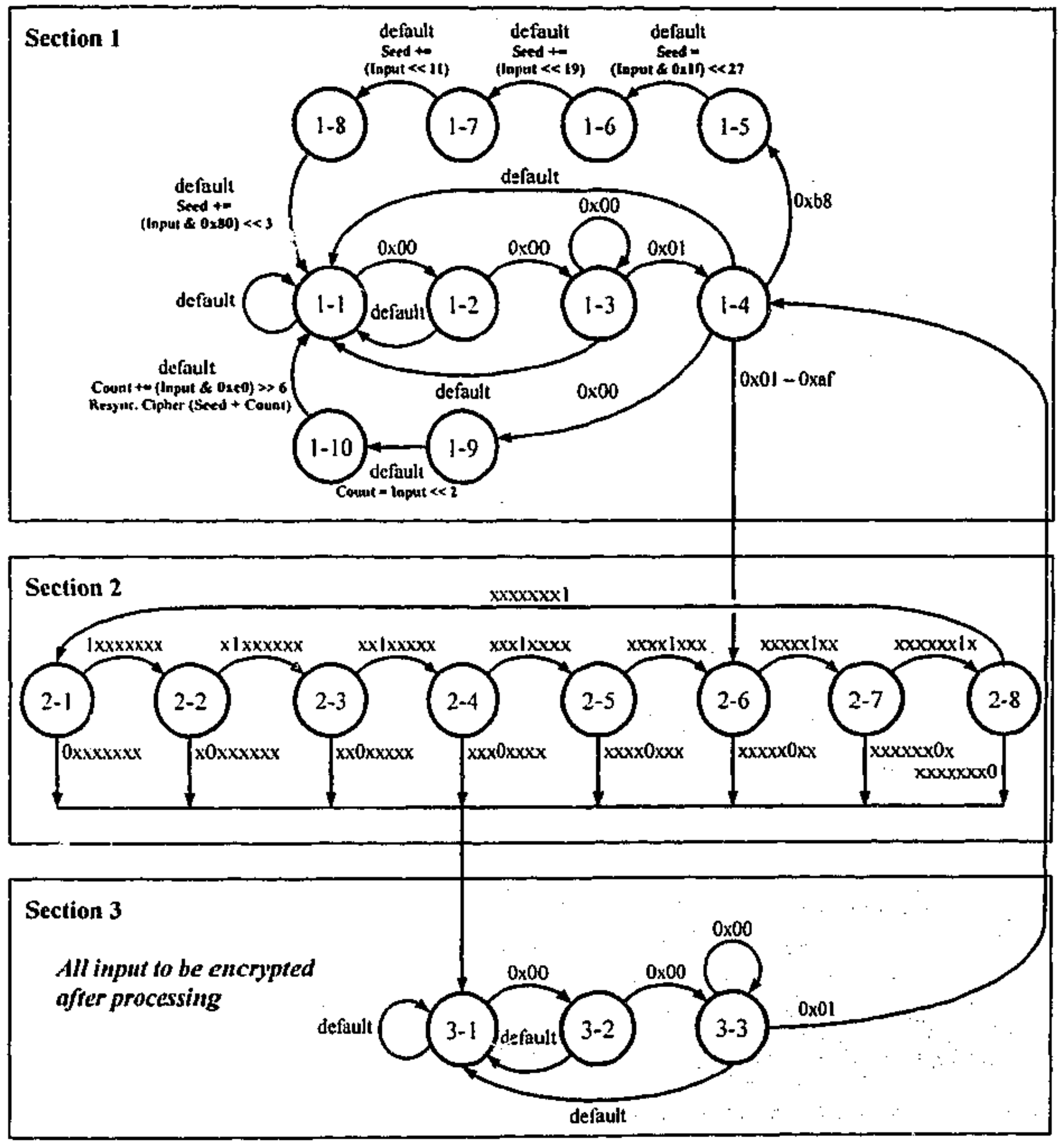


Figure 5-9: Modified State Machine to Encrypt an MPEG-1 Video Stream

- If the next byte processed is 0xb8, then we have found a GOP Header, the next four bytes contain the modified 22-bit time-stamp. We then read and assign the time-stamp to an internal variable (*Seed*) before returning to the initial state to search for another valid MPEG-1 Video Stream Header. This procedure is encompassed by states 1-5 through 1-8
- If the next byte processed is 0x00, then we have found a Picture Header, the next two bytes that are read contain the 10-bit picture count value. We read and assign this value to an internal

variable (*Count*). Once the picture count value is determined, it is added to the previously obtained time-stamp value (which has been stored such that the 22-bits occupy the 22 most significant bits) to form a unique 32-bit value. The SEAL cipher is then resynchronised to this value before we return to the initial state to search for another valid MPEG-1 Video Stream Header. This procedure is encompassed by states 1-9 and 1-10.

- If the next byte is in the range 0x01-0xaf, then we have found a Slice Header and proceed to the sixth state of the second section of the state machine so that the Slice Payload can be encrypted as in the original state machine.

- For any other value for the following byte, we return to the initial state to search for another valid MPEG-1 Video Stream Header.

When this new state machine is combined with the proposed cipher in Figure 5-7 we have a procedure that is able to encrypt an MPEG-1 Video Stream securely as well as retrieve the original plaintext stream given a single 160-bit secret key. The key is used as the primary input to the SEAL cipher to set up the initial tables. While parsing an MPEG-1 Video Sequence, the frame count field of each Picture header is combined with the timestamp field of the outlying GOP header to form a 32-bit value, *n*, which is used to select one of the 2^{32} pseudo-random sequences generated by the cipher for each individual frame. This output sequence is then modified during encryption to ensure that false MPEG-1 headers are not created as well as during decryption to retrieve the correct plaintext.

5.3 MPEG-1 Audio Stream Encryption

Encryption of the encapsulated MPEG-1 Audio Stream is more difficult due to the lack of resynchronisation points available from within the Audio Stream itself. In order to ensure that the encrypted System Stream will be able to be decoded in all playback modes – indexed, high-speed, etc. – there are a number of issues that must be addressed, these are:

- Modification of SEAL for 8-bit keystream generation rather than 32-bit, the same approach as that used for encryption of the MPEG-1 Video Stream.

- How to modify SEAL such that the basic Audio Stream structure is maintained, as required by certain video servers that check the Audio Stream format.

- How to key the SEAL resynchronisation sequence such that indexed playback remains supported at the client end, high-speed playback is performed without audio.

As presented in Chapter 4, the Audio Stream partial selection scheme is extremely simple with all four bytes of each audio frame header left as plaintext while the remainder of the stream is encrypted. I will begin by explaining how SEAL should be modified so that it can be incorporated within the existing prototype encryption system. I will then explore the lack of resynchronisation points within the MPEG-1 Audio Stream and finally propose a solution whereby we sacrifice some parts of the plaintext data within the Audio Stream to use as resynchronisation values.

5.3.1 Incorporation of SEAL Stream Cipher Into Existing System

As described in Section 5.2.1, the SEAL cipher can be again modified to process the plaintext in block sizes of 8-bits, also, this modification has no effect on the security provided by the SEAL cipher. This modification is required as the length of the blocks of plaintext to be processed is in multiples of 8-bits, terminated by a byte aligned sequence that identifies an MPEG-1 Audio Stream header. As was proposed in the Video Stream Cipher, the 32-bit SEAL output word will be broken up into four 8-bit words in a Little Endian format, with the most significant byte of the 32-bit output word first.

The next modification must be made to ensure that the cipher does not create the output byte 0xff. This modification performs the same function as that required for the MPEG-1 Video Stream in ensuring against the production of false MPEG-1 Video Stream headers, instead this guarantees against the production of false MPEG-1 Audio Stream headers. As in Section 5.2.1, where the SEAL cipher was modified to be similar in operation to the simple cipher presented in Figure 4-7, so can SEAL be modified as for the Audio Stream cipher presented in Figure 4-8 to ensure against the production of false MPEG-1 Audio Stream headers, this is shown in Figure 5-10.

$$\begin{array}{lll} \mathrm{f}(x_i,k_i): & \text{byte} & \Rightarrow \text{byte} \\ & 0\text{xff} & \rightarrow 0\text{xff} \\ & k_i \oplus 0\text{xff} & \rightarrow k_i \oplus 0\text{xff} \\ & x_i & \rightarrow x_i \oplus k_i \end{array}$$

Figure 5-10: Modification of SEAL XOR Function for use in Audio Stream Encryption

The same arguments as used in Section 5.2.1.1 apply to show that the security of SEAL is maintained through the modifications presented here. In this case however, the ideal cipher maps 0xff to 0xff and randomly maps the remaining 255 values with probability 0.00392 ($^1/_{255}$). The modified cipher instead results in the probability that a ciphertext byte is equal to the plaintext byte is only 0.00781 ($^2/_{256}$), the probability of other output bytes is 0.00391 ($^1/_{256}$). Again, while the probability that the plaintext byte is equal to the ciphertext byte is slightly higher, it provides no help in determining the actual SEAL pseudo-random output stream in attempt to compromise the SEAL cipher. Similarly, regular resynchronisation of the SEAL cipher will mean regular switching between different pseudo-random streams.

5.3.2 Lack of Resynchronisation Points within the Audio Stream

Unlike the Video Stream, the MPEG-1 Audio Stream does not provide any usable resynchronisation points within the compressed bit stream (Pan, 1995). While the start of each audio frame provides an ideal resynchronisation point, the audio frame header contains no information on the current playback position within the overall stream, thereby also not providing a source for the 32-bit value of *n* required to resynchronise the SEAL cipher. In the situation of indexed playback, streaming would begin from the start of the relevant I-Frame, the next audio frame decoded would be from the start of a valid MPEG-1 Audio Header with no information as to the current location within the overall playback stream.

At this point, there are three potential sources from which a resynchronisation value can be obtained, the first being the outlying System Stream. Unfortunately, while the System Stream does contain timestamp information and could be used to provide a resynchronisation value for the SEAL cipher, this information will not always be available. Using the Microsoft NetShow Theatre product as an example, the client player libraries allow specification of a server and video ID and provide as output and MPEG-1 Video Stream and MPEG-1 Audio Stream – already de-multiplexed from the MPEG-1 System Stream. In this situation there is no capability of obtaining header information from the System Stream in order to resynchronise the cipher.

The second option is to use the information encoded in the parallel Video Stream to provide resynchronisation values for the SEAL cipher. While this approach appears to be suitable, it can suffer from multi-threaded programming issues as the Audio Cipher must be resynchronised at the correct timestamp while the Video Stream is being decoded. Ensuring this dual stream correctness will result in a complex code base.

The final, and most suitable, option is to use the data in the Audio Stream itself in the same way we used data in the Video Stream to resynchronise the cipher. This option provides some challenges in selecting a suitable value for *n* which I will discuss below.

5.3.3 Calculating the Audio Cipher Resynchronisation Value – *n*

Given that there are no useful resynchronisation points within the MPEG-1 Audio Stream, that the presence of the information within the System Stream is not guaranteed, and that resynchronising to the Video Stream is extremely complex, it becomes necessary to manufacture a suitable resynchronisation value from the data contained in the Audio Stream. When considering resynchronisation of the Audio Cipher, only indexed playback is considered as no audio is streamed during high-speed (fast-forward or rewind) playback. I propose to sacrifice some plaintext data (2 bytes) in order to help generate a suitable value of *n* to resynchronise the Audio Cipher. Again, when decrypting the Audio Stream in real-time, this resynchronisation value can be easily extracted and used with the modified SEAL Cipher – already setup with the 160-bit secret key – to decrypt the Audio Stream as it is processed.

While it may be easy to find the start of an Audio Frame Header in a stored Stream – the headers are separated by a fixed number of data bytes – it is more difficult to do this when processing the data as a stream. Therefore, we must process the data in consecutive bytes and discover Audio Frame Headers as they happen to appear in the stream. We also know that these headers are fixed length (32 bits), and begin with the 12-bit byte aligned sequence (1111 1111 1111). We also know that each frame of audio data is representative of a short segment of playback time which varies in duration

depending on the encoding format and encoded audio rate. This implies that we will regularly be resynchronising the Audio Cipher and only using short strings of pseudo-random ouput.

There is no suitable value within the Audio Frame Header to use in calculating a value of *n* for resynchronisation purposes – the contents of the Frame Header is practically constant for each ıudio Frame. I therefore propose to sacrifice the first two data bytes following the Audio Frame Header, leaving them as plaintext to use in calculating a cipher resynchronisation value. In Layer I and Layer II Audio Streams, these two bytes will represent some of the Audio data, but not enough to successfully reconstruct a significant portion of the Audio. In Layer III Audio Streams, this data forms part of the "Frame Side Information" which describes decoding variables to the Audio decoder. In all cases, these two bytes represent changing data – their values are different for each Audio Frame within the Audio Stream. Since these bytes change as the stream is processed, we can use them to calculate changing values for *n*, such that each Audio Frame is encrypted with a different pseudo-random output from the SEAL Cipher. In order to calculate the resynchronisation value, I propose using these two plaintext bytes to form the 16 least significant bits of the SEAL resynchronisation value. The 16 most significant bits will be all set to '1'. By ensuring that the 16 most significant bits are set, we ensure that we do not repeat a resynchronisation value used by the Video Stream Cipher – the format of the timestamp in the GOP precludes this value ever being generated as part of the Video Stream Cipher resynchronisation value. This resynchronisation value will be used to select a different pseudorandom stream for each frame of Audio Data.

Unlike resynchronisation of the Video Stream Cipher, the two bytes used to calculate the resynchronisation value are not necessarily unique within the Audio Stream, meaning that some Audio Frames will be encrypted with the same pseudo-random stream. This is unfortunate and potentially leads to a less secure Audio Cipher implementation than for the Video Cipher, as repeated random streams decrease the protection afforded by a Stream Cipher. This problem is however difficult to solve since we must be able to decrypt the Audio Stream as a standalone bitstream. We cannot use timestamps as for a stored Audio Stream since we only ever see small portions of the bitstream and are unaware of the playback timestamp.

To summarise the procedure, a modification is made to the Audio Cipher State Machine presented in Figure 4-9, two extra states are inserted between the original State 1-4 and State 2-1. These two new states are used to read the first two bytes following an Audio Frame Header, append them to the 16-bit value (0xffff) and resynchronise the SEAL Cipher. The effect of these changes are shown in Figure 5-11. This ensures correct decryption during sequential processing of the Audio Stream without reference to either the containing System Stream or parallel Video Stream and allows construction of a simple module to perform decryption of this bitstream. During regular playback, resynchronisation will occur at the start of each Audio Frame ensuring correct decryption and playback, this factor will also ensure correct decryption following an indexed jump into the MPEG-1 stream. High speed playback modes are implemented without Audio Playback and therefore resynchronisation is not an issue.

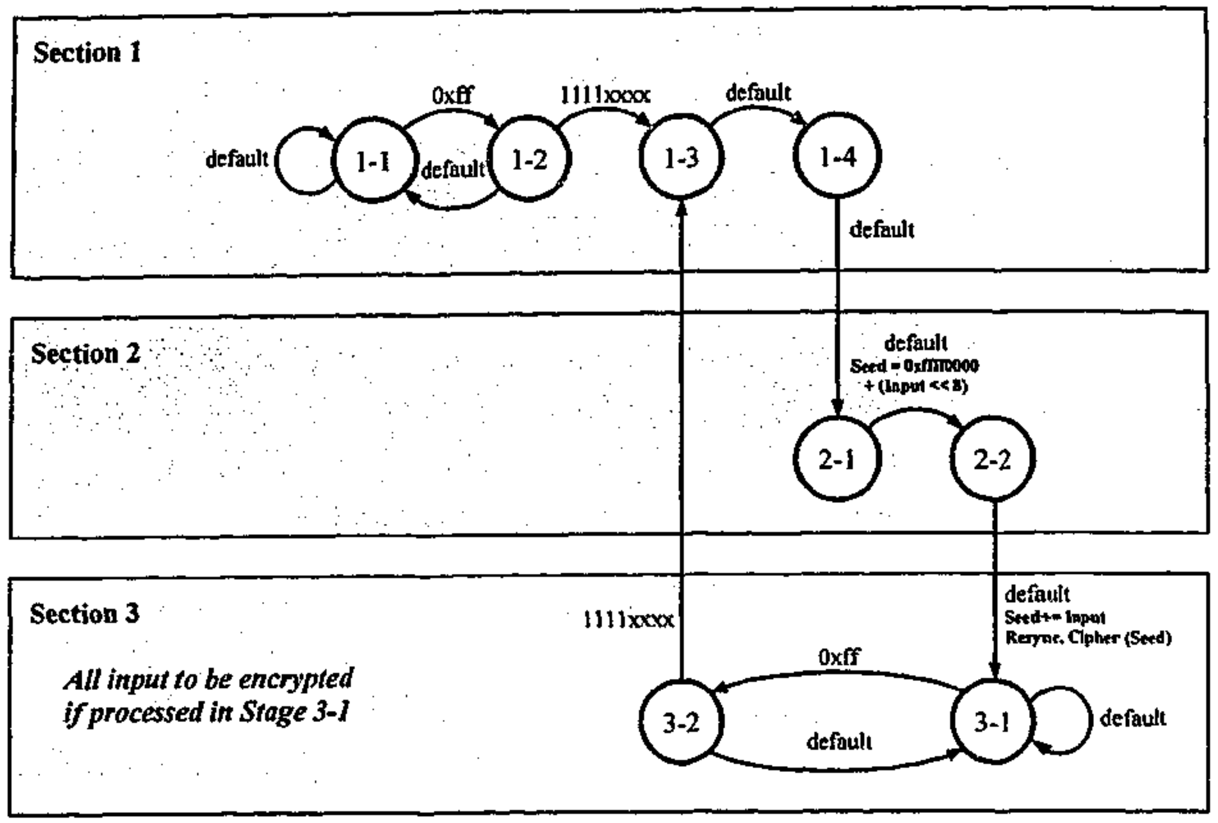


Figure 5-11: Modified State Machine to Encrypt an MPEG-1 Audio Stream

5.4 System Testing

The process of testing the new cipher involves ensuring that the new cipher passes the tests outlined in Section 4.4. These tests – repeatability and reversibility of the encryption process, CPU performance requirements for real-time decryption and ensuring fu.:.:ionality with existing streaming video servers – are performed using the same set of test files. There is no requirement to re-calculate the proportion of the MPEG-1 file selected for encryption, as this will be equal to the results presented in Table 4-1. The results of the remaining tests are outlined in the following sub-sections.

5.4.1 Is the Encryption Process Repeatable and Reversible

As in Section 4.4.2.2, answering this question involves first proving that repeated encryption of a test bitstream produces the same encrypted bitstream – as for the tests on the prototype cipher, the procedure is fully outlined in Section D.4.1 along with the results to show that the process is indeed repeatable. Of more interest is proving that the procedure is also reversible, and that the original plaintext can only be retrieved if the correct decryption key is used.

Unlike the prototype cipher, the secure cipher developed in this Chapter allows for 2^{160} unique keys, making it even more unfeasible to repeat the test using all keys. Again, a subset of four keys was selected – resulting in the same number of tests as for the prototype cipher. The four keys chosen for test purposes are listed in Table 5-1. All 2^{160} possible keys of the SEAL Cipher are considered to be secure, therefore any four keys are suitable for testing purposes. The first two keys were chosen for simplicity in entering the key for test purposes – 160 zero bits and 160 1 bits. The remaining two keys were formulat.d as the ASCII code for a 20 character text string – Key 3 and Key 4 corresponding to the strings "The MPEG SEAL cipher" and "Jason But PhD Thesis" respectively.

Keys
0x 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
0x 54 68 65 20 4d 50 45 47 20 53 45 41 4c 20 63 69 70 68 65 72
0x 4a 61 73 6f 6e 20 42 75 74 20 50 68 44 20 54 68 65 73 69 73

Table 5-1 SEAL Keys Used in Testing Encryption Scheme

The testing procedure is identical to when testing the prototype cipher and a complete tabulation of the results is presented in Table D-14. Again, the results proved consistent for each of the six test files, a summary is presented in Table 5-2. From this, we can see that all files were successfully decrypted when the encryption key was reapplied – also, attempting to decrypt a file with a different key to which it was encrypted resulted in a bitstream that was not equivalent to the original file. Again, all 96 output files were passed through software and hardware MPEG-1 decoders with similar results to the prototype cipher – the bitstream can be parsed but not decoder. As for the prototype cipher, we can conclude that the encryption process is reversible and that the original plaintext can be obtained if the correct key is used.

	Decryption Key			
Encryption Key	Key 1	Key 2	Key 3	Key 4
Key 1	☑	☒	☒	☒
Key 2	☒	☑	☒	☒
Key 3	☒	☒	☑	☒
Key 4	☒	☒	☒	☑

Table 5-2 Reversing the MPEG-1 System Stream Encryption

5.4.2 CPU Requirements for Encryption/Decryption

The second test ensures that there is sufficient CPU power available to perform real-time decryption of the MPEG-1 sequence, the procedure is the same as outlined in Section 0. The same two test platforms were used to generate the results, which are summarised in Figure 5-12.

These results were obtained using both the maximum rate and difference methods explained in Section 4.4.2.3. As expected, the increased complexity of the secure cipher resulted in increased requirements for bitstream encryption or decryption, however the speed of a SEAL implementation has meant that these extra requirements are minimal. Looking at the results, we can see that the secure SEAL Based Cipher requires approximately 50% more available CPU cycles than the prototype system presented in the previous chapter, see Figure 4-10.between 3.7% and 6% on the Pentium II and between 0.8% and 1.2% of total available CPU cycles on the Pentium 4. While this increase is large, it is still small in proportion to the CPU cycles required for decoding and playback as presented in Table 4-3.

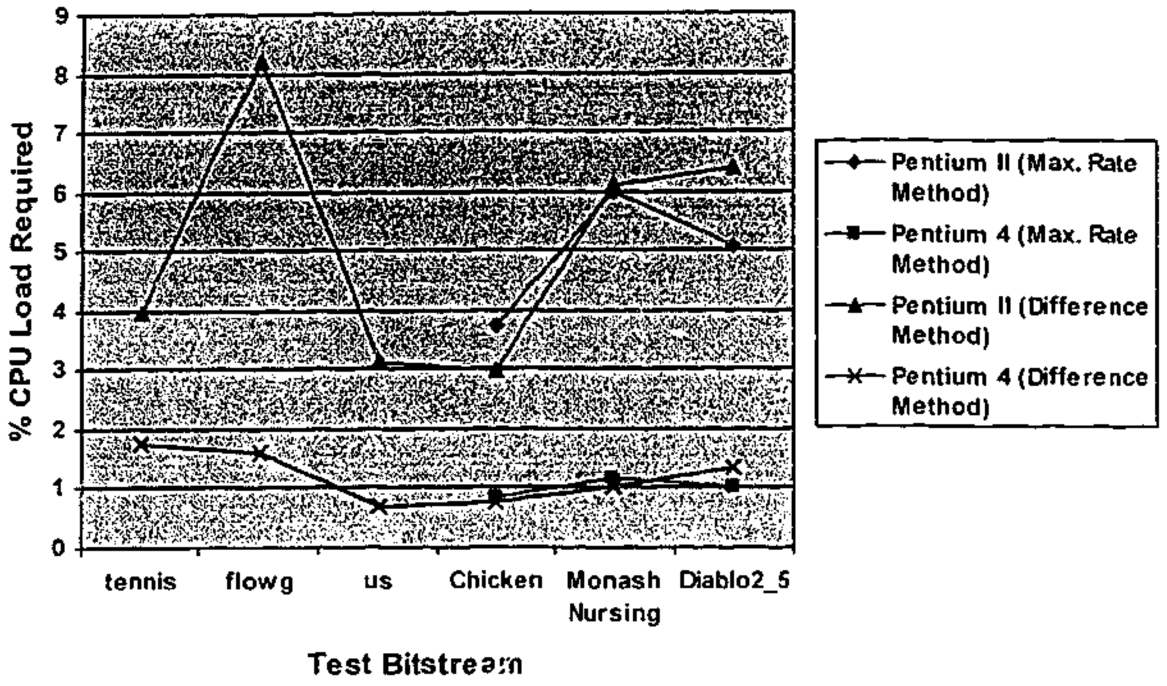


Figure 5-12: Performance Results Using the Secure Encryption Scheme

Given the conclusion that the CPU requirements for the prototype cipher indicated the processing requirements of the stream selection parsers, we can conclude the SEAL Cipher implementation is more efficient that the stream parsing algorithms. Again, these figures show that both processes (decryption and decoding) are able to function in parallel. The results demonstrate that even on the slower Pentium II test platform, as long as approximately 15% of CPU cycles are available when decoding and playing back a plaintext bitstream, then real time decryption and decoding can occur in parallel.

5.4.3 Verification of Functionality with Existing Video Servers

The final test involves verification that the encryption scheme functions in all playback modes on a variety of existing streaming servers. Again, the procedure taken here is unchanged from that described in Section 4.4.2.4. The aim is to ensure that functionality is maintained through all three test server platforms, this is done using the two Client Player applications – *"StreamCipher.exe"* and *"SGIStreamCipher.exe"* – and the Apple QuickTime Player to hint movie files for installation on the Apple QuickTime and Darwin streaming servers. The results – summarised in Table 5-3 – are again identical, all playback modes are supported by the NetShow Theatre server and correct decryption and playback was observed when streaming all test files. Playback from the MediaBase server confirmed that normal speed playback modes were supported by the cipher during real-time playback and that high-speed playback modes were supported by the cipher through a delayed playback test. Successful hinting of the encrypted movie files ensured the subsequent installation on the Apple Streaming Server platforms.

		Playback Mode				Indexed Playback Mode			
Server	Installation	▶	‖	▶▶	◀◀	▶	‖	▶▶	◀◀
NetShow Theatre	☑	☑	☑	☑	☑	☑	☑	☑	☑
Mediabase	☑	☑	☑	☑	☑	☑	☑	☑	☑
Quicktime	☑								

A blank entry in the table signifies that the functionality was not tested.

Table 5-3 Streaming an Encrypted MPEG-1 File from a Streaming Video Server

5.4.4 Summary

To summarise, the tests prove the following three important points:

- **The encryption process is reversible** – ensuring that it will be possible, through the use of the correct key, to retrieve the original sequence for decoding and playback.
- **CPU requirements for decryption are low** – ensuring that is will be possible to decrypt and decode the video stream in real-time.
- **All playback modes are supported** – playback functionality is not restricted as a result of encrypting the video stream, therefore not restricting user interaction with the video streaming service.

Having passed these requirements, we can say that the proposed video encryption scheme can be utilised in a streaming video service while remaining invisible to the end user – enabling service providers to provide a secure streaming service for content owners while not unduly inconveniencing paying customers.

5.5 Effects on Streaming Video Implementations

Having proven the functionality of the MPEG-1 cipher, it remains to be seen what effects the cipher has on the performance of the MPEG-1 streaming application. Of particular interest are the effects of network errors on the encrypted video stream as opposed to the same errors occurring on the un-encrypted video stream. In this section I will discuss the effects of the following transmission errors, comparing the cases of streaming both the encrypted and plaintext video. In each case, the comparison will consider the effect on the effective performance – to what extent is the quality of the video and audio affected as observed by the end user? The types of network errors considered are:

- **Bit Error** – an individual bit (or multiple bits) within the stream is flipped to an incorrect value.
- **Bit Loss** – an individual bit (or series of bits) within the stream is lost in transmission.
- **Dropped Packet** – an IP packet is dropped somewhere in the network during transmission.
- **Late Packet** – an IP datagram has arrived late (after the scheduled decode time) at the client player.

5.5.1 Effect of a Bit Error

A bit error occurs when a single bit – or multiple bits – has its value flipped during transmission across the network. Since video streaming is often accomplished using the User Datagram Protocol (UDP), these sorts of errors can usually be detected but not corrected, as data re-transmission would result in the corrected data arriving too late to be displayed. The probability of this error occurring is entirely dependent on the medium being used to implement the network connection, fibre networks provide for extremely reliable data transfer (Bit Error Rate = $10^{-8} - 10^{-12}$) while error rates on wireless links are higher ($10^{-3} - 10^{-6}$). The problem is not how to fix the bit errors, since these are a function of the network and the underlying protocol layers, but rather what effect these bit errors have on the streaming video being displayed to the user.

If a bit error occurs in the process of streaming back a plaintext video, it will result in one of a number of visible or audible glitches presented to the user. The overall effect of this glitch is dependant on which bit in the video stream has been flipped to an incorrect value. Since the video bit stream is compressed, the bit error will result in a larger number of errors to the un-compressed video stream which is displayed to the user, ranging between a small area of the screen being in error in a single frame, the error propagating over several frames, an instantaneous error in the audio playback (presented as a squeak or a clicking sound) or no visible or audible error at all. Due to the nature of video streaming – transmission of a compressed bit-stream over a potentially error-prone channel – these errors cannot be repaired and are therefore part and parcel of streaming video data. What is of interest is – if the same bit error occurs in the encrypted video bit-stream, is the resultant effect on playback equal to or greater than if the bit error occurred in the plaintext bit-stream. There are three possible scenarios to consider in which the value of a bit is in error:

- The erroneous bit occurs within a macroblock or audio packet that has been encrypted.
- The erroneous bit occurs within a part of the video bit-stream that is not encrypted.
- The erroneous bit occurs in a part of the bit stream that is used to determine when to switch the cipher module on or off or to determine a resynchronisation value.

The first of these scenarios can be summarised as resulting in the same playback errors as the un-encrypted bit-stream, this is due to the underlying XOR nature of a Stream Cipher. At the client end the same pseudo-random bit-stream will be generated, when this is XORed with the video stream, the bit that was in error will remain in error – the erroneous bit does not propagate to other bits within the video stream. In the scenario of an encrypted bit having its value flipped, there is no difference in playback between the encrypted and plaintext video bit-stream. The second scenario also results in the same playback errors as the plaintext bit-stream – this is obvious since the error in both bit-streams is exactly the same.

The final scenario is more interesting. Certain elements of the MPEG-1 Video Stream and MPEG-1 Audio Stream are used to both synchronise and start and stop encryption of bytes within the bit-stream. If any bits which encode this information suffer from bit errors, this error will propagate to cause larger errors in the decoded video. The effects could be catastrophic.

Consider for example a bit within the timestamp of a GOP header being in error, this will cause an incorrect cipher resynchronisation value to be used for all video frames within that GOP, resulting in an incorrect pseudo-random bit-stream being used to decrypt the video. The overall effect will be incorrect video playback for the duration of the GOP, approximately 0.5 seconds of bad data. A less extreme example occurs if a bit determining the picture count is in error, leading to an incorrect resynchronisation value for decoding the given video frame. Errors in the two resynchronisation bytes following an audio frame header will lead to an incorrect resynchronisation value for the given audio frame, again completely destroying the data in that frame. Errors in bytes signifying the start of MacroBlock or audio frame data will result in the cipher not being started (or stopped), causing erroneous decoding until the next frame header is encountered. While the percentage of the overall stream that makes these critical points is small, the potential effects can be dramatic, both visually and aurally. This implies that streaming the encrypted video over an error prone channel can potentially cause a lower quality of playback that streaming the plaintext video over the same channel. It is also important to note that a bit error in any one of these fields could cause the Video and Audio decoder modules to incorrectly process that block of data, leading to problems in playback of the same duration but not quite as severe.

Another issue that can become a factor is how bit errors are dealt with by the video streaming software. As noted, video streaming is typically performed using UDP, allowing the discovery but not the correction of transmission errors. Different implementations of video playback software can deal with this scenario in one of two ways:

- Consider the importance of the real-time data and determine that playback of erroneous data is preferable to no playback. In this case, the errors described above become a potential issue, especially in the scenario where the bit that is in error plays a key role in the implementation of the cipher.
- Determine that the packet is in error and throw it away, this has the same effect as a dropped or lost packet as described in Section 5.5.3

When considering the effect of a bit error in streaming an encrypted video, it is important to note that if the bit error occurs in a part of the bit stream that is used to synchronise and control the cipher, the error is propagated through a large number of bits, potentially affecting up to half a second of playback. Seeing that the bytes that synchronise the cipher are limited in number and form a small part of the entire stream, this type of error should not be prevalent. Of greater interest, if the bit error occurred in another part of the bit-stream, the resultant playback error will be exactly the same as if the error had occurred in streaming the plaintext video. Combined with the low probability of the propagated error, we can conclude that using encrypted video has only a minor effect on the perceived quality of video playback.

5.5.2 Effect of Lost or Dropped Bits

A lost or dropped bit occurs when the physical transmission medium loses a bit in transfer. Data is usually transmitted using a serial protocol at the Physical Layer and there is always the potential for a bit to be lost during transmission. The final result is that the bit-stream received at the destination is missing a bit. However, it is important to note the effects of the Link, Network and Transport Layers that sit underneath the video streaming application. In the instance of the Internet Protocol (IP), any packets that suffered from a dropped bit during transmission would cause the IP packet to be dropped, having the same effect as a lost or dropped packet discussed in Section 5.5.3. To summarise, lost or dropped bits are handled by lower layers in the protocol stack, therefore ensuring that a lost bit error does not propagate to the application as a dropped bit error, but instead as a lost or dropped packet error. This is fortunate as dropped bit errors are not handled well by either Stream Ciphers – which result in all bits following the dropped bit to also be in error – or the MPEG-1 format – which relies on certain parts of the bit stream being byte aligned.

5.5.3 Effect of Lost or Dropped Packets

A lost or dropped packet in video streaming occurs when one of the transmitted UDP datagrams fails to arrive correctly at the client playback application, therefore the data contained within the datagram cannot be decoded and played back. UDP datagrams can get lost or dropped in transit, as each datagram is often fragmented into a number of IP packets (dependent on the size of the Maximum Transmission Unit), an entire UDP datagram is dropped if one of these IP fragments is dropped. IP fragements can be dropped or lost in one of a number of ways:

- **By the router** – Due to lack of buffer space within the router queues.
- **By the network** – Due to a timeout during transmission of the IP fragment.
- **By network failure** – Causing some packets to be lost while the network repairs and re-routes traffic.
- **By bit error** – An error in the bitstream will cause the IP Layer to drop the packet.

To minimise the effect of a dropped IP fragment, the size of the UDP datagrams should be kept as small as possible – in this way, a single dropped IP fragment would effect a smaller portion of the overall data stream. However, larger UDP datagrams place less stress on the Operating System of the streaming server and can lead to more efficient implementation of a Streaming Video Server. While some streaming servers allow specification of the UDP datagram size. many do not, what is common however, is that many streaming servers have a default UDP datagram size of 16kB. This forms a compromise between having many small UDP datagrams of 1500 bytes (Ethernet Frame Size) and the maximum datagram size of 64kB.

When a UDP datagram is dropped from a plaintext video stream, the effect this has on the ultimate visual/aural playback of the stream is variable, depending on the size of the UDP datagram and what sections of the overall MPEG-1 stream was contained within that packet. The decoder will not only lose all the data that was contained within the packet, but also all subsequent data until it can resynchronise itself with a valid MPEG-1 Header sequence within the stream. Considering an MPEG-1 stream encoded at 1.5Mb/s, this would be streamed as 12 separate 16kB UDP datagrams per second. Table 5-4 shows average percentages of an MPEG-1 stream given for each of the individual types of frames.

Frame Type	# in typical GOP	Overall % of stream	Per-frame % of stream
I-Frame	1	50 %	50 %
P-Frame	3	40 %	13.33 %
B-Frame	8	10 %	1.25 %

Table 5-4 Proportion of Video Stream given Frame type

Now consider a dropped datagram that contained data from an I-Frame, a single I-Frame requires approximately 36kB of the overall stream – implying that the datagram will contain solely information from that I-Frame. This will disrupt the decoding and display not only of that I-Frame, but also of all subsequent frames within the GOP, since they are encoded as variations on the initial I-Frame. Dropped datagrams that effect P and B-Frames will generally directly effect more than one actual frame, with the respective sizes of these frames being on average 12kB and 2.5kB.

To summarise the effect of a dropped datagram, missing I-Frame information will affect the display of all remaining frames within the GOP, causing about 0.5 seconds of discontinuity within the video playback, audio playback will be less affected since each audio frame is approximately the same size and the data within the audio frame is independent of previous frames. Missing P-Frame information will also affect the display of any remaining frames within the GOP, this is because all subsequent P and B-Frames are dependent on the preceding P-Frame. This effect is not as dominant as missing I-Frame information but can still lead to discontinuities of up to 0.35 seconds. Missing B-Frame information will only effect the decoding and display of the frame under consideration, however the small encoded sizes of B-Frames means that it is likely that a 16kB datagram will not only contain multiple B-Frames but also portions of other frames as well (I or P-Frames). In a plaintext video stream, the effect of a dropped datagram leads to considerable video playback discontinuity – the question remains if this problem is exacerbated for an encrypted video stream.

Dropped datagrams within an encrypted video stream would cause a slightly greater effect than for the unencrypted stream – this is primarily due to the fact that both the cipher and MPEG-1 decoding module must be resynchronised instead of just the decoder. If any of the data representing the cipher resynchronisation is lost, all subsequent data cannot be decrypted until the cipher is resynchronised. To summarise, if a GOP Header is dropped, all subsequent video and audio for that GOP cannot be decrypted, leading to both visual and aural loss of playback for approximately 0.5 seconds. If a Picture header is dropped, the subsequent frame represented by that Picture also cannot be decrypted. In many respects, the extra data that cannot be decrypted while waiting for a cipher resynchronisation point is equal to the block of data that couldn't be decoded correctly due to missing information from previous frames.

In conclusion, dropped UDP datagrams can happen due to network congestion and transmission errors. When streaming encrypted video, a dropped datagram has a slightly larger effect than for streaming unencrypted video where there is a potential of up to 0.5 seconds of playback discontinuity. This means that dropped datagrams are a serious problem for video streaming in general and not only to streaming encrypted video.

5.5.4 Effect of Late Delivery of a Packet

Streaming video applications typically use the UDP Protocol to transfer the video across the network because UDP is more suited to real-time applications. It allows the data to be put onto the network at regular time intervals independent on the current state of network congestion. Because of this dependence on real-time delivery of the video stream, it is essential that the UDP datagrams arrive at the destination within a given time-period – defined by both the MPEG-1 decoder and the amount of buffering used at the client. Given the non-QOS nature of the underlying IP network, it is possible for datagrams to arrive beyond their intended playback time. In this instance, late delivery of a packet or datagram means that the data cannot be displayed on time, this leads to one of two results depending on the client player implementation:

- Any late data is not played, as soon as data arrival falls within the required time boundaries, playback recommences – this results in a discontinuity in video and audio playback.
- When data is not present in the decoder, playback pauses until data arrives and playback recommences – this results in a pause in video and audio playback.

In all cases, when streaming plaintext video streams, data that arrives late is still fed into the decoder, the decoder will determine that the data is late and not process the stream through the separate Video and Audio decoders. This allows the decoders to process the System Stream headers and ensure that playback can commence promptly the instant that data arrives within the scheduled playback time. Similarly, all received data – even late arrivals – must be decrypted to ensure that cipher resynchronisation is up to date and that the bit-stream will be decrypted for immediate decoding and playback as soon as possible. As such, the effect of late arrival of an encrypted video stream will be exactly the same as that for a plaintext video stream, either a discontinuity or a pause of video and audio playback

5.6 Conclusion

In conclusion, in this chapter I investigated the possibilities of extending the prototype MPEG-1 cipher presented in Chapter 4 in order to improve the level of protection offered. I began by investigating the suitability of certain existing ciphers for their incorporation into the prototype cipher. This resulted in the conclusion that Stream Ciphers were best suited to the type of modification required for incorporation into the existing model. A review on two common Stream Ciphers revealed that the SEAL cipher was particularly suited in that it offered the best encryption speeds and the fastest cipher resynchronisation times. The next step was to describe the necessary modifications the SEAL cipher such that it could be incorporated within the framework of the prototype cipher – this included demonstrating that these modifications would not it any way effect the overall level of security afforded by the basic SEAL cipher.

Having designed the new cipher, the existing prototype cipher implementation was extended to support the newly designed cipher, this new implementation was then tested for functionality. The cipher was subjected to a similar range of tests as those performed in Chapter 4, which ensured that not only was it possible to retrieve the original MPEG-1 sequence, but also that it was possible to stream and playback the file from a variety of streaming video servers. Once the viability of the cipher was proved, it became necessary to compare the effects of network transmission errors on streaming encrypted video as opposed to the same problems occurring in a plaintext stream. To this end, an analysis was performed on the possible types of network transmission errors – this analysis concluded that while the effects of a network error on an encrypted stream resulted in potentially greater visual and audio playback errors, these errors were considered not to be much greater than if the same problem occurred in a plaintext stream. Since the importance of copyright protection is paramount in the provision of a video streaming service, the trade-off against potentially minor quality of video problems in streaming encrypted video is considered to be worthwhile.

Chapter 6

Application to Streaming MPEG-2

In the previous two chapters I have described the development of a new Partial MPEG-1 Stream cipher and shown it to be compatible with the requirements of the ideal cipher outlined in Chapter 2. This cipher is demonstrably secure in the protection of both video and audio content within the streaming media and has proven to be compatible over a range of existing Video Streaming products. I have concentrated on the MPEG-1 Video Compression Scheme in particular since this system is most likely to be used in the first high-quality video streaming applications – while broadband data rates are likely to support MPEG-2 streaming directly, the extra requirements on the streaming servers, the network core and the maximum network throughput of these servers mean that an MPEG-2 service can only service a smaller customer base. While an MPEG-1 based entertainment streaming video application is the most likely initial product, as network core bandwidth and access rates improve, it is inevitable that future streaming video applications will offer higher quality MPEG-2 streams (DVD quality as opposed to VHS quality).

It is important to consider how copyright protection could be enacted in an MPEG-2 Video Streaming system. In this chapter I will describe the existing MPEG-2 encryption scheme, as well as briefly explore how the cipher developed in the previous two chapters could be modified to support MPEG-2 encoded video and audio. This treatment will explore the basic steps required to develop an MPEG-2 cipher but not continue with the development and testing of such a system.

6.1 MPEG-2 Scrambling

The MPEG-2 standard supports the concept of scrambling, or copyright protection of the media stream through encryption. Within the MPEG-2 Program and Transport Streams, there is the capability of specifying details on the cipher chosen and a series of two bits which indicate how the encoded stream has been encrypted.(Haskell et al., 1997)

6.2 MPEG-2 Stream Format

There are two different types of valid MPEG-2 bitstreams, commonly referred to as the MPEG-2 Program Stream and the MPEG-2 Transport Stream. The MPEG-2 Program Stream is analogous to the MPEG-1 System Stream and was defined primarily to provide the same type of services. The MPEG-2 Transport Stream was initially designed for streaming over packetised digital networks. Also forming part of the MPEG-2 standard are the MPEG-2 Video and Audio Stream formats. In the next sections, I will briefly describe the basic format and structure of these different types of bitstreams.(Haskell et al., 1997)

6.2.1 MPEG-2 Program Stream

The basic purpose of the MPEG-2 Program Stream is the same as for the MPEG-1 System Stream, and that is to multiplex together multiple MPEG-2 Video and Audio Streams. While the structure of the streams – including the MPEG start codes – is different between the two digital compression standards, the basic structure is similar. The MPEG-2 Program stream takes sections, referred to as packets in the standards, of the underlying Video and Audio Streams and stores them sequentially with header information indicating which stream the packet belongs to as well as decoding and presentation timestamps to indicate the timing at which these packets should be passed to their respective decoders.(Haskell et al., 1997)

It is interesting to note that the basic structure of the MPEG-2 Program Stream is very similar to that of the MPEG-1 System Stream (Haskell et al., 1997). This implies that MPEG-2 Program Stream decoders are very similar in construction to MPEG-1 System Stream decoders. It also implies that if a Streaming Video Server was to stream video compressed as an MPEG-2 Program Stream, then the construction of the server would be almost identical to the construction of existing MPEG-1 Streaming Servers. In terms of encryption of Streaming MPEG-2 Program Streams, we can in general take the same approach to the MPEG-1 Cipher by leaving the Program Stream information as plaintext.

6.2.2 MPEG-2 Transport Stream

The MPEG-2 Transport Stream was specifically designed for the application of streaming MPEG-2 compressed video over networks, considering the probability of transmission errors. While it is not explicitly stated in the standards, the Transport Stream is particularly suited towards transmission over an ATM network – indeed, the MPEG-2 Standards were discussed when it was still considered a real possibility that ATM, and its various Adaptation Layers, would supersede the Internet Protocol (IP) as the predominant Network Layer protocol in the Internet. This is particularly evident when we consider the small, fixed-sizes of the Transport Stream Packets.(Bridie, 1997; Cornall and Lipton, 1997; Egan, 1998; Grimm and Cornall, 1998)

The current situation suggests that while we are likely to find ATM networks in the core of the network, this is to manage bulk data transfer in a point-to-point transfer as opposed to the original idea of ATM-to-the-desktop. The failure of ATM in this regard has meant that many of the facilities provided by the MPEG-2 Transport Stream are either no longer required or not as important as they once were.

While performing a different function to the MPEG-2 Program Stream, the Transport Stream is nonetheless similar. This is due to the nature of the MPEG-2 Video and Audio Streams which are broken up into packets for inclusion into the Program Stream. In MPEG-2, the Video and Audio Streams are broken up into various sized chunks and termed as a Packetised Elementary Stream (PES) – packetised because the divisions for inclusion into the Program Stream are defined in the Video and Audio Streams and elementary because they form a single segment of the combined multimedia stream.(Haskell et al., 1997)

The packets within the Transport Stream are fragmented into smaller 188 byte payloads, similar to the same way in which the IP Protocol fragments datagrams to fit the Maximum Transmission Unit. The Transport Stream Header then contains information about the contents of the payload. It is up to the Transport Stream decoder to reconstruct the PES packets that made up the original Video and Audio Streams, and forward them to the appropriate decoder (Video or Audio). While the Transport Stream is more complex than the aforementioned Program Stream, the basic ideas of encryption of only the encapsulated Video and Audio Streams can still be applied to Transport Stream encoded media.

6.2.3 MPEG-2 Video Stream

The MPEG-2 Video Stream can be considered to be a superset of the MPEG-1 Video Stream, a valid MPEG-1 Video Stream can be successfully decoded by an MPEG-2 Video Decoder. Also, the layers in the MPEG-2 Video Stream have the same names and basic format as per the MPEG-1 Video Stream. The number and type of fields contained within the various headers of the stream differ, but the argument presented earlier that header information can remain as plaintext is also valid for MPEG-2 Video Streams.(Haskell et al., 1997)

The basic format of the MPEG-2 Video Stream is the same as for an MPEG-1 Video Stream – the compressed data representing the actual video content resides wholly in the encoded Macroblocks. This would seem to imply that the same approach can be taken when selecting which segments of the MPEG-2 Video Stream to encrypt – only processing the MPEG-2 Macroblocks with the chosen cipher. The procedure is not as simple as this however as there still remains the issue of resynchronisation of the cipher at key points. This will be addressed briefly in Section 6.3.1 when we discuss applying the previously developed MPEG-1 cipher to an MPEG-2 Stream.

6.2.4 MPEG-2 Audio Stream

The Audio component that makes up an MPEG-2 Media Stream has many options, including the possibility of simply encoding the Audio using any of the three existing MPEG-1 Audio coding schemes. The more complex approach is to use the MPEG-2 Audio coding scheme which in turn is backward compatible with MPEG-1 Audio Streams.(Haskell et al., 1997)

For the MPEG-2 Audio Stream, the basic structure is very similar to the MPEG-1 Audio Stream. More particularly, the MPEG-2 Audio Headers contain the same information as the MPEG-1 Audio Headers. Any extra information for the MPEG-2 Audio Stream (extra encoded channels and descriptive information) is encoded as ancillary data within the MPEG-1 Audio Stream format. As for the MPEG-2 Video Stream format, this implies that we can adopt the same basic approach when considering encryption of an MPEG-2 Audio Stream. Again the predominant issues will involve resynchronisation of the cipher and more particularly tying the resynchronisation to the MPEG-2 Video Stream.

As a final note, it is interesting to consider that there is also a different option when encoding multiple Audio channels using MPEG-2. In this case the normal stereo channels are encoded using either MPEG-1 or MPEG-2 encoding techniques in one Audio Stream while the remaining channels – centre and rear stereo – are encoded in a separate Audio Stream. While this allows for more efficient encoding of both stereo and surround sound tracks, this can seriously complicate matters when looking at encryption of Audio Streams. In this situation resynchronisation of the cipher becomes not only more important, but also more complex as both Audio Streams will be decoded simultaneously.

6.3 Compatibility with the Proposed Cipher

One of the key aspects of the cipher developed in Chapters 4 and 5 was that it was compatible with all existing Video Streaming Servers. Similarly important, decryption could be performed prior to decoding rather than as a key part of decoding thereby increasing compatibility amongst existing MPEG-1 decoders. This compatibility with a wide range of existing streaming servers and decoders was required to accommodate the issues of multi-platform distributed server implementations as well as the proposed Multi-Party Distributed Streaming Server design. When considering encryption of MPEG-2 compressed video the same conditions are just as important, therefore we must ensure that any proposed scheme is both compatible with existing server products as well as existing MPEG-2 decoders.

The basic technique applied for encryption of MPEG-1 streams can also be applied to encrypt an MPEG-2 stream. The Program and Transport Stream information is left unencrypted as this would not hide any data that can be used to reconstruct any Video or Audio sequences. On the other hand, the encapsulated Video and Audio Streams should be encrypted to a level that their content is protected while any information pertinent to providing indexed or high-speed playback is left in a plaintext format. In the following sections I will discuss the basic procedure for implementation for each of the MPEG-2 bitstream types.

6.3.1 MPEG-2 Video Stream

In Section 6.2.3 I briefly outlined the major differences between the formatting of the MPEG-2 Video Stream as opposed to the MPEG-1 Video Stream, the conclusion being that an MPEG-2 Video Stream is a superset of the MPEG-1 Video Stream. The same basic approach described to encrypt an MPEG-1 Video Stream in Chapters 4 and 5 can be adopted in order to encrypt an MPEG-2 Video Stream. This in turn will function correctly with the decision not to encrypt any data within the MPEG-2 Program or Transport Streams, corresponding to not encrypting the MPEG-1 System Stream.

While the basic approach is compatible with MPEG-2 bitstreams, there are a few technical details that would need to be finalised before an exact algorithm can be proposed. These

details revolve around the resynchronisation of the underlying cipher module, as well as ensuring functionality across a range of different server and decoder platforms. This is primarily an implementation issue – applying applying the aforementioned cipher design to a differently formatted source bitstream and would involve:

- **Analysis of MPEG-2 Streaming Video Servers** – How does an MPEG-2 Streaming Server provide functionality such as indexed and high-speed playback. This enables the determination of key points within the Video Stream that it is essential that the cipher be resynchronised in order to enable correct decryption in all of these playback modes.
- **Playback at the Client** – How does the client-end of a Video Stream obtain and decode the received data in various playback modes. This is also important in both the determination of key cipher resynchronisation points as well as selection of a resynchronisation value.
- **Consideration of a base Stream Cipher to use** – The SEAL Cipher worked well when encrypting MPEG-1 Streams but the higher bit-rates common to MPEG-2 bitstreams may mean that pseudo-random string length of the SEAL Cipher (Schneier, 1996a; Rogaway and Coppersmith, 1998) is unsuitable for encryption of an MPEG-2 Video Stream. If this is the case, it may be necessary to locate a different Stream Cipher that offers suitable protective qualities, while at the same time being amenable to frequent and quick resynchronisation. If another Stream Cipher is chosen, it should be amenable to the modifications outlined in Section 5.2.1 in order to avoid the generation of false MPEG Headers.

6.3.2 MPEG-2 Audio Stream

There is also the requirement to consider how the MPEG-1 Audio Stream Cipher is amenable to application to an MPEG-2 Audio Stream. In this case I note two items:

- An MPEG-1 Audio Stream can form a valid Audio bitstream within an MPEG-2 Program or Transport Stream. This means that any MPEG-2 Streams containing MPEG-1 Audio can be encrypted using the same technique as for the previously developed cipher.
- An MPEG-2 Audio Stream has the same basic format as an MPEG-1 Audio Stream using the same basic header format, again implying that the previously developed technique – encrypting all data bar the MPEG-1 Audio Headers – could be applied to an MPEG-2 Audio Stream.

The primary issues involved in encrypting an MPEG-2 Audio Stream again revolve around the problems of resynchronisation of the cipher to enable support of indexed playback. As when considering the encryption of an MPEG-1 Audio Stream, there are no timed reference points within an MPEG-2 Audio Stream that can be used to provide a resynchronisation key for the cipher. Therefore it will again be necessary to use data stored within the Audio Stream to provide the Cipher resynchronisation values.

The other issue involves the increased complexity of MPEG-2 Audio with its multiple encoded Audio channels. As previously mentioned, this is possible using multiple Audio Streams – each encoding 2 or more of the total surround channels – which are then multiplexed in the Program or Transport Stream. During playback, this potentially involves more than one MPEG-1 or MPEG-2 Audio Streams being decoded and played back simultaneously, thereby requiring more than one Audio Stream being decrypted simultaneously as well. To ensure that the security of the cipher afforded is high, it is necessary to encrypt both Audio Streams using different pseudo-random strings. As the proposed cipher can potentially re-use resynchronisation values within the Audio Stream, this problem becomes more prevalent when multiple Audio Streams are considered. It may be necessary to develop a slightly different approach to protect the encoded Audio content, possibly involving the use of different primary keys for each encoded stream. This will greatly increase the complexity of the original proposed cipher, requiring that the Stream ID of the Audio Streams somehow be incorporated into the selection of the key used to decrypt and encrypt the Audio Streams.

These issues aside, there is no reason that the basic principle used to encrypt an MPEG-1 Audio Stream cannot be reapplied to encrypt an MPEG-2 Audio Stream. Indeed the partial selection scheme that determines which parts of the Audio Stream are selected for encryption outlined in Section 4.3 will still apply. Similarly, the modifications to the chosen stream cipher to protect the basic format of the Audio Stream outlined in Section 5.3 will also still apply. A modified MPEG-1 Audio Stream Cipher would involve revisiting the issue of resynchronisation while at the same time ensuring that multiple Audio Streams could be encrypted for simultaneous decryption and playback.

6.4 Conclusion

Copyright protection of MPEG-2 streamed video is just as important as for MPEG-1. The higher bitrates of MPEG-2 encoded video however ensure that streaming MPEG-1 Video is more likely to be initially adopted as there are lower, and therefore cheaper, network requirements. An obvious question is whether the previously proposed MPEG-1 encryption algorithms can be applied to an MPEG-2 Stream. The easy answer to this question is not directly, but the ideas developed in encryption of an MPEG-1 bitstream are applicable:

- Processing of the outlying Program or Transport Stream is more complex, however the same approach of leaving this bitstream as plaintext data still applies. It still remains possible to encrypt the encapsulated Video and Audio Streams in-place within the outlying Program or Transport Stream.
- Partial selection of only Macroblocks within the Video Stream for encryption purposes also remains a valid technique for protection of video content. It still remains imperative to ensure that false Video Stream headers are not created.
- Leaving the Audio Frame Headers intact within the Audio Stream is also valid for MPEG-2 bitstreams.

- Resynchronisation of the cipher to enable implementation of indexed and high-speed playback modes remains an issue. This would require an investigation into how this functionality is implemented in different Streaming Video Servers. A cipher resynchronisation scheme must then be developed that both ensures that these features are unaffected while allowing correct decryption and decoding under the various playback modes. This is also a key issue when considering indexed playback of the encrypted Audio Stream.
- Security considerations are also complicated by the possibility of one or more concurrent Audio Streams within the outlying Program or Transport Stream. An analysis should be performed to determine whether the existing approach will suffice or whether re-utilisation of pseudo-random sequences becomes too common. In this case, each Audio Stream should be encrypted with a different key – this complicates transmission and usage due to the existence of multiple keys.

Chapter 7

Conclusion

Video Streaming is a networked application which has yet to truly come of age. Until recently, the major challenges facing networked video applications have been technical – developing systems that function under technological throughput and processing power limitations. Now that technical solutions are starting to become available, it becomes important to address other issues that are important when constructing a viable, commercial video streaming service. One of those issues is that of Copyright protection. While not the only issue to be addressed, Copyright is important in that if not addressed, no content owner will allow content to be made available for networked streaming purposes.

Video Streaming is an encompassing term that covers a range of different applications:

- **Low Bit-Rate Streaming to Mobile Terminals** – Current and future wireless networks will have increased capacity to cope with provision of a large scale video delivery service. Delivery to mobile devices necessarily entail delivery to units with a small playback screen, this implies that lower bit-rates can be employed as high-resolution picture quality is not required. This service may find usage in applications such as media access while waiting (public transport, in a bank queue, etc.) where users may choose to watch short shows (30-60 minute comedies or dramas) or obtain a news/sport update. For this application, the key technical issues include efficient coding a low bitrates, delivery of a large number of consecutive streams, coping with network conditions (error rates, congestion) and content access control. While Copyright is always an issue, the quality of video being delivered to these terminals makes copying less of a concern than for a high-quality stream.

- **Internet Streaming to a Personal Computer** – Delivery of video services over the general Internet to any Internet connected workstation. At present, Internet video delivery is either free or uses access control to restrict access to members of a particular site. The video content itself however is not protected against copying or downloading by a user. Current implementations are generally not scalable, servers can deliver a limited number of concurrent streams and quality depends on current network conditions between the two parties. Similarly, Internet video content today is predominantly either copyright free, of little value and therefore not worth stealing, or of sufficiently poor quality to negate against theft.

- **Entertainment quality Video** – Delivery of high-quality digital video content to a user for entertainment purposes. This type of application involves delivery of high resolution video, most likely to a black box connected to a television set in the users living room, but also potentially to an Internet connected PC. The general concept is to replace local playback of video tape or DVD with delivery over the network. The type of content accessed will generally be of pure entertainment value – movies, sports broadcasts, television shows – and its quality will require some form of protection against theft of the digital stream.

For any computer based application to succeed, it must provide a beneficial service to all users of the application. With networked applications in general, there are usually two classes of users, the service provider and the consumer. For streaming video applications there is a third class of user, the content provider. The issues involved in providing a successful video streaming service are not only technical, they also involve placating any concerns of all three user groups. If an implementation does not appear suitable to any one user group, then that group will withdraw its patronage of the service. For a video streaming service to be viable, all three groups must be present – if there is no content provider there is no suitable content to stream, if there is no consumer there are no users to pay for the service, and if there is no service provider there are no platforms to deliver the service from.

One of the primary concerns of the content provider is that of Copyright and Copyright protection. The Copyright on entertainment media, such as movies or sporting broadcasts, costs large sums of money, and is purchased as an investment where the return on that investment is royalty fees paid by those accessing the content. In the case of free-to-air television broadcasts, the royalty fees are paid by advertisers. Under current entertainment systems, the fees are well structured. Television broadcasters pay royalty fees to the Copyright owner and then recoup those costs either from an advertising base, or customer subscription fees. Cinema theatres pay royalty fees which are recouped from ticket sales. Video hire store operators pay royalty fees from a portion of the video rental fee charged to the consumer.

Similarly, content theft is not of major concern. While it is possible to make copies from most sources, these copies are analogue copies and are therefore degraded in quality from the original. Furthermore, repeated copies further degrade in quality. This implies that while copies can be made for personal use, the poor quality means that this copy could not be successfully used for re-broadcasting and denying royalty payments to the lawful Copyright owner. While not happy with the of theft of Copyright material for personal use, content owner must live with this loss of income as this theft is difficult to police.

We can consider video streaming to be a new method of content delivery, again it is essential that Copyright owners are paid their due royalty fees – to obtain their return on investment and ensure that content is made available for streaming purposes. Copyright protection becomes a more contentious issue in the digital realm since perfect digital copies can be made with no degradation in playback quality. As such, copies of digital streaming video can be used for large scale piracy more easily and lead to much greater reduced returns to the Copyright owners. This makes many content providers, who are generally not experts in Internet technology, extremely worried against theft and non-payment for services. This user group can only be placated, and therefore making video streaming a viable proposition, by addressing these concerns.

Copyright protection is one of many issues, both technical and non-technical, that must be addressed in order for video streaming to become viable. The issue of Copyright protection itself is also a large one, and can be sub-divided into a series of smaller problems to be addressed:

- One solution to Copyright protection is the use of watermarking, or passive protection, of streaming content. Watermarking does not actively prevent against theft of content but can be used after theft has been discovered to determine the source of that theft.
- Another approach that can be taken is encryption, or active protection, of streaming content. In this case the video content itself is protected through the use of a cipher, meaning that even if the digital stream is stolen, it must still be decrypted before it can be used again. The process of content encryption can also be broken down into two parts, the actual cipher applied to protect the streaming content, and the secure delivery of the cipher key to the consumer for successful decryption and playback of the encrypted stream.
- The major purpose of protecting Copyright is to guarantee a return on investment. This means that network payment schemes must be adopted. Similarly, the payment scheme must be tied to an access control scheme to ensure that paying customers have proper access to the system.

Watermarking and Encryption are not mutually exclusive. Indeed, an ideal solution would likely incorporate both aspects of content protection – active protection to protect against theft in the first place, and passive protection to enable prosecution if encryption fails. This thesis explores the idea of encryption of streaming video and how it might be employed in a real system.

It is possible to construct a video streaming solution that operates with a single central server. However, this requires guaranteed bandwidth throughout the network, from the streaming server to each consumer end station – if throughput is not guaranteed, service delivery will falter. Also, the costs in bandwidth requirements for the server and its associated hardware can become prohibitive. While many Internet streaming applications today utilise this approach, the number of concurrent users are small enough that these issues can be managed.

A more likely scenario for a true video streaming solution is a distributed server environment. In this case multiple servers together provide a unified streaming service. Content is delivered to the customer from the closest streaming server. The distributed server design enforces fewer requirements on network capabilities, only requiring Quality of Service guarantees between each streaming server and their locally based client. This moves QoS requirements to the edge of the network rather than throughout the core. A distributed server design also allows for improved scalability since supporting more concurrent streams is a simple matter of installing an extra streaming server. Further, the design is more fault tolerant since the service is distributed over many platforms rather than providing a single point of failure on a central platform. Finally, overall costs are lessened since each individual streaming server has fewer CPU and bandwidth requirements.

We can take the concept of a distributed system further by considering a multi-platform distributed server design. This design is similar in scope to a distributed server design but allows each of the individual streaming server platforms to be running a different video streaming solution. This allows for one of two scenarios:

Single Service Provider – The service provider will not be forced to choose a particular streaming server platform. This allows them to choose the most suitable streaming platform at the time it is required. As costs change and new products are made available, this does not lock the service provider into purchasing from a single source, allowing competition to lower implementation costs.

Multi-Party Service Provision – Allows a group to provide a single streaming service. A system is made up of many providers, some providing streaming capabilities while others provide content. The multi-platform design allows this concept as it does not enforce choice of a streaming platform onto service providers, allowing competition to flourish. Further, a multi-party service can allow small content owners to reach a wider market than they could otherwise afford to reach if they were required to provide the video distribution service.

Management of a multi-platform system is more complex and requires standard protocols to ease communications between server platforms. While the actual streaming protocol that the servers use can still be unique, the inter-server protocol must be common to allow for delivery of content to servers. The existence of such a protocol removes platform dependence as a requirement for operators of a video streaming service.

Some existing streaming server platforms offer encryption of streaming video as part of their feature set. These servers often apply the encryption in real-time while content is being streamed. This solution is not scalable due to potentially large numbers of concurrent streams needing to be secured. Further, in a distributed server environment, we will wish the content to be installed onto the servers already in encrypted format.

When we consider active protection of content in a distributed streaming server environment, we should also consider the concept of a multi-party distributed server arrangement. If the video streaming service is being provided by multiple parties, there is no reason for trust to exist between a Copyright owner and a streaming server operator. There is potential for theft to occur either deliberately – by the streaming server operator – or accidently – through poor security measure implemented by the streaming server operator.

The platform independence advantages of a multi-platform system offer problems when considering video encryption methods. In order to maintain the same platform independence, it is necessary for the chosen cipher to also be platform independent. As such it becomes a requirement for encryption of streaming video that the cipher be compatible with a wide range of existing and future streaming server products.

In order to support multiple streaming server platforms, the cipher must be designed with an understanding of how video is actually streamed. This becomes a further issue when we consider that some streaming server platforms offer advanced playback functionality such as indexed or high-speed playback modes. The video stream must be encrypted such that it can be streamed from an existing server without prior knowledge of the cipher algorithm, as well as being successfully decrypted in all possible playback modes. This restriction implies that the video content can be on the installed onto the streaming server in encrypted form. In this case, content can be encrypted by the content owner; the encrypted bitstream can then be made freely available for streaming.

There are also more mundane restrictions required by the Copyright owners. These include security of the cipher – the cipher should not be easy to break and therefore lead to retrieval of the plaintext bitstream. Further, an attacker should not be able to retrieve even portions of the original video or audio content contained within the encrypted stream.

These restrictions on the cipher are complex, and not completely met by any existing MPEG-1 cipher algorithms. In this thesis, I present a novel MPEG-1 Partial Cipher which does meet all of the requirements as outlined above:

- **Compatibility with existing streaming servers** – By ensuring that the cipher does not modify the contents of the MPEG-1 System Stream, as well as maintaining key aspects of the Video and Audio Streams, an encrypted MPEG-1 bitstream can be successfully installed on existing streaming server products. The encrypted file appears to be a valid MPEG-1 bitstream for installation purposes. Streaming servers can also provided advanced playback functionality such as indexed and high-speed playback modes. In order to provide these playback modes, servers must partially decode the installed media asset. Indexed playback requires searching for timestamps within the installed bitstream and high-speed playback involves finding and extracting individual frames from the installed bitstream. The cipher as presented deliberately leaves this information in the bitstream in plaintext format, thereby allowing existing streaming server products to provide these adavanced playback modes without having prior knowledge of the cipher algorithm itself.

- **Compatibility with existing client side decoders** – Because the decryption process is applied at a higher level than many existing ciphers – Slice data in the Video Stream and Audio Data in the Audio Stream – the decryption module at the client end is not intimately tied to the decoder. While other ciphers provide an efficient implementation only when the cipher forms part of the decoder, the proposed cipher is efficient even where decryption is applied prior to decoding. This means that cipher implementations are removed from decoder implementations, allowing the developers to select the best available tools for the application.

- **Supports decryption in different playback modes** – Some streaming servers provide advanced playback modes such as indexed and high-speed playback. In these modes, the bitstream arriving at the client is necessarily different to a simple streaming server. In the case of indexed playback, the delivered stream commences partway through the original bitstream.

When considering high-speed playback, a variety of individual frames are delivered as the bitstream. If the decryption module is to support these playback modes, it must be able to resynchronise its internal state such that the correct plaintext is produced in all playback modes. The presented cipher uses information within the bitstream to determine resynchronisation values and as such can correctly decrypt a bitstream delivered in any of these playback modes.

- **Is secure** – The cipher works in two stages, the first is the selection of bytes from the plaintext bitstream for encryption and the second is the actual encryption of those bytes. The encryption process is based on the SEAL Stream Cipher. This cipher has been openly published and is currently considered to be secure with no known potential weaknesses. Another advantage of the SEAL Cipher is its speed of execution and its ability to allow simple resynchronisation.
- **Ensures the validity of the encrypted bitstream** – The algorithm as designed ensures against the accidental creation of false headers, resulting in an encrypted bitstream that cannot be successfully parsed down to the Slice Layer. This ensures that streaming server products will successfully stream the contents as well as ensuring that the same bytes are selected for decryption at the client end.

This new cipher has been successfully tested. Initial tests prove that the plaintext bitstream is only recoverable with the correct key, and the usage of an incorrect key does not lead to a recoverable plaintext bitstream. Other tests show the low CPU requirements, typically less than 10% of the decoding requirements, of the cipher. These results are valid for decryption without decoding of the bitstream, meaning that there is no performance penalty for not incorporating the cipher with the MPEG-1 decoder. This test also proved that real-time decryption and playback of an encrypted MPEG-1 bitstream was possible.

Further, other tests showed that the encrypted bitstream was accepted for installation onto a range of streaming server platforms. These tests also proved that the encrypted bitstream was able to be streamed in all playback modes supported by those servers. Furthermore, client playback test applications proved that the encrypted bitstream could be successfully decrypted and decoded. This was verified in all playback modes – normal, indexed and high-speed. The proposed MPEG-1 Cipher was compatible with a range of streaming server products in a variety of different playback modes, this verified the design aim of platform independence of the cipher.

The concept of the MPEG-1 Cipher as designed is also amenable to MPEG-2 bitstreams and their subsequent streaming. This is true since the format of an MPEG-2 bitstream is similar to that of an MPEG-1 bitstream. Indeed, being backwards compatible, an MPEG-1 bitstream can be successfully decoded by an MPEG-2 decoder. Streaming server platforms that support streaming of MPEG-2 bitstreams do so in a similar fashion to streaming of MPEG-1 bitstreams. Having proved the viability of the MPEG-1 Cipher, implementation of an MPEG-2 Cipher is a development rather than research problem.

Video encryption algorithms are a relatively new field of study and a range of different algorithms have been proposed. While some of these ciphers have proven insecure, many are not and adequately perform the task of protection of content. However, even though a handful of existing ciphers are compatible with streaming video, these ciphers were not designed with streaming video in mind, but rather with the idea of protection of video content.

Similarly, some commercial streaming server platforms implement a proprietary form of encryption. Unfortunately, these ciphers are often applied in real-time and are tied closely to the streaming server implementation. This restricts freedom of choice when upgrading systems to support more concurrent streams. An ideal solution is a cipher that is platform independent of both streaming server and client end platforms.

In this thesis I present a novel MPEG-1 Partial Encryption Algorithm that has been designed specifically to secure streaming video. I acknowledge that video encryption is not the only factor in producing a viable video streaming solution, however it – and Copyright protection – is one of the important issues that must be addressed. It is my hope that this work will serve as a baseline for further work in this field. Potentially aiding in the development of an improved cipher that also meets the requirements for encryption of streaming video. Alternatively, to use as a cipher for development of suitable key management and access control techniques in a complete video streaming solution.

Appendix A

The MPEG-1 Bitstream Format

In this Appendix I will provide an overview of the MPEG-1 Digital Video Compression Standard. I will present the history and the motives behind the MPEG group of standards. MPEG will be discussed because it is the predominant video compression standard in the world today in the area of high quality digital video. I begin by discussing the history of MPEG and look at why it has become the dominant video compression standard in the world today. I then take a detailed look at the MPEG-1 Compression Standard, the first standard released by the MPEG group. The information presented in this Appendix is a summarised representation of that contained in (Anderson, 1990; Haskell et al., 1997; ISO, 1996a; ISO, 1996b; ISO, 1996c; ISO, 1996d; LeGall, 1991; Mitchell et al., 1996; Noll, 1997; Pan, 1993; Pan, 1995; Puri, 1994; Pereira, 1996). Figures presented have been originally taken from (Mitchell et al., 1996) and modified to better demonstrate their intent.

A.1 History of MPEG-1

The MPEG-1 Standardisation effort commenced in 1988 in an effort to create a single standard that encompassed high quality compression of digital video. Initial meetings elected to devise an algorithm for compression of digital video and accompanying audio at rates of approximately 1.5 Mb/s, this being the data bit rate supported by single speed CD-ROM drives. In January 1988, the MPEG(Motion Picture Expert Group) was formed as a sub group of Working Group 8 of ISO(International Standards Organisation), this group was numbered as Working Group 11. The first meeting of the MPEG Group was in May 1988. At this meeting, a consensus was reached not only to target compression of video to about 1.5Mb/s, but also to devise an asymmetric system: one where the encoder had more complex requirements than the decoder.

Many representatives from both research and commercial backgrounds contributed to the work on standardising MPEG-1, with many competing ideas being compared and debated. Work on the standard progressed until November 1991, where a committee draft was recommended for ballot on becoming an International Standard. The standard was ratified in March 1992 and the standard was named ISO 11172 (MPEG-1). The standard was finally published in 1993 in three parts: Part one of the standard covered the MPEG system stream and how multiple media streams were multiplexed into a single bit stream. The second part of the standard covered MPEG-1 Video Compression whilst part three covered MPEG-1 Audio Compression. Other parts of the standard would emerge later to cover details such as conformance testing and software simulation.

Before work on MPEG-1 was concluded, the MPEG group commenced work on MPEG-2. MPEG-2 concerned with compression of digital video at higher quality for purposes such as broadcast television. Not long after, two more proposals were put forward, MPEG-3, which looked at digital video compression at HDTV quality, and MPEG-4, which is aimed at lower bit rates. It was soon realised that MPEG-2 would meet the MPEG-3 requirements and thus MPEG-3 was dropped.

While MPEG-2 is of great interest, prohibitive costs of network bandwidth mean that in a networked solution, MPEG-2 will incur greater costs than an MPEG-1 or MPEG-4 video streaming solution. This situation will likely improve in the near future, but even then only in the local Intranet. The competition for bandwidth in the Internet is likely to mean that the prospects of providing MPEG-2 quality video on the Internet in the near future are unlikely.

MPEG-4 compressed video is the most likely near term solution for streaming video, as its bandwidth requirements are low. However, the quality of video provided by MPEG-4 compression, whilst good, is not good enough to be considered for the provision of digital media services for entertainment purposes. In this case we must consider MPEG-1 and MPEG-2 compressed video. The higher bandwidth requirements of MPEG-2 mean that MPEG-1 is likely to become more commonplace prior to MPEG-2 streaming video.

A.2 MPEG-1

MPEG-1 was the first standard to come out of the MPEG working group. It was initially targeted at compressed video and audio signals at a bit rate of 1.5Mb/s. This chosen bit rate was not accidental, it being the data rate achieved from a CD running at normal speed. Work on MPEG-1 began in 1988 and the international standard, IS 11172-2, was published in 1993. The MPEG-1 standard was published in three parts, the first of these deals with systems aspects, the second with video compression and the third with audio compression. Together these documents make up a standard which defines an MPEG-1 bit stream as well as the exact requirements for building an MPEG decoder.

The specifications for an MPEG-1 encoder are deliberately omitted from the standard. This was done in an attempt to future proof the standard by allowing improvements in encoder design and implementation as long as the encoder produced a standard MPEG-1 bit stream. In turn, this also meant that as long as the chosen decoder was compliant with the MPEG-1 standard, then any MPEG-1 bit stream could be decoded and played back, regardless of which encoder was used to compress the original video source.

Another important point is that MPEG-1 and MPEG in general is a highly asymmetric system. The encoder is a complex machine that must make decisions on motion and optimal motion vectors. They must also control the bit rate and buffers, locate repeated macroblocks as well as vary all of these parameters dynamically to maximise video quality for a given rate. On the other hand, MPEG decoders need merely to decode the bit stream and display the results on the screen. This approach decreases the cost of implementation in two ways. Firstly, since encoding is usually performed only once while decoding is performed many times, the computationally expensive task of encoding is

performed a minimal number of times. Secondly, the cost of decoding machines is lessened by lower processor requirements leading to a cheaper solution for the vast majority of users, expensive machinery need only be purchased by those encoding video.

The MPEG-1 standard was later extended to include two other parts. The fourth is a document regarding compliance testing, whilst the fifth refers to a software reference model for MPEG-1. While originally designed for bit rates of about 1.5Mb/s, it has been shown that acceptible quality can be achieved at rates of about 1Mb/s while high quality video can be achieved at higher rates of about 6Mb/s. Experiments of visual quality perception carried out at Monash University indicate that bit rates of 2Mb/s result in decoded video at a perceived quality better than VHS.

As mentioned earlier, the MPEG-1 standard consists of five parts:

- The first of these deals with the system aspects of MPEG-1. This part deals with defining a systems layer that provides an envelope for the compression layers which actually store the compressed data. The systems layer concerns itself with two tasks, the first of these is to multiplex compressed bit streams into a single system stream, while the second is to provide timing and buffering information to the decoder to assist in the decoding process.

- The second part of the MPEG-1 standard deals with video compression and defines a bit stream that holds compressed video data. The document defines different layers in the MPEG-1 video stream that contain information on the video sequence being represented. The document does not define how the data is to be compressed, but does define how to decode the data to achieve a visual representation. The absence of encoding standards allows for new techniques to be employed in the creation of MPEG-1 bit streams, as long as the ensuing stream is compatible with the standard.

- The third part of the standard deals with audio compression and defines the bit stream for compressed audio data. This document deals with audio perception models of the human ear and techniques for lossy audio compression with no perceivable differences in the resulting audio. As for video compression, only the bit stream and decoder properties are defined.

- The fourth part of the standard deals with compliance testing and specifies how bit streams and MPEG-1 decoders can be tested to ensure that they comply with the standard.

- The fifth part of the standard is the software reference model and contains software for both an encoder and an arithmetically correct decoder. This allows developers to produce compliant bit streams as well as comparing output from decoders with output from the reference decoder.

MPEG-1 is a standard that will be with us for some time to come. While the quality of MPEG-1 video is inferior to that provided by MPEG-2 at higher bit rates, at lower bit rates MPEG-1 provides better quality video. Also, the lower bit rates provided by MPEG-1 will mean that its use will not be simply discarded. While bit rates available on computer hardware are increasing quickly and MPEG-2 decoders can now be implemented in software on todays machines, the issue of bandwidth over telecommunications links become an issue when networked video is considered. The higher quality provided by MPEG-2 will become more desirable when bandwidth costs decrease even more but at the moment, MPEG-1 is ideally suited to delivery over the network.

A.2.1 MPEG-1 System Stream

The MPEG-1 System Stream is a binary stream which defines the MPEG-1 file itself. When a movie is stored on disc in MPEG-1 format it is stored as an MPEG-1 System Stream. When a movie is passed to an MPEG decoder to be decoded and displayed, it is passed as an MPEG-1 System Stream. Obviously, when a movie is compressed, it must be stored in some specified format so that a decoder can logically reconstruct the sequence to the final output stage.

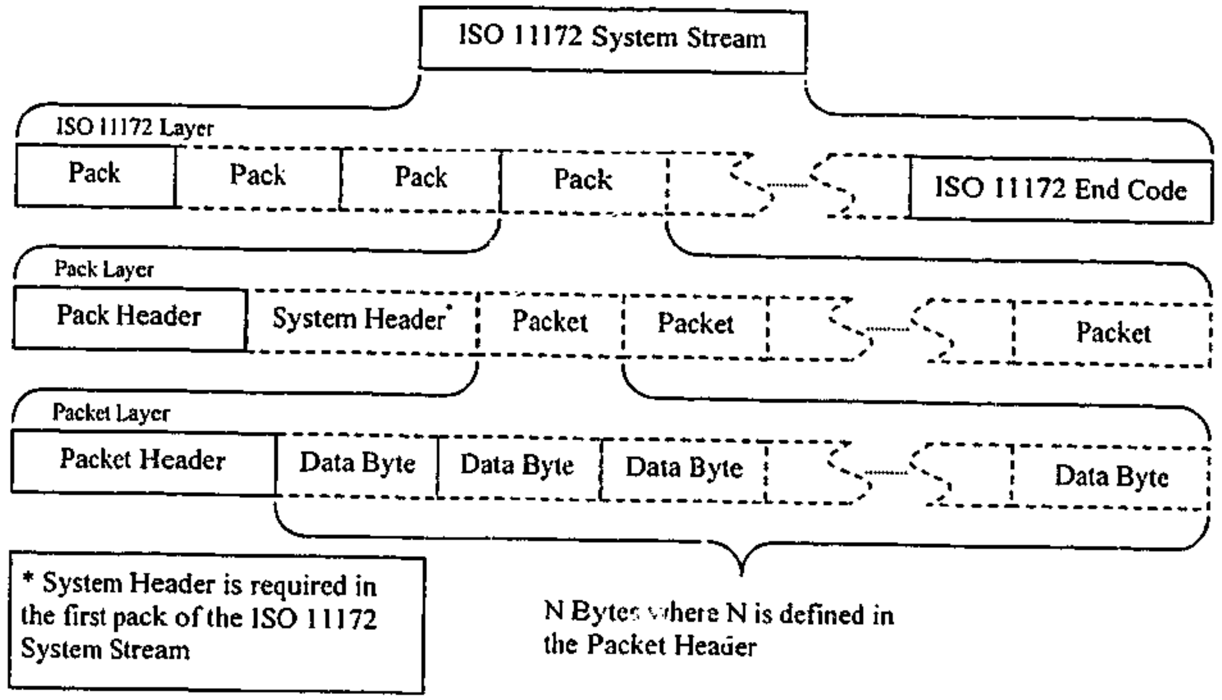


Figure A-1: MPEG-1 System Stream

In MPEG-1, the information is compressed and stored in a variety of layers, the MPEG-1 System Stream forms the top three layers. Each layer further down the chart contains more specific information than the layer above it. At lower levels, raw video is compressed to form a binary bit-stream called the MPEG-1 Video Stream and raw audio is similarly compressed to form a binary bit-stream called the MPEG-1 Audio Stream. These streams, in their own right, contain the information required to reproduce a certain part of the original audio/video asset. The function of the MPEG-1 System Stream is to combine one or more of these streams into a single binary bit-stream. In effect, we are multiplexing the combined information streams into a single data channel. These layers can be seen in Figure A-1.

A.2.1.1 ISO 11172 Layer

The ISO 11172 layer defines the entire MPEG-1 System Stream. The bit-stream consists of one or more packs followed by the ISO 11172 end code, where each single pack is an instance of the pack layer. Note that there must be at least one pack in the MPEG-1 System Stream. The ISO 11172 end code is the unique byte aligned 32 bit start code 0x000001B9). This end code and other unique 32 bit start codes specified in the MPEG-1 System Stream can be found in Table A-1.

Start Code Description	Hexadecimal Value
ISO 11172 End Code	00 00 01 b9
Pack Start Code	00 00 01 ba
System Header Start Code	00 00 01 bb
Reserved Stream	00 00 01 bc
Private Stream 1	00 00 01 bd
Padding Stream	00 00 01 be
Private Stream 2	00 00 01 bf
Audio Stream 0	00 00 01 c0
...	...
Audio Stream 31	00 00 01 df
Video Stream 0	00 00 01 e0
...	...
Video Stream 15	00 00 01 ef
Reserved Stream 0	00 00 01 f0
...	...
Reserved Stream 15	00 00 01 ff

Table A-1 MPEG-1 System Stream Unique 32 Bit Byte Aligned Start Codes

A.2.1.2 Pack Layer

The pack layer defines a binary bit-stream that makes up a single pack in the MPEG-1 System Stream. The pack layer consists of a pack header, optionally followed by a system header and zero or more packets, where each single packet is an instance of the packet layer.

While the presence of the system header is optional in a pack, it is required for the first pack in the MPEG-1 System Stream. The reason for this is that the system header contains the same information for the entire MPEG-1 System Stream and therefore does not need to be repeated within the stream again. It is also legal for a pack to contain only a pack header and no packets.

The pack header, as formally specified in Figure A-2, is always twelve bytes long and begins with the unique byte aligned 32 bit start code 0x000001BA. The next five bytes record the system clock reference (SCR) which forms a time stamp that determines when this packet should be passed from the buffer into the decoder. The SCR is a 33 bit value that is broken up with fixed marker bits to ensure against accidental creation of a start code. The final three bytes of the pack header contain the mux_rate. The mux_rate is encoded as a 22 bit value surrounded by fixed marker bits and measures the rate at which bytes arrive at the decoder in units of 50 bytes per second.

```
pack header()
{
     pack_start_code(32);                /* 0x000001ba               */
     ‘0010’;                             /* 4-bit fixed pattern      */
     system_clock_reference(3);          /* Bits 32-30 of SCR        */
     marker_bit(1);                      /* ‘1’                      */
     system_clock_reference(15);         /* Bits 29-15 of SCR        */
     marker_bit(1);                      /* ‘1’                      */
     system_clock_reference(15);         /* Bits 14-0 of SCR         */
     marker_bit(1);                      /* ‘1’                      */
     marker_bit(1);                      /* ‘1’                      */
     mux_rate(22);                       /* mux rate                 */
     marker_bit(1);                      /* ‘1’                      */
}
```

Figure A-2: MPEG-1 Pack Header Definition

The system header contains further information about the MPEG-1 System Stream and is formally specified in Figure A-3. The length of the system header is variable but has a minimum length of twelve bytes. The system header begins with the unique byte aligned code 32 bit start code 0x000001BB. Following the start code is a 16 bit value containing the number of bytes remaining in the header after processing the header length value. The value stored in the header length field will be the entire length of the system header minus 6 for the 32 bit start code and 16 bit header length fields. The next field contains a 22 bit value called rate_bound, this value is bracketed by fixed marker bits. This forms an upper bound for the mux_rate defined in the pack header and must be equal to or greater than the value from the pack header. The next six bits set an upper bound on the number of audio streams in the MPEG-1 System Stream. The next four bits form flags that signify in turn: fixed or variable bit rate, constrained or unconstrained bitstream, lock SCR to audio or not, and lock SCR to video or not. All these flags have an effect on how the decoder behaves but are mainly informative. These flags are then followed by a fixed marker bit and a five bit value setting an upper bound on the number of video streams in the MPEG-1 System Stream. Following this value is a reserved byte which is set to 0xFF. Here ends the fixed twelve byte length of the system header.

The system header can be extended with further information on each stream that is to be multiplexed within the system stream. If the next bit in the bitstream is a 1, then the system header continues, if it is a 0, then the system header ends. If the system header continues then the next 8 bits in the bits stream signify the stream identifier, a value to identify that the following information concerns the stream identified by this value. This is followed by two fixed marker bits. The next bit, STD_buffer_bound_scale, signifies a unit choice for the final parameter, STD_buffer_size_bound, which forms a thirteen bit value stating the desired buffer size for this stream. If the scale parameter is 0, the buffer size is measured in units of 128 bytes, otherwise it is measured in units of 1024 bytes. Generally 0 is used for audio streams and 1 is used for video streams. This buffer information segment is exactly three bytes long and can be repeated for each of the valid 53 streams in the MPEG-1 System Stream. When the next bit is no longer a 1, then we have reached the end of the system header and have found either the beginning of an individual packet or the beginning of the next pack.

```
system header()
{
     system_header_start_code(32);     /* 0x000001bb              */
     header_length(16);                /* num. of bytes in header*/
     marker_bit(1);                    /* ‘1’                     */
     rate_bound(22);                   /* Upper bound on mux_rate*/
     marker_bit(1);                    /* ‘1’                     */
     audio_bound(6);                   /* Upper bound on number   */
                                       /* of audio streams        */
     fixed_flag(1);                    /* Fixed Bit Rate?         */
     CSPS_flag(1);                     /* Constrained Bit Stream?*/
     system_audio_lock_flag(1);        /* SCR locked to audio?    */
     system_video_lock_flag(1);        /* SCR locked to video?    */
     marker_bit(1);                    /* ‘1’                     */
     video_bound(5);                   /* Upper bound on number   */
                                       /* of video streams        */
     reserved_byte(8);                 /* Always ‘1111 1111’      */
     while (nextbit == ‘1’)            /* Not a start code        */
     {                                 /* Stream buffer info.     */
          stream_id(8);                /* Stream Identifier       */
          ‘11’;                        /* 2-bit fixed pattern     */
          STD_buffer_bound_scale(1);   /* Buffer Scale factor     */
          STD_buffer_size_bound(13);   /* Buffer size upper bound*/
     }
}
```

Figure A-3: MPEG-1 System Header Definition

A.2.1.3 Packet Layer

The packet layer defines a binary bit-stream that makes up a single packet in the pack layer and forms the lowest level layer of the MPEG-1 System Stream. The packet layer consists of a packet header followed by a byte stream that makes up the packet information. The packet header contains information about the bytes included in the packet and therefore also which sub-stream these bytes belong to. The decoder must process the packet header, extract the encapsulated bytestream and forward it on to the appropriate decoder for that stream.

The packet header, as formally specified in Figure A-4, is of indeterminate length but can be described as follows. The packet header begins with a unique byte aligned 32 bit start code in the range 0x000001BC through 0x000001FF. The last byte of the start code identifies which stream the packet belongs to and determines where the byte data of the packet should be sent. The next sixteen bits identify the size of the packet in bytes beginning from immediately after these 16 bits. The sixteen bits allow for a maximum size of 65536 bytes minus the remainder of the packet header as the maximum number of bytes stored in each packet.

Following the packet length field, there is an anomaly in the packet header definition in which the following information is found in all packet headers except that with stream ID 0xBF or private data stream 2. In the case of all other stream identifiers we can have an arbitrary number of stuffing bytes, including none. These stuffing bytes are always 0xFF and are used to pad out the bitstream if there is not enough data to fill the required bit rate. Following the stuffing bytes there are a number of other optional fields.

```
packet header()
{
    packet_start_code_prefix(24);   /* 0x000001               */
    stream_id(8);                   /* Stream Identifier      */
    packet_length(16);              /* Num. of bytes in packet*/
    if (stream_id != 0xbf)          /* If not Private Stream 2*/
    {
        while (nextbits(8) == 0xff) /* More stuffing bytes    */
        {
            stuffing_byte(8);       /* Always '1111 1111'     */
        }
        if (nextbits(2) == '01')    /* Buffer Size Information*/
        {
            '01';                   /* 2-bit fixed pattern    */
            STD_buffer_scale(1);    /* Buffer scale           */
            STD_buffer_size(13);    /* Buffer size            */
        }
        if (nextbits(4) == '0010')  /* Only PTS is present    */
        {
            '0010';                 /* 4-bit fixed pattern    */
            present_time_stamp(3);  /* Bits 32-30 of PTS      */
            marker_bit(1);          /* '1'                    */
            present_time_stamp(15); /* Bits 29-15 of PTS      */
            marker_bit(1);          /* '1'                    */
            present_time_stamp(15); /* Bits 14-0 of PTS       */
            marker_bit(1);          /* '1'                    */
        }
        elseif (nextbits(4) == 0011) /* PTS and DTS is present */
        {
            '0011';                 /* 4-bit fixed pattern    */
            present_time_stamp(3);  /* Bits 32-30 of PTS      */
            marker_bit(1);          /* '1'                    */
            present_time_stamp(15); /* Bits 29-15 of PTS      */
            marker_bit(1);          /* '1'                    */
            present_time_stamp(15); /* Bits 14-0 of PTS       */
            marker_bit(1);          /* '1'                    */
            '0001';                 /* 4-bit fixed pattern    */
            decode_time_stamp(3);   /* Bits 32-30 of DTS      */
            marker_bit(1);          /* '1'                    */
            decode_time_stamp(15);  /* Bits 29-15 of DTS      */
            marker_bit(1);          /* '1'                    */
            decode_time_stamp(15);  /* Bits 14-0 of DTS       */
            marker_bit(1);          /* '1'                    */
        }
        else                        /* Neither PTS nor DTS    */
        {
            '0000 1111';            /* 8-bit fixed pattern    */
        }
    }
}
```

Figure A-4: MPEG-1 Packet Header Definition

The first of the optional fields indicate the buffer size in the decoder for this stream. If this field is present, the next two bytes in the header are encoded as two fixed bits of value '01' followed by a 1 bit value signifying the buffer scale and a thirteen bit value signifying the buffer size. These values have the same meaning as the stream information contained in the system header.

Following the buffer size is the timestamps field of the packet header. These fields contain information of the timestamps allocated to this particular packet. Again, the format of this field is variable and it can contain either both the presentation time stamp (or PTS: the time at which the packet is presented to the viewer) and the decoding time stamp (or DTS: the time at which the packet is presented to the decoder), the PTS alone or neither. If only the PTS is present, then it is encoded in five bytes with the first four bits containing the fixed code '0010' followed by the 33 bit PTS encoded as 36 bits in the same format as the SCR in the pack header. If both the PTS and DTS are present, then it is encoded as ten bytes with the first four bits containing the fixed code '0011' followed by the 33 bit PTS encoded in the same format as the SCR in the pack header. This is then followed by the four bit fixed code '0001' and the 33 bit DTS encoded in the same format as the SCR in the pack header. Finally, if no time stamps are present, then it is encoded as a single byte with the fixed byte value 0x0F.

Following the optional time stamps field, the rest of the packet is filled with data bytes to be de-multiplexed into their respective MPEG-1 Video Streams or MPEG-1 Audio Streams and presented to the appropriate decoder.

A.2.2 MPEG-1 Video Compression

The MPEG-1 Video Compression standard is defined in part two of the MPEG-1 standard. This document is concerned with standardising the MPEG-1 Video bit stream as well as defining a decoder that can reconstruct the video sequence from this bit stream. We will be looking mainly at the format of the bit stream itself to help gain a clear picture of not only the binary format of the bit stream but also the purpose of each individual layer in the MPEG-1 Video Stream.

A video sequence is made up of a series of frames where each frame can be considered as a single two dimensional picture. Standard image compression techniques work by removing redundancy found in the image itself. If this idea was applied to moving picture sequences, we would have a technique whereby each individual frame was compressed to the smallest size possible. Whilst some good compression can be obtained in this fashion, final bit rates are an order of magnitude greater than those that can be achieved using MPEG-1. Compressing each individual frame removes spatial redundancy in the images being compressed but not temporal redundancy. Temporal redundancy refers to similarity between two different frames in the same video sequence. By removing temporal redundancy during video compression, we can improve compression rates by an order of magnitude. This is possible since in most video sequences adjacent frames have a high degree of correlation.

Individual frames in MPEG-1 are usually compressed into what are called I, P or B frames. I-Frames are encoded without reference to other frames and are similar to JPEG compression. I-Frames serve as a reference point where decoding can begin on a random seek into an MPEG Video Stream as well as markers to remove any error due to predictive coding. P-Frames look for similarity between the encoded frame and a previously encoded I or P-Frame. They use predictive coding to remove any temporal redundancy between these two frames and the second frame is encoded as the necessary changes to make to a previous frame. B-Frames are similar to P-Frames except that they use both backward and forward prediction, encoding a frame as the differences between both a preceding and an upcoming I or P-Frame. B-Frames offer the greatest compression. A fourth type of frame is the D-Frame, this offers the greatest compression rates with low picture quality. A D-Frame is encoded independantly of other frames but less coefficients are used when encoding DCT blocks.

Frames within the MPEG-1 Video Stream are stored in the decoding order which does not necessarily match the presentation order. A B-Frame must have both frames it depends on encoded before it is encoded, this means that the succeeding I or P-Frame is encoded in the bit stream before the B-Frame. An example is shown in Figure A-5.

The heart of MPEG-1 compression is the DCT (Discrete Cosine Transform) transform, a 16x16 block of pixels is broken up into four 8x8 blocks of luminance pixels and two 8x8 blocks of chrominance pixels. The chrominance pixels are samples at half the resolution of the luminance pixels. These blocks of pixels are traversed in a zig-zag pattern and their values are converted to the frequency domain using a DCT transform. The DCT coefficients are then encoded using a huffman code to ensure maximal compression.

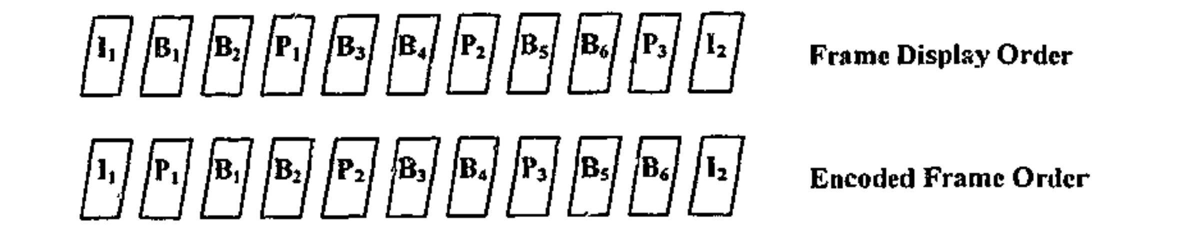

I1 B1 B2 P1 B3 B4 P2 B5 B6 P3 I2
Frame Display Order
I1 P1 B1 B2 P2 B3 B4 P3 B5 B6 I2
Encoded Frame Order

Figure A-5: MPEG-1 Encoded Frame Order

While DCT compression is relatively straight forward, determination of motion vectors and maintaining good quality video at a specified bit rate are difficult problems that must be solved by MPEG-1 encoders. Motion vectors are used when encoding P and B Frames. High compression is achieved by locating a similar block in a preceeding frame and encoding it as a motion vector – the current macroblock can be copied from a previously decoded frame. Research continues to be performed on determining the best method of calculating motion vectors.

A.2.3 MPEG-1 Video Stream

The MPEG-1 Video Stream is a binary stream which defines a single video stream within an MPEG-1 System Stream. The MPEG-1 Video Stream is multiplexed along with other Video Streams and Audio Streams within the MPEG-1 System Stream. The MPEG-1 Video Stream on its own provides information necessary to decode and display a compressed video stream. As in the case of the System Stream, the MPEG-1 Video Stream is expressed as a number of layers, only in the case of the Video Stream there are six layers. At lower levels, the layers represent individual macro blocks within a single frame of the compressed video, at higher levels, these layers represent concepts such a single frame, a group of frames or the entire video sequence. The format of the MPEG-1 Video Stream is shown in Figure A-6.

Like the MPEG-1 System Stream, the MPEG-1 Video Stream also has a series of unique 32 bit start codes which are specified in Table A-2.

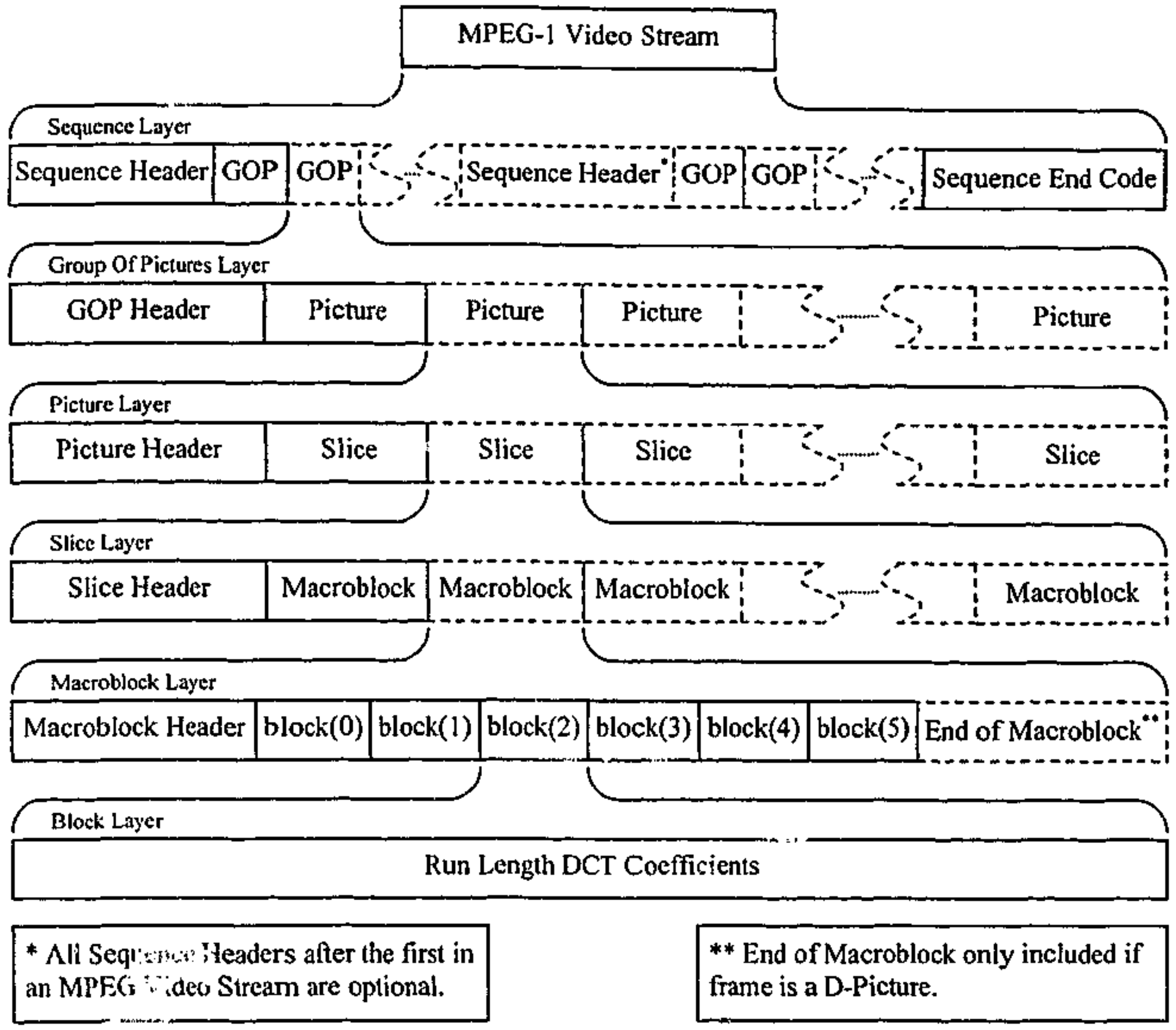


Figure A-6: MPEG-1 Video Stream

A.2.3.1 Sequence Layer

The sequence layer defines the entire MPEG-1 Video Stream. The bitstream consists of a sequence header, followed by at least one Group of Pictures (or GOP), where each individual GOP is an instance of the GOP layer. There can be an arbitrary number of GOPs before the sequence layer is terminated with the sequence end code. Also, it is possible to insert additional sequence headers into the sequence layer as long as a GOP both immediately precedes and succeeds it. The purpose of inserting additional sequence headers may assist in the provision of fast play or random seek commands.

The sequence header, as formally specified in Figure A-7, can be of variable length but always begins with the unique byte aligned 32 bit start code 0x000001B3. The next twenty-four bits are used to encode the picture width and height of the video stream with twelve bits used for each dimension. Restrictions in these values are that neither is allowed to be zero and that the vertical picture size must be even. The next four bits signify the pixel aspect ratio of the video stream and the value is used as a reference into a table of predefined aspect ratios. As we are only describing the bitstream format and not building a decoder, the actual values are unimportant. The following four bits also form a reference into a table to signify the frame rate of the video sequence.

Start Code Description	Hexadecimal Value
Picture Start Code	00 00 01 00
Slice 1 Start Code	00 00 01 01
...	...
Slice 175 Start Code	00 00 01 af
Reserved	00 00 01 b0
Reserved	00 00 01 b1
User Data Start Code	00 00 01 b2
Sequence Header Code	00 00 01 b3
Sequence Error Code	00 00 01 b4
Extension Start Code	00 00 01 b5
Reserved	00 00 01 b6
Sequence End Code	00 00 01 b7
Group Start Code	00 00 01 b8

Table A-2 MPEG Video Stream Unique 32 Bit Byte Aligned Start Codes

```
sequence header()
{
     sequence_header_code(32);        /* 0x000001b3              */
     horizontal_size(12);             /* Picture width           */
     vertical_size(12);               /* Picture height          */
     pel_aspect_ratio(4);             /* Sample aspect ratio     */
     picture_rate(4);                 /* Frame rate              */
     bit_rate(18);                    /* Bit rate                */
     marker_bit(1);                   /* '1'                     */
     vbv_buffer_size(10);             /* Decoder buffer size,    */
                                      /* lower bound             */
     constrained_parameters_flag(1);  /* Parameters Constrained?*/
     load_intra_quantiser_matrix(1);  /* Intra Quantiser matrix?*/
     if (load_intra_quantiser_matrix) /* Matrix exists           */
     {
          intra_quantiser_matrix(512);/* 64 x 8-bit values       */
     }
     load_nintra_quantiser_matrix(1); /* NonIntra Quantiser mat?*/
     if (load_nintra_quantiser_matrix)/* Matrix Exists           */
     {
          nintra_quantiser_matrix(512);/*64 x 8-bit values       */
     }
     if (nextbits(32) == 0x000001b5)  /* If extension start code*/
     {
          extension_start_code(32);   /* 0x000001b5              */
          while (nextbits(24) != 0x000001)
          {
               extension_data(8);     /* Byte of extension data */
          }
     }
     if (nextbits(32) == 0x000001b2)  /* If user data start code*/
     {
          user_data_start_code(32);   /* 0x000001b2              */
          while (nextbits(24) != 0x000001)
          {
               user_data(8);          /* Byte of user data       */
          }
     }
}
```

Figure A-7: MPEG-1 Video Sequence Header Definition

The next eighteen bits in the bitstream signify the bit rate of the compressed video stream in units of 400 bits/s, a maximal value of 0x3FFFF signify a variable bit rate. The bit rate is followed by a fixed marker bit and a ten bit parameter to signify the lower bound for the decoder buffer size in units of 2048 bytes. The buffer size is followed by a flag to indicate whether the parameters are constrained to certain maximum values or not, again, these values are not important.

The next section of the sequence header concerns the intra quantizer matrix and the non-intra quantizer matrix. This is encoded as a single flag to determine if the intra quantizer matrix exists. If the matrix exists, the next 512 bits encode the 64 eight bit values for the matrix. We then encode a single flag to determine if the non intra quantizer matrix exists, if so, this matrix is represented in the next 512 bits as the 64 eight bit values to fill the matrix.

Finally the sequence header terminates with the addition of extension data or user data. We look for the next start code, if it forms the extension start code 0x000001B5, then there exists an arbitrary number of bytes which form extension data until the next start code is reached. We then look for the user start code of 0x000001B2. If this exists, then there are an arbitrary number of bytes which form user data until the next start code is reached. It should be noted that extension and user data is optional and need not be encoded into the bitstream.

Once a sequence header is processed, it is immediately followed by a GOP which must also be processed. At the end of the GOP we can have either another GOP, another sequence header or the sequence end code. The next MPEG start code needs to be checked to determine what follows and how it should be processed.

The sequence layer is terminated by the sequence end code which is the unique byte aligned 32 bit start code 0x000001B7.

A.2.3.2 Group Of Pictures Layer

The Group Of Pictures Layer (or GOP layer) defines a binary bit stream that makes up a single GOP in the MPEG-1 Video Stream. The GOP layer consists of a GOP header followed by one or more pictures, where each single picture signifies an individual frame and is an instance of the picture layer. Note that the definition requires at least one picture be present within the GOP.

The GOP header, as formally specified in Figure A-8, is of variable length and begins with the unique byte aligned 32 bit start code 0x000001B8. The next 25 bits in the bitstream encode the time code for the first picture in this group of pictures. The time code is the same format as those used by video recorders. The twenty five bits consists of a drop frame flag which is 1 if the frame rate is 29.97 Hz, followed by the hours into the video stream encoded as five bits, the minutes encoded as six bits, one marker bit, the seconds encoded as six bits and the picture number within the second encoded as six bits. Following the time stamp for theGOP are two flags, the first indicates whether the GOP is closed or open, an open GOP means that some pictures in the GOP refer to pictures located in other GOPs, a closed GOP means that the GOP is self-contained. The second indicates if the GOP has been broken and that the original sequence of the GOP has been modified by editing. These fields including the header make up fifty nine bits, five bits of zero padding are required to byte align the end of the GOP header for the next start code.

```
group of pictures header()
{
     group_start_code(32);                /* 0x000001b8              */
     time_code(25);                       /* SMPTE Time Code         */
     closed_gop(1);                       /* Is GOP closed?          */
     broken_link(1);                      /* Is GOP broken?          */
     if (nextbits(32) == 0x000001b5)      /* If extension start code*/
     {
          extension_start_code(32);       /* 0x000001b5              */
          while (nextbits(24) != 0x000001)
          {
               extension_data(8);         /* Byte of extension data */
          }
     }
                                          /* Byte Align Bit Stream  */
     if (nextbits(32) == 0x000001b2)      /* If user data start code*/
     {
          user_data_start_code(32);       /* 0x000001b2              */
          while (nextbits(24) != 0x000001)
          {
               user_data(8);              /* Byte of user data       */
          }
     }
}
```

Figure A-8: MPEG-1 Group Of Pictures Header Definition

As for the sequence header, the GOP header has the option of extension or user data appended to the end of the GOP header. The exact same format for this optional data is used.

A.2.3.3 Picture Layer

The picture layer defines a binary bit stream that makes up a single picture in the GOP layer. The picture layer consists of a picture header followed by one or more slices, where each slice signifies a section of visual image that makes up the picture. Note that the definition requires at least one slice to be present within the picture, this is required since at least one slice is needed to encode the picture itself.

The picture header, as formally specified in Figure A-9, is of variable length and begins with the unique byte aligned 32 bit start code 0x00000100. The next ten bits in the bitstream form the temporal reference of the frame in question. This signifies the position of the frame within the group of pictures to help determine display order. The next three bits form the picture coding type and the values are used as a lookup into a table to determine whether the picture is an I-Frame, P-Frame, etc. As mentioned previously, as we are only describing the bitstream format and not building a decoder, the actual values are unimportant. The next sixteen bits form the vbv buffer delay. This value defines the number of bits that must be in the input buffer before the decoder begins decoding this picture.

The next part of the picture header is dependent on the picture coding type defined earlier in the header. If the picture is a P-Frame or a B-Frame, then the header encodes the forward motion

vector scaling information. This is encoded as four bits, 1 bit signifying 1 pixel and three bits signifying the vector range. If the picture is a B-Frame, we also need to encode the backward motion vector scaling information. This is also encoded as four bits with the same format. Note that if the picture is neither a P-Frame nor a B-Frame, then none of these bits are contained within the header.

```
picture header()
{
     picture_start_code(32);              /* 0x00000100              */
     temporal_reference(10);              /* Picture count MOD 1024 */
     picture_coding_type(3);              /* Picture type            */
     vbv_delay(16);                       /* VBV Buffer delay        */
     if (picture_coding_type == 2 or 3)/*P or B-type picture     */
     {
          full_pel_forward_vector(1); /* Full or half pel       */
          forward_f_code(3);              /* For. motion vect. range*/
     }
     if (picture_coding_type == 3)        /* B-type picture          */
     {
          full_pel_backward_vector(1);/* Full or half pel       */
          backward_f_code(3);             /* Bwd. motion vect. range*/
     }
     while (nextbit == '1')               /* While extra info.       */
     {
          extra_bit_picture(1);           /* Signifies extra info    */
          extra_information_picture(8);/*Byte of extra info.    */
     }
     extra_bit_picture(1);                /* '0' - No more info.     */
                                          /* Byte Align Bit Stream  */
     if (nextbits(32) == 0x000001b5)      /* If extension start code*/
     {
          extension_start_code(32);       /* 0x000001b5              */
          while (nextbits(24) != 0x000001)
          {
               extension_data(8);         /* Byte of extension data */
          }
     }
     if (nextbits(32) == 0x000001b2)      /* If user data start code*/
     {
          user_data_start_code(32);       /* 0x000001b2              */
          while (nextbits(24) != 0x000001)
          {
               user_data(8);              /* Byte of user data       */
          }
     }
}
```

Figure A-9: MPEG-1 Picture Header Definition

The final part of the header contains the extra information. At the moment this information is undefined but can be present in a valid header. Each byte of extra information takes up nine bits of the picture header. The first bit is used to signify that there is a byte of extra information whilst the next eight bits make up this information. This goes on until the more information bit is set to 0. At this point the extra information ends and we have the option of extension or user data appended to the end of the picture header, exactly the same as for the sequence and GOP headers.

Once the picture header is processed, it is immediately followed by a slice, which must also be processed. At the end of the slice, we can either have another slice or we have reached the end of the picture layer.

A.2.3.4 Slice Layer

The slice layer defines a binary bit stream that makes up a segment of a single picture in the picture layer. The slice layer contains a slice header followed by one or more macroblocks, where each macroblock signifies a 16x16 block within the picture. Note that the definition requires at least one macroblock to be present within the slice, this is required since no macroblocks would imply a slice that defines no part of a picture.

The slice header, as formally specified in Figure A-10, is of variable length and begins with a unique byte aligned 32 bit start code in the range 0x00000101 to 0x000001AF. The final byte of the slice header defines the number of the slice being identified and determines the macroblock row at which this slice starts, thereby defining the vertical start point on the screen for this slice. The horizontal start point is encoded within the macroblock. The next five bits are used to indicate the quantiser scale factor.

```
slice header()
{
     slice_start_code(32);               /* 0x00000101 - 0x000001af*/
     quantiser_scale(5);                 /* Quantiser scale        */
     while (nextbit == '1')              /* While extra info.      */
     {
          extra_bit_slice(1);            /* Signifies extra info   */
          extra_information_slice(8);    /* Byte of extra info.    */
     }
     extra_bit_slice(1);                 /* '0' - No more info.    */
}
```

Figure A-10: MPEG-1 Slice Header Definition

The final part of the slice header contains extra information which is encoded in exactly the same as for the picture header. Each byte of extra information is encoded as nine bits with a zero bit determining the end of the extra information segment of the slice header.

Once the slice header is processed, it is immediately followed by a macroblock, which must also be processed. At the end of the macroblock, we can either have another macroblock or we have reached the end of the slice layer.

A.2.3.5 Macroblock Layer

The macroblock layer defines a binary bit stream that makes up a single 16x16 block of pixels within a single picture. The macroblock layer consists of a macroblock header, which is formally specified in Figure A-11, followed by 6 blocks, where each block defines an 8x8 grid of values used to determine the makeup of the 16x16 pix. ' .acroblock. If the macroblock is part of a D-Frame, then the macroblock layer is terminated with an end of macroblock bit which is set to 1.

A macroblock begins with zero or more macroblock stuffing codes. This code is eleven bits long and is of the format '00000001111'. This code is then followed by zero or more macroblock escape codes, which are also eleven bits long and encoded as '00000001000'. Following the escape codes are macroblock address increment and macroblock type codes. Each of these are encoded using huffman codes to minimise their lengths. The address increment is encoded with length one to eleven bits and is used to determine the horizontal position of this macroblock within the picture. The macroblock type is encoded with length one to six bits and is used in a look up table to determine whether this macroblock is intra or pattern coded, contains forward motion vectors, backward motion vectors and whether the quantiser value is changed.

```
macroblock header()
{
     while (nextbits(11)=='0000000111')
     {
          macroblock_stuffing(11);      /* '00000001111'          */
     }
     while (nextbits(11)=='00000001000')
     {
          macroblock_escape(11);        /* '00000001000'          */
     }
     address_increment(1-11);           /* Field length variable  */
                                        /* from 1 to 11 bits      */
     macroblock_type(1-6);              /* Type of macroblock,    */
                                        /* length variable from 1 */
                                        /* to 6 bits              */
     if (macroblock_quant)
     {
          quantiser_scale(5);           /* New quantiser scale    */
     }
     if (macroblock_motion_forward)     /* If forward motion vect.*/
     {
          fwd_motion_vectors(2-34);     /* Forward motion vectors */
                                        /* encoded as four values */
                                        /* with total length of   */
                                        /* between 2 and 34 bits. */
     }
     if (macroblock_motion_backward)    /* If backward motion vect*/
     {
          bkwd_motion_vectors(2-34);    /* Backward motion vectors*/
                                        /* encoded as four values */
                                        /* with total length of   */
                                        /* between 2 and 34 bits. */
     }
     if (macro_block_pattern)
     {
          coded_block_pattern(3-9);     /* Coded block pattern of */
                                        /* length 3 to 9 bits     */
     }
}
```

Figure A-11: MPEG-1 Macroblock Header Definition

Following the macroblock type are optional values encoded only if their presence is determined by the macroblock type field. If the quantiser value has changed then the new value is encoded as a five bit value. If a forward motion vector exists then this is encoded as four distinct values with a length between two and thirty-four bits. The backward motion vector, if required, is encoded the same way.

If the macroblock is pattern coded, determined by the macroblock type field, then the coded block pattern is encoded as a huffman code of length three to nine bits. Decoding this field will tell us which of the six blocks have been coded, if a block is not coded then all of its DCT coefficients will be zero and takes up no bits in the bit stream. Otherwise the block is encoded as part of the bit stream. Following this optional field is the block data itself with each of the six blocks encoded unless specified by the pattern code. If the macroblock is an intra macroblock, determined by the macroblock type field, then the block is encoded with a separate DC coefficient, otherwise the DC coefficient is encoded with the AC coefficeients. This fact is used to determine how the block will be decoded.

This is a test to see how many more lines can possibly be squeezed to fit onyo the current page. It appears that two lines is the upper limit that word will be happy with. Now padding with rubbish to ensure that all is going to be happy and correct before we try shuffling the position of the figure

Once all six blocks have been decoded, we reach the end of the macroblock layer. If the picture is a D-Frame, then we have a single '1' bit to terminate the macroblock, otherwise the macroblock ends at this point.

A.2.3.6 Block Layer

The block layer defines a binary bit stream that encodes a single 8x8 DCT block, it also forms the lowest layer of the MPEG-1 Video Stream. In cases where all the DCT coefficients are zero, the block is simply skipped, this is usually done in the macroblock layer.

Blocks can be encoded either with the DC coefficient encoded seperately from the AC coefficients or with the AC coefficients. Which decoding procedure is used is decided at the macroblock layer where this information is encoded for each block. If the DC coefficients are encoded seperately then they are encoded as follows. First we have a value that ranges between two and seven bits in length, this forms a unique huffman code that determines the bit length of the next value. As for similar instances mentioned earlier, these exact values are unimportant. The next value determines the difference of the DC coefficient from the predicted value. If the DC coefficient is encoded with the AC coefficients, then this is simple encoded as a huffman code of length two to twenty-eight bits at the start of the block. If we are decoding a D-Frame then the block ends here with a single DC coefficient only.

Following the DC coefficient are the AC coefficients listed one after the other until all remaining coefficients are zero. These are encoded in a huffman code of length three to twenty-eight bits for each coefficient, the huffman code '10' indicates that all following coefficients are zero and that the block ends here.

A.2.4 MPEG-1 Audio Compression

The MPEG-1 Audio Compression standard is defined in part three of the MPEG-1 standard. This document is concerned with standardising the MPEG-1 Audio bit stream as well as defining a decoded that can reconstruct the audio sequence from this bit stream. We will be looking mainly at the format of the bit stream itself to help gain a clear picture of the binary format of an MPEG-1 Audio Stream. We will also take a brief look at the principles of audio compression and how they are applied to MPEG-1 Audio. While other audio compression standards exist, it is interesting to note that MPEG Audio Compression algorithm is the first standard for digital compression of high quality audio.

MPEG Audio Compression relies heavily on human perception of sounds. In order to achieve a high rate of audio compression, we must, as in video compression, resort to lossy compression. This means that the reconstructed audio stream will not be an exact match to the original audio stream. The main aim is to reproduce an audio stream that whilst not physically matching the original audio sample, must perceptually match the original, that is, it must sound the same as the original audio sample. As for MPEG Video Compression, the aim of the standard is to specify the format of the MPEG Audio bit stream and how a decoder should decipher this bit stream to reproduce an audio sample. The standard specifically denies specification of an Audio encoder to allow new technology and compression ideas to be integrated with MPEG Audio; the only requirement is that the encoder produces a valid MPEG Audio bit stream.

While most lower audio quality compression algorithms take into account special features of the audio they intend to compress (such as human speech), the MPEG Audio compression algorithm uses perceptual limitations of the human auditory system to decide which sections of the original audio sample to discard. By following this model, only perceptually irrelevant audio signals are discarded, and MPEG Audio compression is suited to any audio sample that is meant to be heard by the human ear.

The key to MPEG Audio compression lies in the psycho-acoustic model. The basic premise of an encoder is that the original audio stream passes through both a bank of filters that convert the audio data into multiple subbands of frequency values, and through the psycho-acoustic model which determines the ratio of signal energy to masking threshold for each subband. What this means is that the psycho-acoustic model determines the number of bits required to store the information in each subband such that the reproduced audio signal is perceptually the same as the original audio stream. The difficult part is in building the psycho-acoustic model to correctly decide on which bits to encode. The decoder is far easier to build as it merely needs to unpack the bit stream, reconstruct the frequency samples and finally convert the frequency samples into a raw audio data stream.

The job of the MPEG Audio filter bank is to divide the audio signal into 32 equal width frequency subbands. These subbands can later be recombined into an audio stream at the decoder. The process of reconstructing the original audio signal is not lossless, but the design of the filter bank minimises this loss to inaudible effects. As mentioned earlier, the psycho-acoustic model is used to determine how the human ear perceives an audio signal. Many experiments on perceptual hearing were performed which created statistical data on threshold hearing. These results showed that a strong tonal frequency masked out the perception of weaker nearby tonal frequencies. They also showed that a load tonal frequency also masked the ability to hear weaker tonal frequencies immediately after the louder tone stopped. All this data was accumulated into the psycho-acoustic model to help the encoder decide how many bits to assign in encoding each subband.

The MPEG standard defines two psycho-acoustic models, which offer varying degrees of complexity in order to produce better compression ratios. As mentioned earlier, the job of an MPEG Audio decoder is far simpler as it merely needs to unpack the frequency values and reconstruct an audio signal for playback.

Finally, the MPEG Audio compression standard defines three layers. These layers should not be confused with the layers defined in the video compression standard. The MPEG Audio layers refer to differing audio quality at a certain compression ratio. The most complex of these, Layer III, has become extremely popular as a tool for compression of high quality audio sources for storage on personal computers. The format of the bit stream varies for each of these three layers and therefore a compliant MPEG Audio decoder must be able to decode each of these three different types of MPEG Audio streams.

A.2.5 MPEG-1 Audio Stream

The MPEG-1 Audio Stream is a binary stream which defines a single audio stream within an MPEG-1 System Stream. The MPEG-1 Audio Stream is multiplexed along with other Video Streams and Audio Streams within the MPEG-1 System Stream. The MPEG-1 Audio Stream on its own provides the information necessary to decode and playback a compressed audio stream. As mentioned previously, the audio stream can be encoded in one of three different ways, called layers in MPEG Audio terminology. Each of these three layers depicts a different decoding scheme that must be used to reconstruct the raw audio data. The general format of the MPEG-1 Audio bit stream remains constant with the same packet header format being used for each of the three audio compression layers. However, how the data is stored following each packet header differs depending upon which layer has been used to represent the encoded audio stream.

A.2.5.1 MPEG-1 Audio Stream Packet Header

The MPEG Audio Stream header format is common to all three types of MPEG-1 Audio compression. The MPEG Audio Stream is periodically broken up into frames where each frame is begun by the MPEG Audio Stream Packet Header. The packet header is exactly 32 bits long and is formally defined in Figure A-12. Following the header is the audio frame data, which is of variable length, and differs depending upon which type of compression has been chosen. The level of audio compression is stored in a field of the MPEG Audio header.

The MPEG-1 Audio Packet Header always commences with a synchronisation word of length 12 bits. This word has all its bits set to one to indicate the start of the Audio Header. Following the synchronisation word is a single bit, which is set to one to identify an MPEG Audio stream. This is then followed by a two bit field which identifies which of the three layers should be used in decoding the audio stream. The Layer number field is followed by an error protection flag and then by a four bit field signifying the bit rate of the audio stream. This is then followed by a two bit sampling frequency field, a single padding bit, a single private bit, a two bit mode field and a two bit mode extension field. Finally the header is rounded up with a copyright flag, original/copy flag and a two bit emphasis field.

```
audio header()
{
     synchronisation_word(11);          /* '11111111111'          */
     stream_ID(1);                      /* '1' for MPEG Audio     */
     layer(2);                          /* MPEG Compression Layer */
     error_protection(1);               /* Is error CRC present   */
     bitrate_index(4);                  /* Stream bit rate        */
     sampling_frequency(2);             /* Sampling Frequency     */
     padding_bit(1);                    /* Padding bit            */
     private_bit(1);                    /* Private bit            */
     mode(2);                           /* Mode information       */
     mode_extension(2);                 /* Mode extension info.   */
     copyright(1);                      /* Copyright on stream    */
     original(1);                       /* Is original or copy    */
     emphasis(2);                       /* Emphasis field         */
}
```

Figure A-12: MPEG-1 Audio Header Definition

In the interests of defining the MPEG-1 Audio Stream, the only fields in the MPEG Audio header that are of importance is the Layer field and the error protection flag. The value contained in the Layer field informs us of the compression layer in use and therefore tells us the format of the data following the frame header. The value of the error correction flag tells us whether or not included in the frame data is a 16 bit CRC error-check word. If present, the error-check CRC immediately follows the audio header and immediately precedes the audio data itself. The layout and format of the audio data is dependant on the audio compression layer in use, the format of each layer will be more fully described in the following sections.

A.2.5.2 MPEG-1 Audio Layer I Data Representation

In MPEG-1 Audio Layer I compression, samples from the 32 filter subbands are grouped in lots of 12 samples. These 384 audio samples are stored in a single frame with the audio header. The first data field in the Layer I payload represents the bit allocation for each group of 12 samples that are stored. Each value is of length four bits and represents a value between 0 and 15 bits. An example value of '0110' would mean that each of the 12 samples for that particular subband are stored as a value of length six bits. The length of the bits allocation field can be either 128 bits, or 256 bits if the audio being stored is a stereo sample.

Following the bit allocation field are the scale factors for each sample. If the bit allocation for a sample group is not zero then six bits are allocated as a scale factor. The eventual sample is multiplied by the scale factor to recover the quantised subband value. If the bit allocation for a sample group is zero, this means that zero bits are used to store the sample and so no scale factor is required as the eventual subband value will merely be zero. As such, the scale factor field can be anywhere from zero bits to 384 bits in length as long as the field length is a multiple of 6 bits.

Finally the sample data for the frame follows the scale factors. The length of this field is dependant on the values in the bit allocation field which determine the number of bits allocated to each sample. Once the end of the samples has been reached, there is the option of ancillary data in the frame. Ancillary data contains extraneous information about the MPEG Stream such as lyrics, composer or other information.

A.2.5.3 MPEG-1 Audio Layer II Data Representation

The Layer II compression algorithm is a simple enhancement of the Layer I algorithm. It encodes more samples into each frame, whereas Layer I encodes samples in groups of 12 for each subband per frame, Layer II encodes samples in three groups of 12 for each subband per frame. Layer II also imposes restrictions on possible bit allocations for values from middle and higher subbands, as well as representing the bit allocation, scale factors and samples themselves with more compact code; thus leaving more bits available to improve audio quality.

The first data field in the Layer II payload again represents the bit allocation. One bit allocation value is assigned to each trio of 12 samples from each subband. This means that there are a total of 32 bit allocation fields. These values are encoded using variable length codes and bit allocation field can be anywhere from 26 to 188 bits in length. Following the bit allocation fields is the Scale Factor Selection Information (SCFSI) Field. This field informs the decoder of how the scale factors are to be applied. In general, one scale factor is assigned to each group of 12 samples from each subband. However, the presence of the scale factor in the final code is determined the value in the bit allocation field and the SCFSI Field for that scale factor. Obviously, if the bit allocation for a sample is zero, there is no need to encode a scale factor. Similarly, information stored in the SCFSI Field indicate whether a scale factor for a group of 12 samples in a subband will be shared by any of the other two groups of 12 samples in the same subband. This is done when the scale factors for two groups are sufficiently close or when temporal noise masking will hide any distortion caused by using an incorrect scale factor. What this means is that each trio of 12 samples in a subband will have either 0, 1, 2 or 3 scale factors. The information in the SCFSI Field indicates how the scale factors are to be shared amongst the groups. The length of the SCFSI Field can vary from 0 to 60 bits.

Following the SCFSI Field is the Scale Factor Field. As for Layer I encoding, this field contains all the non zero scale factors required by the groups of samples being coded. Since a more compact variable length code is being used, this field can have a length ranging in size from 0 to 1080 bits. The actual samples follow the Scale Factor Field, again these are coded using a variable length code and so produce better compression ratios (or better quality at the same compression ratio). As for Layer I encoding, ancillary data on the MPEG-1 Audio stream can follow the sample data to round out the frame.

A.2.5.4 MPEG-1 Audio Layer III Data Representation

The Layer III audio compression algorithm is more complex than the preceding two algorithms. On the other hand, it provides greater compression rates and therefore better quality sound at similar bit rates. While the Layer III algorithm is based on the same filters as those in Layer I and II, it compensates for some of the design deficiencies in the filter bank with a modified discrete cosine transform. The MDCT further subdivide the frequency subbands to provide better spectral resolution.

These more accurate values can then be used by the encoder to cancel some aliasing introduced by the filter bank. The MDCT values are then entropy encoded with Huffman codes to produce the shortest bit stream possible.

An interesting feature of MPEG Layer III encoding is that if not much information is present in a block of audio, then the block is encoded short, at a far lower bit rate than the rest of the audio stream. This means that Layer III coding is the only MPEG audio compression layer that supports variable bit rate encoding. The Layer III algorithm maintains a bit reservoir where it keeps track of how many bits have been used to encode the bit stream versus how many bits have been allocated to encode the bit stream. The difference in these two values keeps track of how many extra bits we currently have to store information. If a block of audio data contains a lot of information, bits can be borrowed from the bit reservoir to help encode the extra information. Thus we encode information rich blocks at a higher bit rate than the overall bit rate. It is important to note that the bit reservoir is a debit account, while we can invest spare bits in the bit reservoir to be used to encode data at a later point, we are not allowed to go into credit on the bit reservoir to encode extra data. The reason for this is twofold: one, if we go into credit, we cannot predict when or if we will ever pay back the extra bits to the reservoir, and two, if the compressed audio stream is being streamed over a communications link, we are obliged not to encode past the agreed bit rate. Storing extra bits in the reservoir is OK, data for blocks of audio to be decoded in the future arrive at the decoder a little early and the decoder processes them when ready. If we go into credit on the bit reservoir, bits that are needed to decode for this time period will not arrive until the next, at which point it will be too late.

The interesting thing with Layer III compression is how the bit reservoir is integrated into the existing audio bit stream format. As for Layer II, Layer III encodes samples in three groups of 12 for each subband per frame, however the data is stored in a different format. Following the audio header and optional CRC is the audio frame side information and then the main data. The side information contains information on how the main data for the audio frame is compressed, including values for the MDCT transforms, aliasing and noise reduction. The side information also contains a pointer called main_data_begin. This pointer refers to the negative offset that needs to be applied to the start of the main data section to where the audio data for this frame begins. The value is a negative pointer because as mentioned before, the bit reservoir cannot go into credit, so the audio data for this frame must start by the start of the main data. An offset stored in main_data_begin means that the audio information for this audio frame actually begins in the main data section of the previous frame. As such, the actual information for an audio frame can come earlier in the bit stream than the header for that frame.

When constructing a Layer III decoder, we would extract all of the data from the main data section and place it into a queue. We would then process the audio header and side information for a frame, from this we could then begin extracting bits from the queue to decode an audio frame. Whatever bits remain in the queue belong to the next audio frame. When processing the next frame, we extract the next block of main data and add it to the end of the queue. We then similarly process the next frames header and side information before extracting bits from the queue to decode audio data.

While the audio frame headers are equally spaced throughout the audio stream, the location of the start of audio data for each frame is not. For playback putposes commencing from the middle of an audio stream, this means that the first audio frame received cannot be player if the main_data_begin pointer contains an offset. If this is true then the first frame received serves to hold part of the audio data for the next frame to be received and decoded.

A.3 MPEG-2

Work on the MPEG-2 Audio/Visual compression standard begun as the MPEG-1 standard was being finalised. The goal of MPEG-2 was to provide compression of higher quality video signals for entertainment purposes. Indeed, MPEG-2 not only supports higher resolution images, HDTV and interlaced video, but also updates the audio compression standard to include multi-channel audio encoding, predominately for surround sound purposes. MPEG-2 video is the compression standard employed on the (Digital Versatile Disk) DVD entertainment media.

The basic format of the MPEG-2 Stream is similar to that for MPEG-1. In MPEG-2 the System Stream has been renamed as the Program Stream, however its contents and purpose is similar in scope. Similarly for MPEG-2 Video and Audio Streams, the formats have been updated to store extra information required for the new format. It is interesting to note that an MPEG-2 is backwards compatible with MPEG-1, an MPEG-2 Program Stream decoder can decode any MPEG-1 encoded bitstream.

A new stream type within MPEG is the Transport Stream which can be used in place of the System Stream. The Transport Stream was designed with streaming of MPEG-2 video over a network in mind. Data is stored in much smaller packets of a fixed size, this fixed size allows for implementation of a hardware system to deliver video over a network. The MPEG-2 Transport stream is a bold idea for implementation in systems that do not involve direct playback from storage media and was included in the standards by MPEG committee members mindful of the then new application entitled video serving. MPEG-2 Transport Stream packets are made up of a 4 byte header followed by a 184 byte payload.

When it comes to providing video on demand services, it is more likely that this will be done over a generic data network. The current trend is to move away from one network for each application and to use a single network for all telecommunications. In this case, we have to look at supporting the MPEG-2 Transport Stream over existing data networks. The two predominant networks are ATM and IP: ATM uses fixed 53 byte packets with a 48 byte payload and IP uses variable size packets. In both cases the packet size does not properly fit the Transport Stream packet size.

In the case of ATM the major issue is that ATM is likely to be used only in the backbone of networks and is unlikely to be found at the desktop. For IP networks, while it is possible to have a 188 byte IP packet, this would cause a large number of IP packets to be generated to support a stream of 6 Mb/s. IP Packets need to be transported through the network by routers, these routers perform their function in software and are generally limited in the number of packets that they can route every second. Smaller IP packets are a waste of resources and would assist in bringing the network down.

A.4 MPEG-4

MPEG-4 Audio Visual compression is extremely interesting, not least because its bit rate allows it to be the most likely candidate in providing quality video on the Internet. MPEG-4 is geared towards higher compression rates and specifies for compression of audio and video at bit rates between 64 kB/s and 1 Mb/s. MPEG-4 is also extremely ambitious in that it also seeks to provide interactive functions with video rather than using video as a pure entertainment medium such as MPEG-1 and MPEG-2. As such, MPEG-4 video is highly likely to be used in the Internet, an extremely interactive environment.

MPEG-4 has many features in common with MPEG-1 and MPEG-2 in compression techniques but also introduces new concepts. Some of these features include new techniques of encoding such as wavelet encoding and fractal encoding. With wavelet encoding, an image is converted to the frequency domain, sub sampled with filters and then the frequency components are compressed. The original image is built up from reassembling in the frequency domain and then converting the image back to the spatial domain. Fractal encoding consists of encoding some parts of the image as an image itself. Other sections of the image are encoded as references to these other parts at a different magnification and rotation. Fractal compression exploits self-similarities in the image and can produce extremely high compression rates. Another new feature is the introduction of object encoding where objects are defined in the original scene and encoded separately. These objects could then form the background, one major character, a moving object, etc. MPEG-4 would then allow different coding techniques to be applied to each object, also the decoder would allow the user to remove objects from the scene or allow different actions to occur when one object is selected.

While MPEG-4 compression offers hope for immediate use in the Internet, it appears more likely that it will be used where user interactivity will be at a premium. This is exemplified in instances such as on line travel agencies, real estate and current affairs services. These provide environments where the user is after information more than entertainment and an interactive multimedia presentation can best convey that information. Encryption of audiovisual content is mainly important when owners of the material wish to prevent others from illegally using it. This is initially going to occur in the entertainment market with the provision of movies over a public network. In this case, the video provided by MPEG-4 compression is not of sufficiently high quality for entertainment purposes.

Appendix B
An Introduction to Cryptography

In order to develop a secure system for streaming video over a network, it will be necessary to use modern digital encryption techniques to protect the video stream whilst it is being transmitted over an insecure communications medium. In this Appendix, I provide an introduction to cryptographic techniques for readers unschooled in this area. The science of information protection is called cryptography, the science of breaking encrypted codes is called cryptanalysis. I will not cryptanalyse different encryption algorithms, details on how secure certain algorithms are can be found in many of the provided references.

I will begin my review with an introduction into the art of cryptography and a brief look at basic concepts. I will then look in turn at both Public Key and Private Key cryptography. For Public Key Cryptography I will examine the principles behind its application, present a description of the RSA Public Key algorithm, and discuss the applicability of Public Key schemes to streaming multimedia. I will also discuss the importance of Key Management and how it applies to the application of streaming video, this issue will be regarded as external to the actual encryption technique and will not be considered when developing a secure encryption scheme. For Private Key Cryptography, I will examine the concepts of both Block and Stream Ciphers, again looking at principles and applicability to our task. I will look at a common example of Block Ciphers (DES) and Stream Ciphers (SEAL), and describe their algorithms. Finally, I will conclude by presenting a recommendation on which algorithm type is best suited to the application of streaming multimedia.

The information presented in this Appendix is a summary of that found in the following references. (Denning, 1983; Diffie and Hellmann, 1976; Fernandes, 1999; Jurišic and Menezes, 1997; Kaliski and Robshaw, 1996; Menezes et al., 1997; Meyer and Matyas, 1982; NIST, 1993a; Preneel et al., 1998; Rivest et al., 1978; Rogaway and Coppersmith, 1998; RSA, 1996; Schneier, 1996a; Schneier, 1998; Stinson, 1995)

B.1 Basic Cryptographic Techniques

Up until recent times, secure telecommunications was strictly the purview of military organisations; the general public only had access to relatively weak cryptography. With the advent of modern day personal computers, vast computing power is now available to everybody. This puts high level telecommunications security within reach of the average computer user, techniques that were once available only to military organisations are now available to everybody.

This has caused some consternation in both political and military circles, which believe that cryptography would be a powerful tool in the hands of criminals and terrorists. This Appendix ignores the political issues of cryptography and instead looks at the implementation issues involved with encryption and decryption of a high bitrate datastream such as as multimedia.

B.1.1 Terminology

Before we go any further, it is essential to review the main terms that are used in the science of cryptology. Table B-1 lists the common terminology in use.

Term	Meaning
Plaintext	The message to be sent securely from the source to the intended destination of the message.
Encryption	The process of disguising a message in such a way as to hide its substance.
Ciphertext	An encrypted message that is sent over an insecure communications medium.
Decryption	The process of reverting ciphertext back into plaintext.
Cryptography	The art and science of keeping messages secure.
Cryptographers	People who practice cryptography.
Cryptanalysis	The art and science of breaking ciphertext.
Cryptanalysts	Practitioners of cryptanalysis.
Cryptology	Branch of mathematics encompassing both cryptography and cryptanalysis.

Table B-1 Common terminology used in the science of cryptology

Plaintext is usually denoted by **M**. A plaintext message need not necessarily be an ASCII text message, it can be any arbitrary stream of bits, which can be used to denote a text file, a bitmap, a stream of digitized voice, digital video, etc. For purposes of cryptography, **M** is simply binary data which can be intended for either transmission or storage. **M** is the message to be encrypted.

Ciphertext is usually denoted by **C**. Like plaintext, cyphertext is also a stream of binary data. Depending on the encryption scheme utilised, **C** can either be the same size as **M**, or possibly larger. The encryption function **E()**, operates on **M** to produce **C**. Or, in mathematical notation:

$$E(M) = C \tag{B.1}$$

In the reverse process, the decryption function **D()** operates on **C** to produce **M**:

$$D(C) = M \tag{B.2}$$

Since the whole point of encrypting and then decrypting a message is to recover the original plaintext, the following identity must hold true:

$$D(E(M)) = M \tag{B.3}$$

These three equations can be represented diagrammatically as in Figure B-1. This diagram demonstrates the basic framework in which cryptographic functions operate, the plaintext message **M** is encrypted by function **E()** to form a ciphertext message **C** which can then safely be transmitted across a possibly compromised telecommunications medium. At the receiver end, the ciphertext **C** is decrypted by function **D()** to retrieve the plaintext message **M**.

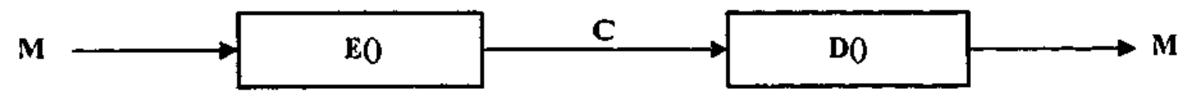


Figure B-1: Basic Cryptographic Framework

As well as being used for the protection of information, techniques developed in cryptology are also used to provide authentication, integrity and non-repudiation.

- **Authentication** – A means of proving the originator of the message.
- **Integrity** – A means of detection of message tampering.
- **Non-repudiation** – Provides a situation where a sender of the message cannot deny sending the message.

All these tasks are important in the art of cryptology in general but less so when applied to protection of streaming multimedia. Authentication and integrity would be used in the key management stage of streaming multimedia to ensure that a user is allowed to view the transmitted media. Once the user has the key required to decrypt and view the encoded multimedia stream, these issues become irrelevant. This is because the viewer is already authorised to decode and view the stream and that any tampering will be evident in the visual display produced for the viewer.

A cryptographic algorithm, also called a cipher, is the mathematical function that is used for encryption of plaintext for secure communications and the decryption of the cyphertext back into its original plaintext. Some algorithms obtain their security by keeping the actual algorithm secret, these algorithms are known as restricted algorithms. These are generally considered inadequate, since they cannot be used effectively by a large or changing group. This is because if one person leaves the group, all users must switch to a different algorithm. Also, if one user accidentally leaks the algorithm, then all users must again switch to a different algorithm. Further, these algorithms provide little control over the strength of the encryption, every group must develop their own algorithms to keep secret and cannot use a product developed by an external source. Despite these problems, restricted algorithms are widely used for low security applications where merely hiding information is the main goal of encryption. An example of this type of application would be storing a table of high scores for a game on disk, this avoids users editing a plaintext file containing the games high scores by hiding the information.

B.1.2 Cryptographic Keys

The problem with restricted cryptographic algorithms is solved using the concept of cryptographic keys, usually denoted by **K**. The key is one of a large number of values and the range of possible values is usually referred to as the keyspace. A cryptographic system inclusive of keys can be described diagrammatically as in Figure B-2. The encryption system now takes as input both the plaintext **M** and the encryption key **K_1**, similarly, the decryption system takes as input both the ciphertext **C** and the decryption key **K_2**. In this situation, the plaintext **M** is encrypted to a ciphertext **C** that is dependant not only on the encryption function **E()**, but also on the encryption key **K_1**. Similarly, in order to retrieve the original plaintext from the ciphertext, the receiver requires knowledge not only of the decryption function **D()**, but also of the decryption key **K_2**.

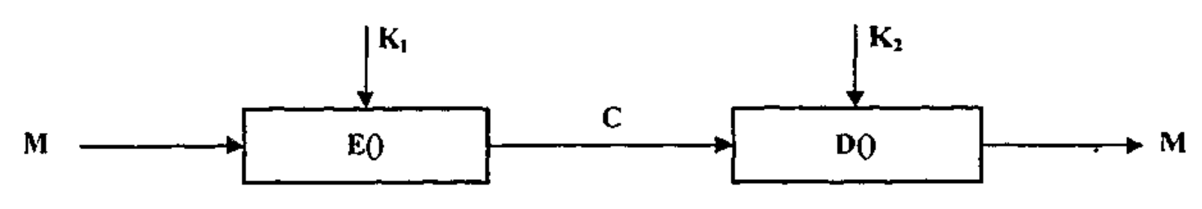


Figure B-2: Cryptographic System Inclusive of Keys

As such, the mathematical encryption and decryption operations become dependant on the key such that the mathematical notation of these functions become:

$$E(K_1, M) = C \tag{B.4}$$

$$D(K_2, C) = M \tag{B.5}$$

$$D(K_2, E(K_1, M)) = M \tag{B.6}$$

The security provided by these algorithms is now based in the secrecy of the chosen algorithm, as well as the secrecy of the chosen key. If the algorithm can be proven to provide secure encryption of plaintext regardless of the chosen key, then this algorithm can be freely published and the security of the algorithm is based in the key.

In general, encryption algorithms can be broken up into two types, the first of these is where the decryption key is identical to the encryption key, or can be directly calculated from the encryption key. These algorithms are termed as private key encryption algorithms and the security that they provide rests in the secrecy of the key. As long as the key itself remains secret, secure communications can be provided between two parties. An example of a private key encryption algorithm is diagrammatically shown in Figure B-3. Within the class of private key encryption, algorithms can be further classified as either block encryption or stream encryption algorithms. Block encryption algorithms operate on the plaintext in blocks of a fixed size, typically 64 bits. Stream encryption algorithms, on the other hand, operate on each individual bit of the plaintext.

The second type of encryption algorithms are called public key encryption algorithms, where the encryption key is not only different from the decryption key, but also that the decryption key is not calculable from the encryption key and vice-versa. One of these keys is usually made public whilst the second is kept private. Other people can then securely send a message encrypted with the well-known public key, as only the person holding the private key can decrypt the message.

In many existing public key ciphers, messages encrypted with the private key can only be decrypted using the public key. Here we instantly obtain the property of authentication as if a message can be decoded using a public key, only the person owning the corresponding private key could have sent that message.

There are many encryption algorithms available to be used, each offering differing degrees of security, however all algorithms are eventually beaten by a brute force attack in which every single key in the keyspace is tried until the plaintext is retrieved. When choosing an encryption algorithm and the degree of security required, it is important to consider the cost required to break the message encrypted with that algorithm. If the cost is greater than the value of the plaintext message, then the level of encryption is adequate for the task, as the cost to the cryptanalyst is greater than obtaining access to the message through legal means. If the message is one of military importance, then the level of security need be enough to withstand a code breaking effort of many years. If, on the other hand, the message is a multimedia stream for entertainment purposes, then the cost involved to break the code need only be more than the cost to legitimately purchase access to view the media in the first place. Even accounting for access to a digital copy of the media at hand, this cost is likely to be under a few thousand dollars. As such, the security level of the encrypted multimedia stream need not be of the very highest level available.

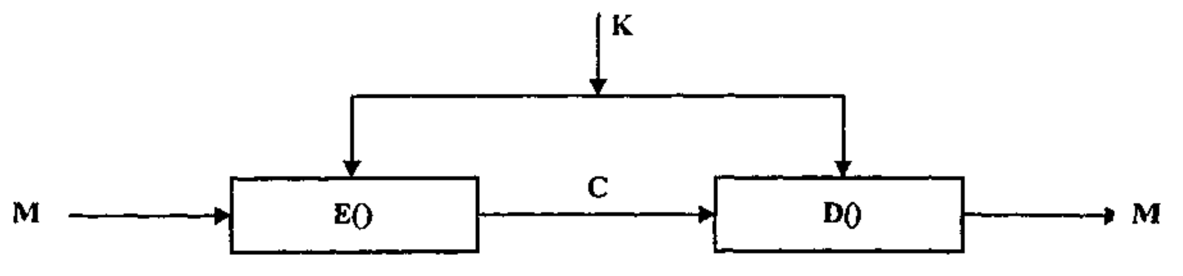


Figure B-3: Private Key Cryptographic System

B.1.3 One Time Pad Cipher

Most available encryption algorithms are not guaranteed to provide absolute security. This is because if the algorithm is known, then the encrypted messages can always be decoded given enough computing time. By simply trying every single possible key in a brute force approach, we will eventually decode the message to its original form. Similarly, knowledge of the algorithm in question can allow us to tailor our attack in order to decode the encrypted message in a shorter period of time. There is however, one perfectly secure encryption technique that cannot be broken, called the One Time Pad Cipher. This encryption method was invented in 1917 by Major Joseph Mauborgne and AT&T's Gilbert Vernam.

In simple terms, a One Time Pad consists of a large, non repeating set of random key letters, one key written on each page of the pad. There are two such identical pads, one with the sender and the second with the receiver of the encrypted message. When sending the message, the sender adds the random key written on the first page of the pad to the first plaintext character to encrypt, once this is done, the top page of the pad is removed and completely destroyed. When the message is received at the destination end, the random key written on the top page of the pad is subtracted from the first character of the received message to retrieve the first plaintext character. The top page of the pad is again removed and completely destroyed. The random keys in the One Time Pad should never be used again to encrypt a different message. In the real world, an entire sequence of random keys would be written on a single page but the rule remains that once a key from the One Time Pad has been used, it should be irretrievably destroyed.

In modern cryptography, we can change the concept of a truly random series of keys to a truly random binary series of bits. These individual bits can then be XORed with the bits that make up the plaintext. At the destination end, the ciphertext can then be XORed with the same sequence of random bits to reproduce the original plaintext. As for the concept of destroying physical pages of a real pad, we must still ensure that the One Time Pad of bits is destroyed and not retrievable after use. This can be performed by storing the One Time Pad on a digital tape that is destroyed as it is used.

One of the important features of the One Time Pad Cipher is that the pad consists of a truly random series of bits, it is in fact this feature that ensures that the cipher is absolutely secure. The first reason for this is that if the Pad is randomly generated, then each and every Pad the length of the plaintext is equally likely and as such no information can be obtained about the key stream encoded in the One Time Pad. Since the Pad bit stream is random and contains no information, once this stream is encoded into the plaintext to produce the ciphertext, the ciphertext also has the property of being truly random and containing no information other than the length of the plaintext. As such, the ciphertext could be decoded into any and all plaintexts of the same length with equal probability: We could encrypt the word MPEG to form the ciphertext HWEF, this cipher text is just as likely to decode into MPEG as into CAKE or FISH. Since no information can be obtained from the ciphertext, the cryptanalyst has no clues to help him decode the ciphertext. Even if some of the plaintext is known or guessed, and this is used to retrieve a part of the key bit stream, the fact that the key is truly random would not allow the cryptanalyst to deduce any other part of the key. To underline the importance of a truly random series of bits, consider using a pseudo-random number generator. Since a pseudo-random number generator must be deterministic, they are not truly random. As such, their non randomness can be used against them in a known plaintext attack to deduce the random bit sequence used and therefore to decode the entire message.

The other important feature is that the random sequence is irretrievably destroyed after use. If the same random sequence is used twice to encrypt two different plaintexts, then some information is being presented to the cryptanalyst who can use it to determine the random key sequence and therefore both original plaintexts and any other message encrypted using the same random sequence.

There are two difficulties involved in using the problem of a One Time Pad Cipher, the first of these is in generating the random bit sequences. This is not a trivial task and can only be truly performed by sampling a natural, random occurrence, an example of which could be taking the least significant bit of a regular sampling of a white noise source. The second difficulty lies in distributing an exact copy of this random sequence to both parties involved in communications. It is obvious that this technique cannot be used with a random sequence generated in real time as the receiver would have no way of generating the exact same random key sequence, therefore, the One Time Pads need to be generated ahead of time and distributed to both parties.

The One Time Pad Cipher is provably secure, regardless of how much computing power is thrown at the problem. This is due to the fact that the ciphertext contains no information of the plaintext that has been encrypted. However, the difficulties involved in utilising this scheme make it impractical for our purpose. While there are problems with using a psuedo-random number generator in that the random bit sequence produce is deterministic and therefore not truly secure, the concept of using a psuedo-random generator to emulate the properties of a One Time Pad Cipher is appealing and discussed with Stream Ciphers.

B.1.4 Cryptographic Attacks

Cryptanalysis is the science of breaking secure ciphers. The aim of cryptanalysis is, given ciphertext C, to recover either the original plaintext M, the decryption key K, or a weakness in the cipher algorithm that will eventually lead to the discovery of M or K. Cryptanalysis assumes no prior knowledge of the key K, if the encryption key is compromised through other means then the plaintext is retrieved using non cryptanalytic methods. There are four types of attack possible using cryptanalysis. These are the Ciphertext Only Attack, Known Plaintext Attack, Chosen Plaintext Attack, and Adaptive Chosen Plaintext Attack.

As a cryptographer, one must always assume that an opposing cryptanalyst has access to the encryption algorithm being used. If the algorithm is a piece of software, then that software can be bought and reverse engineered. If the algorithm is implemented as a hardware module, then that hardware can be obtained and reverse engineered. It is important to assume that the crypanalyst will not only have access to encrypted messages, but also access to which algorithm has been utilised to implement that encryption.

B.1.4.1 Ciphertext Only Attack

We have the ciphertext of several messages encrypted using the same encryption algorithm. The task is to recover the plaintext of these messages or to discover the keys used to encrypt these messages. This is the most difficult attack to undertake as the cryptanalyst has the least amount of information available to him, this being knowledge of the algorithm in use and some intercepted encrypted messages.

A common application of the ciphertext only attack is often referred to as the brute force attack. In this case a block of ciphertext is decoded with every key within the keyrange. This leads to a multitude of possible plaintexts. These plaintexts are then analysed for common structures, if the message was known to be a text message then the plaintexts are analysed for known words, similarly sounds can be analysed for similarities with human speech. Other types of plaintext usually have some well defined structures. This analysis reduces the list of possible plaintexts to a smaller number to choose from. If a multiple number of messages have been encrypted using the same key, then we can look for intersecting sets of possible plaintexts to reduce the possible list of keys even further. With the advent of faster computers, this sort of attack is becoming easier to perform. The original DES encryption algorithm used a 56 bit key which has a key range of over 7 x 10^{16} possible keys. It is now possible to build a machine to decode DES encrypted messages with all possible keys within a couple of hours.

B.1.4.2 Known Plaintext Attack

We have the ciphertext of several messages encrypted using the same encryption algorithm, we also have the corresponding plaintext to either some of these messages or to segments of these messages. Again our aim is to deduce the keys used to encrypt the messages such that we can decode the remaining messages. This form of attack was largely used by the English to break the Enigma code used by the Germans in World War II. Since most military message have a common format and use known wording for certain parts of the message, English cryptanalysts had some plaintext to work with when deducing the key used by the Enigma system for that day.

B.1.4.3 Chosen Plaintext Attack

Like for the Known Plaintext Attack, we have the ciphertext of several messages, the associated plaintext, but also the ability to encrypt chosen blocks of plaintext. In this scenario, the cryptanalyst has the ability to enter chosen plaintext into a black box that performs the same encryption function as that used to encrypt the messages. This technique is more powerful as the cryptanalyst can now choose specific blocks of plaintext that may indicate certain properties of the key that was used.

B.1.4.4 Adaptive Chosen Plaintext Attack

The cryptanalyst has better access to the black box and cannot only choose certain plaintexts to be encrypted, but, based on the results, can select another block of plaintext to be encrypted again. By repeated access to the encryption module, the cryptanalyst can more quickly narrow down the range of possible keys.

B.2 Public Key Cryptography

There is a large amount of interest in Public Key Cryptography, which, when introduced in 1976, changed the world view on crpytographic systems. Until then, the concept of cryptography meant using a secret key to protect the plaintext and then using the same secret key to retrieve it. Public Key Cryptography allows us to utilise two separate keys in the process of protecting communications, one which is kept secret by the receiver of the ciphertext and the other which is made public to the world at large. The keys are used as an encryption key/decryption key pair with either of the two keys able to be used as the encryption key.

This scenario completely does away with the problem of agreeing on a secret key for secure telecommunications and can be used in such a way as to guarantee the concepts of authentication, integrity and nonrepudiation. As long as we can ensure the integrity of the public key, we can provide secure telecommunications between a large, changing group of people. Integrity of the public key is important as if a cryptanalyst can incorrectly advertise the public key of an enemy successfully, then he, having the corresponding private key, can masquerade as his enemy and retrieve all messages destined for him. This deception can continue by then re-encrypting the plaintext with the correct public key and forwarding the message to the enemy who remains unaware that his security has been compromised. The problem of ensuring public key integrity is generally solved through the use of a trusted authority that maintains a list of all public keys, which can be requested. These systems would usually be maintained by real world trusted authorities such as governments or banking institutions.

B.2.1 Principles Behind Public Key Cryptography

Public Key Cryptography algorithms are based on the principle of trapdoor, one-way functions. The concept of one-way functions in the form of hash values has been known for a long time in Computer Science. One-way functions in themselves do not provide Public Key Cryptography, but form a fundamental building block not only to Public Key Cryptography, but also to many protocols governing secure telecommunications. By extending the concept of one-way functions to include a trapdoor effect, we are generating the concept of Public Key Cryptography.

The point of a one-way function is to use the plaintext as input to the one-way function, the ciphertext being the output of an irreversible function. In order to retrieve the plaintext, we must reverse an irreversible function. This task is obviously impossible for a true one-way function so we must introduce a trapdoor into this function. A trapdoor consists of knowledge of a secret, usually the secret key, that in conjunction with the ciphertext allows simple reversal of the one-way function.

B.2.1.1 One-Way Functions

Before any further discussion, it is important to note that there is no mathematical proof for the existence of one-way functions. While it is true to say that we cannot guarantee the existence of one-way functions, it would also be true to say that there are many functions that are easy to compute in one direction but extremely difficult to compute in the reverse direction. A simple example is that it is relatively easy to compute that 52,396 x 842,412 is equal to 44,139,019,152. On the other hand, determining the prime factors of 44,139,019,152 is a difficult problem. While it is true to say that reversal of some of these functions is difficult today, that is not to say that new algorithms will be developed in the future that make these problems relatively simple. In fact, there have been many algorithmic developments in the past ten years that have greatly simplified the task of generating the factors of a large number.

I have already mentioned that one-way functions are relatively easy to compute, but significantly harder to reverse. In mathematical terms, this means that given x, it is easy to compute f(x), but given f(x), it is hard to compute x. In this context, hard can be defined as a similar statement to: *It would take millions of years to compute x from f(x), even if all the computers in the world were assigned to the problem.*

A good example of a one-way function was presented by Schneier (Schneier, 1996a). He illustrated that breaking a plate is a good example of a one-way function. It is extremely easy to smash a plate into thousands of small pieces, however, it is extremely difficult to reassemble the plate from the multitude of small pieces.

One-way functions in themselves cannot be used to encrypt a message. There is no point writing a message on Schneier's plate before breaking it: it would be too difficult to reassemble the plate and therefore retrieve the plaintext enscripted on the plate. However, one-way functions do find a great deal of use in secure communications as hash functions. Hash functions are a many-to-one function, where a large, variable length input is transformed into a small, fixed length output. Properties of a good hash function include:

- All possible hash values should be generated with equal probability. This means that for any given document, all possible hash values are equally probable to occur, no one hash value should be more likely than another.
- A single bit change in the original document should lead to a large and indeterminate change in the calculated hash value. This means it should not be possible to predict a new hash value based on changes to the original document.
- Extremely difficult to generate a document that hashes to a particular value. This means it should be near impossible to create a false replacement document that returns the same hash value.

Hash functions are commonly used in cryptographic protocols to ensure authenticity of a document without requiring retransfer of the same document. This sort of functionality provides a good example where one-way functions are useful in the world of cryptography. Common one-way hash functions used in cryptographic protocols today include MD5 and SHA. There exist other one-way hash functions that are either in use or are no longer in use. Details of these hash algorithms and the security provided can be found in works by A, B and C [references].

B.2.1.2 Trapdoor One-Way Functions

Trapdoor one-way functions consist of a family of functions where calculation in one direction (encryption) is easy, however the reverse calculation (decryption) is extremely difficult without knowledge of a secret (private key). Knowledge of the secret or private key reduces the problem of reversing the one-way function to a simple case. This means that we have now realised our two-key cryptographic system described in Figure B-2.

B.2.1.3 Generating Prime Numbers

Prime numbers figure predominately in Public Key Cryptography. A number is prime if its factors only include 1 and itself. In order to generate a prime number, one must select a number at random and then test that number to see if it is prime. There are several probabilistic prime number tests that can be used to determine if a given number is prime with a certain degree of confidence. If this degree of confidence is large enough, then we can be reasonably confident that the given number is indeed a prime. Increasing the degree of confidence with these tests involve increased iterations of the loops within the tests.

Some Cryptographic algorithms require the use of "Strong Primes", this is usually the case when we wish to multiply two primes (*p* and *q*) together to form ***n***, and we wish to increase the difficulty of the problem of factorising ***n***. Even so, there is still speculation as to whether there is a real need to use "Strong Prime" numbers and if they truly offer a greater level of security. A "Strong Prime" is a prime number with the following properties:

- The greatest common divisor of *(p-1)* and *(q-1)* should be small.
- Both *(p-1)* and *(q-1)* should have large prime factors, respectively *p'* and *q'*.
- Both *(p'-1)* and *(q'-1)* should have large prime factors.
- Both *(p+1)* and *(q+1)* should have large prime factors.
- Both *(p-1)/2* and *(q-1)/2* should be prime.

B.2.1.4 Relative Prime Numbers

One of the concepts used in Public Key Cryptography is that of relatively prime numbers, two numbers are relatively prime when they share no factors in common other than 1. The greatest common divisor can be calculated using Euclid's Algorithm, originally devised over 2,000 years ago. A simple description and implementation of Euclid's Algorithm can be found in Sedgewick [ref].

The concept of finding relatively prime numbers is used in RSA Public Key Cryptography when calculating the both the private and public key of the scheme.

B.2.1.5 Extended Euclidean Algorithm

The Extended Euclidean Algorithm is used to determine the inverse of a number modulo another number. In this situation, finding the modulo *n* inverse of a number *a*, means finding *x* such that $(a \cdot x) \bmod n = 1$. In general, this problem has a unique solution if *a* and *n* are relatively prime, and no solution if *a* and *n* are not relatively prime. An implementation and description of the Extended Euclidean Algorithm can be found in [reference].

B.2.1.6 Weaknesses and Practicalities

Most Public Key Cryptographic systems that are in use are based on the difficulty in factoring large numbers. While new algorithms to solve this problem in a shorter time frame are being developed and refined all the time, cryptographers can always stay one step ahead of cryptanalysts by using larger keys and therefore larger numbers that require factorisation. However useful the concept of a Public Key system may be, in most practical implementations it is used to secure and distribute session keys used in Private Key Cryptographic communications. These systems are sometimes called hybrid cryptosystems. In these systems, all users still utilise a Public Key Cryptographic system, but this system is not used to secure all communications. Instead, secure communications using Public Key Cryptography is used to agree on a private key to be used for a Private Key Cryptographic session. All communications using the private key system are then carried out securely. The above procedure is a brief description of the Diffie-Hellman algorithm for secure telecommunications.

The main reason for using this form of hybrid system is due to the speed of Public Key Encryption algorithms, which are usually about 1000 times slower than Private Key Encryption algorithms. This severely limits the rate at which data can be encrypted using Public Key systems, even with computer speed increasing exponentially as suggested by Moores Law. This is because not only will required encryption rates increase due to more available bandwidth, but increased computing power also means extra security requirements in the form of longer keys and increased encryption time. Another reason for using the hybrid system is that Public Key Cryptography solves an important key management problem. Because the session keys used for Private Key Encryption are calculated and communicated at the last minute, there is less chance of this key being compromised than one that was communicated earlier by other means. Also, the session key is destroyed once communication ends and a new key is generated for each communication session. Continuously changing the session key means that even if one message is compromised and the key discovered, other messages are still secure.

Finally, Public Key Cryptographic systems can be vulnerable to chosen plaintext attacks, especially if it is known that the plaintext must conform to a known format. If it is known that plaintext M is one of a set of n possible plaintexts, then a cryptanalyst can easily encrypt all possible plaintexts M_1 through M_n using the public key to obtain a set of cyphertexts C_1 through C_n. These ciphertexts can then be compared to the intercepted ciphertext to determine which of the n tested plaintexts is equal to the original plaintext. This form of attack could be particularly useful in the case of a scenario where the message fits a common form, and example of this could be details of financial transactions where differences in a message may merely be the amount of money being transferred.

B.2.2 Current Public Key Cryptographic Algorithms

Since the introduction of Public Key Cryptographic systems by Whitfield Diffie and Martin Hellman, and later by Ralph Merkle, a number of Public Key cryptosystems have been proposed. Of these, some have been proven to be insecure, others have been proven to be suitable for key distribution but not for data encryption, others have also been proven suitable for digital signatures. Of all the proposed algorithms, three Public Key Cryptographic algorithms, RSA, ElGamal and Rabin, have been proven cryptographically secure and suitable for data encryption. However, all three of these algorithms are slow and both the encryption and decryption processes are many times slower than similarly secure Private Key Cryptosystems. For information, I will now present a review of the RSA Public Key Encryption algorithm, explaining how the algorithm functions.

B.2.2.1 RSA Public Key Cryptosystem

The RSA Public Key Cryptosystem, named for its inventors – Ron Rivest, Adi Shamir and Leonard Adleman, was the first of Public Key systems that is considered secure and has withstood many years of cryptanalysis. The security in the RSA algorithm is based on the difficulty in factoring large numbers. The two cipher keys are calculated as functions of a pair of large prime numbers, of 100 digits and upward in size. It has been proposed that the difficulty in recovering the plaintext given the public key and the ciphertext is equal to factoring the product of the two primes. This proposal has not been proven but the algorithm has withstood many years of cryptanalysis which indicates a fair degree of confidence in it. The RSA algorithm works as follows:

First choose two random large prime numbers *p* and *q*, for maximum security, these two numbers should be of approximately equal length. Once the two numbers have been chosen, calculate *n*, the product of the two primes, then randomly choose an encryption key *e*, such that *e* and $(p-1)(q-1)$ are relatively prime. Finally, use the extended Euclidean algorithm to compute the decryption key *d* such that $e \cdot d = 1 \bmod ((p-1)(q-1))$. At this point *d* and *n* are also relatively prime, either *e* or *d* can be used to formulate the encryption key. We can then publish *e* and *n* as the public key, *d* and *n* is kept as the private key, it is also important to discard *p* and *q* completely, they must never be revealed.

Now that the private and public keys have been chosen, we can encrypt messages. To encrypt a plaintext M, we must first divide into blocks smaller than *n*, the product of the two primes. Each one of these blocks can then be encrypted by raising it to the power of *e*, and taking the modulus *n* of the result. In mathematical terms, if the plaintext M is broken up into blocks M_i, then the corresponding ciphertext blocks can be calculated with the formula:

$$C_i = M_i^e \bmod n \qquad \text{(B.7)}$$

Decrypting the ciphertext back into plaintext involves taking each block of ciphertext, raising it to the power of *d*, and then taking the modulus *n* of the result. This calculation will return the plaintext block corresponding to that ciphertext block. In mathematical terms, this forms the formula:

$$M_i = C_i^d \bmod n \qquad \text{(B.8)}$$

The formulae required to encrypt and decrypt plaintext using RSA appear relatively simple and easy to implement, but the difficulty lies in the fact that *e*, *d* and *n* are very large numbers. Therefore, the processes involve raising a large number to the power of an equally large number which is time consuming. In fact, the time consuming process of decryption coupled with the recommended key lengths of over 500 bits make a brute force attack on RSA unfeasible.

The security provided by RSA encryption is assumed to be as difficult as factorising ***n***. This is because *n* is known and finding all the known factors of *n* will allow good guesses for the values of *p* and *q*. From this point the cryptanalyst could calculate *(p-1)(q-1)* and determine *d* from *e*. It has never been mathematically proven that *n* must be factorised in order to determine the plaintext from the ciphertext but current approaches to cryptanalysis of RSA tend towards this technique.

Like all Public Key Encryption systems, RSA is liable to chosen plaintext attacks, especially if the format of the message is known or the message is short. RSA recommendations include insertion of random numbers throughout the message to guard against chosen plaintext attacks. There are several other attacks possible on RSA systems that can be guarded against by following a set of rules as listed in Table B-2.

Recommendation	Reason
A common modulus **n** should not be shared in a community of RSA users.	Knowledge of one pair of **e** and **d** for a given **n** enables an attacker to more easily factor **n** and therefore calculate other **e** and **d** pairs for the same **n**.
Plaintext should be padded with random values.	This helps prevent chosen plaintext attacks as it effectively randomises the plaintext where it otherwise might be predictable.
d should be large.	An attack developed by Michael Wiener shows that it is possible to calculate **d** when it is up to one quarter the size of **n**.

Table B-2 RSA Encryption Recommendations

B.2.2.2 Other Public Key Cryptosystems

The Rabin Public Key Cryptosystem gets its security from the difficulty of finding square roots modulo a composite number. This problem is as difficult as factoring. The Rabin method has been extended by Williams to overcome a few shortcomings. Both the original Rabin scheme and the modified Williams scheme can be proven to be as difficult as factoring to cryptanalyse, however whilst encryption is somewhat faster to implement than RSA, execution speeds are still slow.

The ElGamil Public Key Cryptosystem gets its security from the difficulty in calculating discrete logarithms in a finite field. One of the major drawbacks of using ElGamil for encryption of messages is that the ciphertext is twice the length of the plaintext. Encryption and decryption speeds of ElGamil are comparible to RSA.

Other algorithms, namely McEliece and Niederreiter are based on forward error correction codes. Neither of these algorithms have been successfully cryptanalysed and both are relatively quicker than RSA, but both suffer the drawback of the ciphertext being substantially larger than the corresponding plaintext.

Still other algorithms exist that use the properties of elliptic curves or finite state automata. Of course, all Public Key Cryptosystems are interesting in their own right but have the distinct disability of being slower than their private key counterparts. This is why they are used in most systems as a means of either encrypting one-off short messages or to enable secure exchange of a randomly selected private key. The faster private key algorithms are then used to encrypt and decrypt large, continuous streams of data.

B.2.2.3 Recommended Key Lengths for Public Key Cryptosystems

While not all Public Key Cryptosystems are based on the problem of factoring large numbers, the most popular are, and the degree of difficulty in cryptanalysing different Public Key schemes is generally accepted to be equal. With this is mind, we can look at the difficulty of breaking the RSA encryption scheme and relate this to other Public Key Cryptosystems.

A brute force attack on the RSA Encryption scheme involves finding all of the factors of **n** in an attempt to discover the secret prime numbers **p** and **q**. Once all possible pairs of **p** and **q** are found, we can use the fact that the public key is relatively prime to **(p-1)(q-1)** to select the correct **p** and **q** pair and then use the Extended Euclidean Algorithm to determine the private key. Once all possible choices of private keys are found, we can attempt to decrypt the ciphertext to find the correct private key and therefore decipher all future ciphertexts as well. The complex step in this procedure is the factorisation of **n**. Great advances have been made recently in Mathematics in developing faster algorithms to calculate the factors of a large number. Using modern techniques such as the General Number Sieve mean that the length of the key required to provide adequate security using RSA has double from 512 bits to 1024 bits in the last ten years.

The upshot of all this, is that as a result of better techniques to factor large numbers coupled with Moores Law which observes that computing power doubles every 18 months, it is essential to keep increasing the key length in the RSA encryption scheme in order to keep the problem of cryptanalysis of the ciphertext unfeasible. Unfortunately, an increasing key length also increases the complexity and time required to encrypt and decrypt plaintext, therefore keeping encryption speeds relatively constant, regardless of the increase in processing power.

Current RSA systems use a key length of 1024 bits. More important systems however, like certificate authorities, use key lengths of 2048 bits as the information they protect is far more important. Schneier makes the recommendation in his book that a key length of 2048 bits is required today to provide good security for approximately 20 years. This is a far larger key than those used with Private Key Ciphers where key lengths of 64 or 128 bits are the norm. It also puts into perspective the speed of public key ciphers must perform complex mathematical functions on numbers of this size.

B.2.3 The Trusted Authority Public Key Database

We have seen the usefulness of a Public Key/Private Key pair and how it allows us to publicly advertise one key and still allow people to communicate securely with us. If however, Alice wishes to securely communicate with Bob, then she must be sure that she has Bob's Public Key. Bob may have advertised the Public Key but if a malicious hacker can falsely advertise his own Public Key to be Bob's Public Key, then any message Alice encodes for Bob can easily be intercepted and decoded by the hacker. He can then re-encrypt the message using Bob's real Public Key and forward the message to Bob neither partner knows that their communications have been compromised.

The aim in this instance is to ensure that a malicious person cannot impersonate somebody else by falsely advertising Public Keys. A simplistic solution would be to list everybody's Public Key in a printed book such as a Telephone Book, but due to the fact that most Keys are very long, a single misprint or incorrectly typed in number could lead to errors. We can extend the idea of a single printed book by moving the solution and putting the database of public keys online. In this case, Public Keys can be obtained from a central database that maintains a list of all Public Keys. This idea still hasn't solved all potential problems, as the database is now vulnerable to attack, we also have to trust the organisation that supplies and maintains the database. The trust factor is usually solved by using organisations that are be trusted to maintain such information today – governments or banking institutions. To guard against the risk of a direct attack on the centralised database, this database is guarded exceptionally well against both physical and digital attack by potential wrongdoers.

Thus the concept of the Trusted Authority Public Key Database is born. A trusted authority or institution maintains a list of all users and corresponding Public Keys. The repository itself owns its own Private Key/Public Key pair and its Public Key is very well advertised. This key in effect becomes a Well Known Public Key, one that would be impossible to forge or falsely represent because it is so common and known by all. Now if Alice wishes to communicate with Bob, she can obtain his Public Key from the central authority. Alice encrypts her request for Bob's key using the Trusted Authority Public Key. By encrypting the request, Alice knows that only the Trusted Authority, with its Private Key, can decrypt and read the request. The Trusted Authority then obtains Bob's Public Key from the database, formulates a reply, encrypts it using its own Private Key and returns the message to Alice. When Alice receives the message, she decrypts it using the Well-Known Public Key. If the message decrypts correctly, she can be sure that it came from the Trusted Authority since only they could encrypt the message with the Private Key. This message now contains Bob's Public Key and Alice can send any message to Bob knowing that only he can decrypt the message to read the original plaintext.

There is some debate at the moment on how best to implement the Trusted Authority scheme is such a way that it can scale to service the entire online community. However, it seems certain that in the near future the use of Public Key Cryptography will become commonplace, and Trusted Key Databases will play a major role in secure communications on the Internet. As such, these databases will form an integral part of the future of telecommunications and be widely available.

B.2.4 Amenability to Encryption of Streaming Multimedia

The speed of Public Key Encryption is one of the two major drawbacks to using this scheme to protect streaming multimedia. Video material encoded using MPEG-1 is usually encoded at rates between 1.5 Mb/s and 2.0 Mb/s whereas Public Key Encryption schemes encrypt data at the rate of hundreds of kilobits per second or slower. While the speed of computers and computing power is increasing all the time, the required key length to ensure security also increases, thereby ensuring that encryption rates remain relatively constant.

This observation is enough on its own to discount Public Key Encryption in the field of encrypting streaming multimedia, but there is another valid reason to discount it. All Public Key Encryption schemes have one thing in common with Private Key Block Ciphers, and that is that they operate on the plaintext in blocks of a fixed size. If the data is not a multiple of the block size, then the plaintext must be padded to make it the required length. To develop an encryption system that will function correctly with existing MPEG video servers requires a scheme where we can selectively encrypt individual bytes in the multimedia stream. This will allow us to ensure that the length of the encrypted MPEG bitsteam is the same as the length of the plaintext MPEG bitstream.

These two facts in conjunction rule out the use of Public Key Encryption to protect the actual streaming asset as it is transmitted across the network. It does not, however, rule out its use in a key management scheme to transmit a private decryption key to the viewer of the asset. The use of Public Key Encryption and the Diffie-Hellman algorithm would be quite valid in both validating the user to view the encrypted stream and then for transmitting the private key required to both decrypt and view the streaming asset.

B.3 Private Key Cryptography

Private Key Cryptography involves a single key, which must be known by both parties in order to ensure secure telecommunications. The same key is utilised to both encrypt and decrypt any data that needs to be communicated securely. Private Key Cryptography is the oldest known form of cryptography and can be traced back to some of the oldest ciphers used, including a simple rotated alphabet scheme used by the Romans in which each textual character was replaced with the character three places to the right in the alphabet. In this case, the algorithm involves using a shifted alphabet and the Secret Key is the number of characters we shift the alphabet through to both encrypt and decrypt the message. Obviously a system like this is very insecure, especially given knowledge of the algorithm used, it would be a simple matter of trying each of the 26 possible Secret Keys in a brute force attack until a meaningful message can be retrieved from the ciphertext.

Private Key Cryptographic systems fall into two distinct classes, Block Ciphers and Stream Ciphers. A Block Cipher operates on a block of plaintext of a fixed size and produces a block of ciphertext, usually of the same size but possibly longer. Similarly, the decryption algorithm takes a block of ciphertext to produce the original block of plaintext. On the other hand a Stream Cipher operates on both the plaintext and ciphertext one bit at a time. Each bit of the plaintext is encrypted to form a single bit of ciphertext, at the destination, each ciphertext bit is then decrypted by the Stream Cipher to obtain the original plaintext bit. In general, Private Key Cryptographic systems operate much faster than their Public Key counterparts, also the required Key Length to ensure secure communications is much shorter. With this knowledge in mind, it is easy to see why most applications that require a sizeable amount of data transfer utilise Private Key algorithms to implement their communications.

B.3.1 Block Ciphers

A Block Cipher is a form of encryption algorithm that operates on the input data in blocks of a fixed size. The most common Block Ciphers in use today operate on input data in blocks of 64 bits. A Block Cipher takes a plaintext block of a specified size and an encryption key, and then operates on this data to produce a ciphertext block of the same size. Similarly, the decryption algorithm takes as input the fixed size ciphertext block and the decryption key, identical to the encryption key. It then performs the function of retrieving the plaintext block of data. The nature of this description necessarily implies that a particular block of plaintext will always encrypt to the exact same ciphertext block given the same encryption key.

Using a Block Cipher in this way is known as using the cipher in Electronic CodeBook (ECB) Mode. There are other modes in which we can utilise a Block Cipher which provide us with various advantages and disadvantages, the other modes are alternatively known as Cipher Block Chaining (CBC) Mode, Cipher Feedback (CFB) Mode and Output Feedback (OFB) Mode.

B.3.1.1 Block Cipher Modes

In Electronic CodeBook or ECB Mode, the Block Cipher algorithm is used in its most obvious and simple way. If the algorithm we are using operates on input data in blocks of n bits, then we break our plaintext M into segments M_i of exactly n bits in length. Each one of these n-bit segments is then encrypted using the chosen cipher to produce a corresponding n-bit block of ciphertext C_i. If the same n-bit block of plaintext is encrypted at some later stage, the same block of ciphertext will be created. If a cryptanalyst wanted to, they could encrypt every possible block of plaintext with a key to produce every corresponding ciphertext block, these results could then be stored in a codebook and matched against ciphertext to decode and reproduce the original plaintext. While this may seem like a weakness of this mode, given that most Block Ciphers operate with 64 bit blocks, there are a total of 2^{64} possible different blocks of plaintext. Storing this information would require an extreme amount of storage space along with exceptional processing power to access and search the database.

Whilst using a Block Cipher in ECB Mode is secure against a full codebook attack, a cryptanalyst can still cause damage with a partial codebook attack where a message can be partially decoded due to the nature that the beginning and ending of most messages are predictable. Similarly, a strictly formatted message containing money transfer information could be easy to forge and manipulate if the cryptanalyst knows which ciphertext blocks contain destination account information.

One of the advantages of using a Block Cipher in ECB Mode is that we have random access into the encrypted data. Because each block of plaintext is encrypted independantly, we can decrypt a block of ciphertext independently of any other block. In effect, we could extract a block of ciphertext from the middle of the message to decrypt and retrieve the corresponding block of plaintext at that point of the message. This is significant when considering encrypting streaming multimedia files as random access into the streamed video is considered a major feature of digital video, as such, we must be able to start decrypting and playing back the streamed data from any point in the stream.

A Block Cipher being used in ECB Mode means that by necessity, the length of the plaintext must be an exact multiple of the block size. This is usually performed by padding the plaintext with zero bits so that it is of the correct length. Obviously, what this means is that the modified plaintext is of different length to the original plaintext and by default that the ciphertext is of different length to the original plaintext. There are many situations where this is not a problem, unfortunately in the case of encrypting streaming data, the length of the data to be encrypted is variable and this Thesis shows that it is essential that the ciphertext be of the same length as the plaintext.

The second type of Block Cipher mode is referred to as Cipher Block Chaining or CBC Mode. This mode is very similar to ECB Mode except that a feedback mechanism is added such that the block of ciphertext produced by the algorithm is a function of not only the original block of plaintext, but also the previous block of ciphertext. In effect, the same block of plaintext will not always encrypt to a given block of ciphertext. However, two separate messages that have the exact same beginning will encrypt to the same ciphertext stream up until the point where the messages become different. In order fix this, the CBC Mode cipher will often seed the algorithm with a random key block as the first block to be encrypted. The receiver will then decrypt this key block and throw it away. The key block need not be secret as it is merely used to ensure that if the first real block of plaintext is common, it will not always encrypt to the same block of ciphertext.

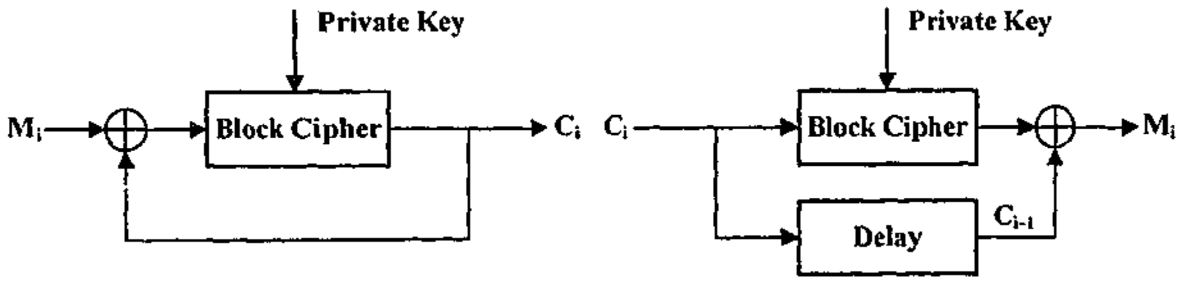


Figure B-4: CBC Mode Encryption/Decryption Process

These additions make the job of the cryptanalyst more difficult at the expense of removing the ability to provide random access into the encrypted stream. Unfortunately, however, the modifications to ECB mode still mean that the length of the plaintext must be a multiple of the cipher block size. The exact technique to use a Block Cipher in CBC Mode is demonstrated in Figure B-4. The first block of plaintext, or more usually the key block, is passed through the cipher to produce the first block of ciphertext, this block of ciphertext forms part of the encrypted message for transmission and is designated C_1. The next block of plaintext, M_2, is then XORed with the previous block of ciphertext C_1, the resultant modified plaintext is then passed through the cipher to produce the next block of ciphertext C_2. This process is repeated with plaintext block M_i being XORed with ciphertext block C_{i-1} before being passed through the cipher to produce C_i. The receiver of the encrypted message receives the blocks of ciphertext in order. They first pass C_1 through the cipher to reproduce the first block of plaintext, usually the key block which would be discarded, M_1. When the next block of ciphertext C_2 is received, it is passed through the cipher and then XORed with C_1 to produce M_2. This process is continually repeated with ciphertext block Ci passed through the cipher before being XORed with ciphertext block C_{i-1} to reproduce M_i.

Block Ciphers can be modified to work on block sizes that are smaller than those specified by the actual encryption algorithm; the final two Block Cipher Modes work in this fashion. The first of these modes, Cipher Feedback or CFB Mode, uses a large block size Block Cipher and its inherent security to implement a self synchronising system which operates on a smaller block size. This technique removes the need of the length of the plaintext being a multiple of the block size. In CFB Mode, a key block is used to store the current state of the system. This key block is passed through the cipher to produce an *n*-bit cipherblock. As required, the leftmost *m* bits (where *m* is less than *n*) are then XORed with the next *m* bits of the plaintext to produce the next block of ciphertext of size *m*. The state block is then left shifted by *m* bits and the recently produced block of ciphertext is inserted into the recently vacated bits in the state vector. This new state vector is then encrypted to produce another cipherblock of which the leftmost *m* bits are XORed with the plaintext. At the receiver end, the same initial state vector is encrypted to form a cipherblock. The leftmost *m* bits of this block are XORed with the ciphertext block to retrieve the plaintext block. The state vector is then left shifted by *m* bits and the ciphertext block is inserted into the recently vacated bits in the state vector. An interesting aside is that after the first few ciphertext blocks are decrypted, the state vector is made up entirely of the last *n* bits of the ciphertext. The system is called self-synchronising because if enough random bits are inserted before the plaintext at the encryption end, then the state vector will automatically synchronise to the ciphertext and the initial state is unimportant. This process is demonstrated in Figure B-5.

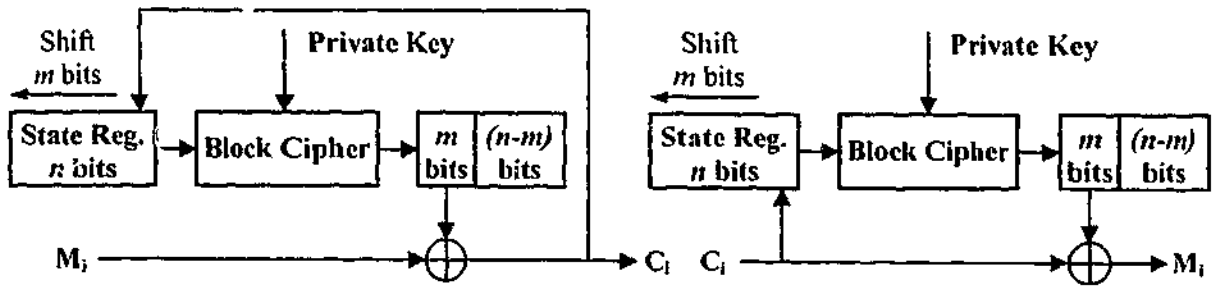


Figure B-5: CFB Mode Encryption/Decryption Process

A Block Cipher running in CFB Mode removes the constraints on the length of the plaintext. It also allows almost random access into the encrypted stream as long as we are prepared to accept a few blocks of incorrect ciphertext. This makes this technique almost suitable for encryption of streaming multimedia. For better suitability, we should consider a Block Cipher running in Output FeedBack of OFB Mode. In OFB Mode, we build a system very similar to that used in CFB Mode except that the state vector is modified based on the cipherblock at the output of the Block Cipher instead of the ciphertext block at the output of the entire system. What this means is that the feedback mechanism is entirely internal to the system and we are in effect producing a system which generates pseudo random *m* bit blocks which are then XORed with the plaintext, this is shown in Figure B-6. As for CFB Mode, we have a system where the internal state vector is automatically updated based on the initial key block used to key the system. As a further advantage, we now have a system whereby a single bit error in the ciphertext will not propagate through a few blocks of recovered plaintext. Also, we maintain the ability to encrypt the plaintext in smaller block sizes, down to a possible single bit if necessary. In the implementation, the bits used to fill the empty space in the state vector come from the output of the block cipher instead of the produced ciphertext itself, otherwise the implementation is exactly the same as for CFB Mode.

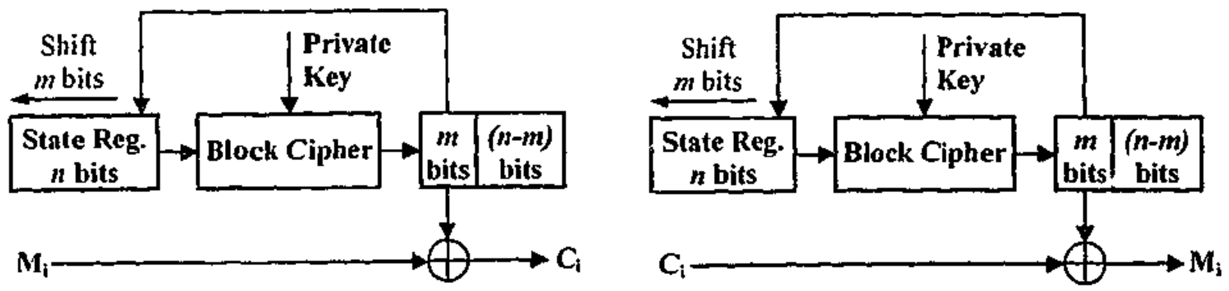


Figure B-6: OFB Mode Encryption/Decryption Process

The last two modes of operation actually utilise a Block Cipher in a fashion similar to a Stream Cipher where the plaintext is XORed with a pseudo random string of bits except that the XOR function is performed on a block larger than one bit in size. In fact, if desired, a Block Cipher can be used in CFB or OFB Mode reduced to one bit blocks and therefore become a true Stream Cipher. All Block Encryption algorithms can be utilised in this way, however their speed effectiveness will be reduced. Each operation to encrypt a block normally of *n* bits is now being used to encrypt a smaller block and therefore more operations are required to encrypt a block of plaintext of the same size. A summary of the advantages and disadvantages of the possible modes of Block Cipher usage can be found in Table B-3.

B.3.1.2 Principles of Block Cipher Cryptology

The general principles used in most Block Ciphers can be summed up using the terminology confusion and diffusion. Confusion, in relation to encryption, attempts to obscure any relationship between the plaintext and the ciphertext. This can usually be performed with simple substitution of bits within the plaintext message, in fact the One Time Pad Cipher is a confusion based cipher where each bit in the plaintext is either inverted or left untouched depending on the random key used. Diffusion, in relation to encryption, tries to remove any redundancies and statistical relationships in the plaintext by spreading the effect of the plaintext over as much ciphertext as possible. In reference to an n-bit block cipher, diffusion would attempt to spread the effect of a single bit of plaintext over all n bits of the ciphertext block produced by the cipher. A strong cipher would succeed in this whilst an insecure cipher would not be able to ensure that each plaintext bit affected all ciphertext bits in the block. Ciphers that employ diffusion techniques only are not particularly secure and can be easily cryptanalysed and broken. Most Block Ciphers use both confusion and diffusion techniques in their algorithmic designs.

Cipher Attributes	ECB	CBC	CFB	CBC
Concealment of plaintext patterns.	☒	☑	☑	☑
Randomisation of input to block cipher.	☒	☑	☑	☑
Security in reusing the same key.	☑	☑	Needs unique start register.	Needs unique start register.
Encryption Speed.	Same as block cipher.	Same as block cipher.	Slower than block cipher by a factor of n/m.	Slower than block cipher by a factor of n/m.
Length of Ciphertext equal to length of plaintext.	☒	☒	☑	☑
Fault tolerance of ciphertext error.	Affects one full block of plaintext.	Affects one full block of plaintext and the corresponding bit in the next block.	Affects corresponding bit of plaintext and the next full block.	Affects corresponding bit of plaintext.
Fault tolerance of bit loss/insertion error.	Unrecoverable.	Unrecoverable.	Recoverable if $m = 1$.	Unrecoverable.

Table B-3 Summary of Block Cipher Modes of Operation

Common features of Block Ciphers are called S-Boxes or Substitution Boxes. These modules can usually be defined as a black box that takes a range of bits as input and produces a range of bits as output. The S-Boxes in a Block Cipher are responsible for confusion of the data as they substitute bits at their input with different bits at their output. The S-Boxes used in the popular DES algorithm take two inputs, a four bit input which will be substituted by the S-Box and a two bit input which is used to select one of the four substitutions performed by the S-Box. The chosen substitution will then produces the four bit output of the S-Box. Obviously, the S-Box in itself is rather insecure and corresponding input is relatively simple to determine, this is why the S-Boxes usually form part of a more complex system which will form the entire Block Cipher. The transformation functions of the S-Boxes, whilst usually known, must also be carefully chosen such that the interaction of the S-Box with the rest of the Block Cipher will ensure that the ciphertext is adequately randomised and that statistical properties of the plaintext will not be evident in the ciphertext.

Another common operation of a Block Cipher is a simple permutation. This contributes to diffusion of the data over the entire cipherblock. Simple permutations often involve completely reordering the input bits to produce a new value of the same length. In these cases no substitution takes place as there are exactly the same number of one bits and zero bits as before the operation. This operation spreads the bits over the length of the cipherblock, ensuring that any subsequent operation within the Block Cipher will allow input bits to affect each output bit of the cipherblock. As for the S-Boxes, how the bits are reordered in a simple permutation are usually carefully chosen to ensure that good diffusion takes place. In some Block Ciphers, the permutation step may choose the reordering sequence based on the key rather than being fixed.

Another commonly utilised building block is the Expansion Permutation of input bits. In this case, not only are bits reordered as in a simple permutation, but some input bits are also repeated or combined in order to form extra output bits and create a larger output than the input. The main aim of this module is in diffusion and to help spread the input bits over the output much more quickly. However, it is also useful in expanding a data set in the middle of a cipher operation so that more bits are available to run through other modules such as S-Boxes, this step can also be useful to expand the input to the same size as the key for an XOR operation.

Other common operational modules in a Block Cipher include XOR, bit-shift operations and plaintext splitting. An XOR operation involves a simple XOR between two sets of input bits, one set of which is usually related to the key. A bit-shift operation involves shifting a set of input bits through a set number of operations, this operation can be performed on both the input plaintext itself and the key between steps of the cipher algorithm. Plaintext splitting involves breaking the input block into two and performing different operations on each half the input bits.

Finally, most Block Ciphers involve iterations of repeated steps. In most block ciphers, the above modules are usually integrated in a set configuration to form a relatively simple step, this step is then normally repeated a fixed number of times. By repeating the step, it makes the overall algorithm more difficult to analyse as each step further confuses and diffuses the input plaintext. If enough iterations are performed, we can ensure that full diffusion of the plaintext to the ciphertext occurs and that multiple iterations of the non-linear S-Boxes cause statistical properties which make analysis difficult. Using DES as an example, each step of the cipher process is repeated 16 times, this repetition ensures good diffusion and 16 repetitive applications of the S-Boxes creates ciphertext with strong confusion properties.

Many Block Ciphers are Feistel Networks. A Feistel Network follows a simple design principle that ensure that exactly the same algorithm can be used in both encryption and decryption processes. In a Feistel Network, the n-bit input plaintext is broken into two equal length halves of size *n*/2. Each step of the Feistel network now involves passing the right half of the plaintext through a function f() and then XORing it with the left half of the plaintext, this result becomes the new right half of the plaintext whilst the original right half of the plaintext becomes the new left half of the plain text. The iteration process of block ciphers now occurs as many times as is desired. The calculation and swapping occurs for each iteration except the last, where the two halves are not exchanged. This allows the entire process to be completely reversible using the same algorithm, regardless of how complex or irreversible f() happens to be. This works for the following reasons: if there are *m* iterations involved in the Block Cipher, and the plaintext is originally broken up into two halves $\mathbf{L_0}$ and $\mathbf{R_0}$, then the formulae the left and right halves at each iterative step of the cipher are:

$$L_i = R_{i-1} \tag{B.9}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \tag{B.10}$$

Where $\mathbf{K_i}$ is the key or permutation of the key used at this step of the cipher. These formulae hold for all but the last step of the cipher as the halves are not swapped, in this case the values of $\mathbf{L_m}$ and $\mathbf{R_m}$ are:

$$L_m = L_{m-1} \oplus f(R_{m-1}, K_m) \tag{B.11}$$

$$R_m = R_{m-1} \tag{B.12}$$

As long as our key schedule is used is such a way that $\mathbf{K'_i = K_{(m+1)-i}}$, the exact same algorithm will work in the decryption process. This can be shown assuming that the cipherblock is broken into the original two halves $\mathbf{L'_0}$ and $\mathbf{R'_0}$ and the keys are $\mathbf{K'_i}$. In the first iteration $\mathbf{L'_0}$ is equal to $\mathbf{L_m}$ and $\mathbf{R'_0}$ is equal to $\mathbf{R_m}$. The formulae for the left and right halves after the first iterative step of the cipher are:

$$L'_1 = R'_0 = R_m = R_{m-1} \tag{B.13}$$

$$R'_1 = L'_0 \oplus f(R'_0, K'_1) = L_m \oplus f(R_m, K_m) = L_{m-1} \oplus f(R_{m-1}, K_m) \oplus f(R_{m-1}, K_m) = L_{m-1} \tag{B.14}$$

Similarly, the formulae for the left and right halves after the second step become:

$$L'_2 = R'_1 = L_{m-1} = R_{m-2} \tag{B.15}$$

$$R'_2 = L'_1 \oplus f(R'_1, K'_2) = R_{m-1} \oplus f(L_{m-1}, K_{m-1}) = L_{m-2} \oplus f(R_{m-2}, K_{m-1}) \oplus f(R_{m-2}, K_{m-1}) = L_{m-2} \tag{B.16}$$

Which generalise to:

$$L'_i = R_{m-i} \tag{B.17}$$

$$R'_i = L_{m-i} \tag{B.18}$$

Remembering that the halves are not swapped in the last step of the cipher, we can show:

$$L'_m = L_{m-m} = L_0 \tag{B.19}$$

$$R'_m = R_{m-m} = R_0 \tag{B.20}$$

And we have retrieved our original plaintext. This useful property of Feistel Networks explains why they are often used when designing Block Ciphers.

B.3.1.3 Weaknesses and Practicalities

There are many Block Cipher algorithms from which to choose from, each with their own individual strengths and weaknesses. Many ciphers are currently considered to be secure, meaning that a brute force attack is the only known viable attack. This means that as long as this requirement is taken into consideration, and the keyspace size is suitably large, the choice of which cipher to use is academic. What is far more important after selecting the actual cipher to use, is to determine in which mode we wish to utilise the cipher.

There are some concerns about implementation speed of block ciphers, the same argument with Public Key Ciphers and increased computing power is valid. However, it is entirely feasible that real time encryption/decryption of data at the required bit rates can be accomplished by most block ciphers without many concerns, but note that Stream Cipher implementations are far quicker than Block Ciphers and should therefore be considered as an alternative to block ciphers. It is imperative to consider the required CPU power to perform the decoding and displaying of the MPEG media stream. Similarly, if we want to encrypt a bytestream without increasing the length of data to be transmitted, we can only consider using a block cipher in CFB or OFB Mode. In either of these modes, running a 64 bit block cipher in 8 bit mode will increase the execution time by a factor of eight.

Block Ciphers are amongst the most difficult algorithms to cryptanalyse, mainly because they contain non-linear properties. For this reason many algorithms are determined to be very secure and susceptible only to a brute force attack. On the other hand, the mathematical properties of Stream Ciphers are well known and easier to cryptanalyse. However, it is conceivable that future study into Block Ciphers will yield better knowledge and lead to better attacks on algorithms. This may show that Block Ciphers will not be as secure as once thought. This should not discourage use of Block Ciphers as they offer extremely good security given today's knowledge.

B.3.1.4 DES Block Cipher

The DES (Data Encryption Standard) Block Cipher algorithm was first published in 1976 and was the first government approved encryption standard. Politically this caused many problems as people assumed that the algorithm was cryptographically insecure as they assumed that government agencies would only approve a cryptographic algorithm with a backdoor in it. However no backdoor to DES has been found to date. DES is now beginning to date and the security that it affords is no longer adequate for many applications. When considering the application of protection of streaming media, we have to look at the value of the media compared to the cost of acquiring that media illegally.

DES is an example of a Block Cipher implemented as a Feistel Network. Feistel Networks were described in the previous section as a means whereby we could continuously apply any complex function to a set of input data and yet still guarantee to be able to easily reverse the entire process. The process uses a 56 bit key which makes the total keyspace of over 7×10^{16} possible private keys. Whilst this may sound like a lot, this keyspace is not large enough to prevent a simple brute force attack on the ciphertext to retrieve the plaintext. The Fiestel Network in the DES encryption algorithm uses a complex function f() with a changing key for each cycle of the implementation. Standard DES performs the cycle a total of 16 times. The f() function takes as input the stage key and the 32 bits forming the righthand side of the plaintext. These inputs are then put through phases of expansion permutation, a series of S-Boxes and a simple permutation. Added on to the standard Feistel Network implementation is one extra simple permutation at the start of the encryption process and its inverse permutation at the end of the process. Since this simple permutation is known, it adds no security to the actual DES implementation itself. The belief is that this permutation is included to simplify loading the 64 bit plaintext block into the internal shift registers using a hardware design of the 1970's.

Given this brief description, a DES encryption module looks very much like the block diagram depicted in Figure B-7, remembering that the left and right halves are not swapped at the end of the last stage of the network. The initial permutation, **IP**, simply rearranges the bit ordering of the plaintext block according to a fixed transform, while the final permutation, **IP**$^{-1}$, is the inverse function of **IP**.

Within each stage of the Feistel Network, we see that the input block, $\mathbf{R}_{i-1}$, of 32 bits in passed through the f() function to provide a output block which is to be XORed with the left hand input block $\mathbf{L}_{i-1}$ to form $\mathbf{R}_i$. The breakdown of the f() function within the DES Feistel Network is presented in Figure B-8. This diagram demonstrates the function taking an input block, $\mathbf{R}_{i-1}$, of 32 bits in length and stage key, $\mathbf{K}_i$, of 48 bits in length, and produces an output block of 32 bits in length. There are four stages in the f() function, the first of these is an expansion permutation whereby the input block of 32 bits is expanded to be equal to 48 bits in size. The exact details of the expansion permutation can be found in [reference] but the permutation has been designed in a way to assist in the third stage of f() which hosts the S-Boxes. In the expansion permutation, each block of 4 bits is kept intact with two new bits being inserted between each of these blocks. These new bits are determined from the values of the first or last bits of the blocks that they divide. The second stage of the f() function involves the interaction with the stage key. At this point, the 48 bit value derived from the expansion permutation is XORed with the stage key to provide a new value which will be used in the third stage of f().

The third stage of the f() function comprises of 8 different S-Boxes, each of these S-Boxes takes two inputs to produce a single output. The S-Box performs a transform of one 4 bit value to another 4 bit value with the second input of two bits being used to select from one of four possible transforms. The 48 bit output from the previous stage is divided into eight groups of six bits of which two are used to select the S-Box transform and four are used as input to each S-Box. Looking back at the expansion permutation of the previous stage, the two transform selection bits are taken from the four input bits of the surrounding groups of four bits. Since each S-Box produces a 4 bit output, the output from this stage of the f() function is of 32 bits in length, back to the original size. Each of the eight S-Boxes performs a different set of transforms, the details of the transforms provided by each of the S-Boxes can be found in [reference].

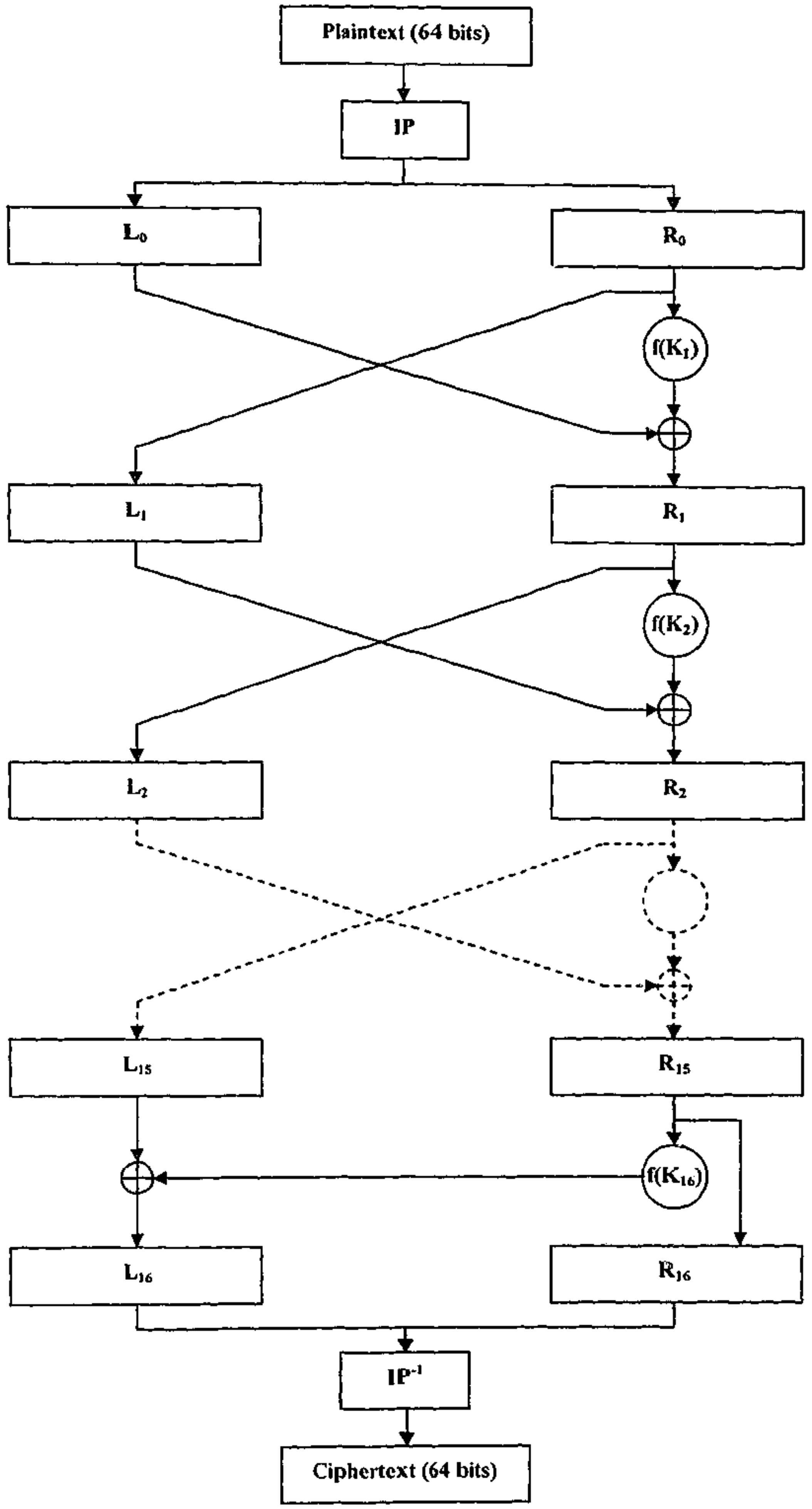


Figure B-7: DES Encryption Algorithm

The final step of the f() function is just a simple permutation where the locations of the 32 output bits of the previous step are rearranged. The purpose of this step is to spread out the bits so that each bit causes a greater effect over a larger range of output bits in the cipherblock, assisting us to perform our intended goal of introducing diffusion into the encryption algorithm. The final step involved in implementing a DES encryption algorithm is the key schedule, or determining how to calculate the 48 bit stage key that will be used during each cycle of the Feistel Network. The key scheduling algorithm for DES is shown in Figure B-9. I have already mentioned that the DES private key is 56 bits in length, for each cycle through the Feistel Network, this key is split into two halves of 28 bits each. Each of these 28 bit values is then circularly left shifted through either one or two bits, depending upon the current round of the algorithm. Following the key shift, the new 56 bit key is then passed through a compression permutation where the order of the bits are rearranged according to a fixed transform and 8 of the bits are ignored in order to form a 48 bit key. This 48 bit key is then used as the stage key for each cycle through the Feistel Network, the new 56 bit key produced as a result of the circular shifts is used to calculate the stage key for the next cycle of the Feistel Network. The details of how the circular shifts occur and the transform used in the compression permutation can be found in [reference].

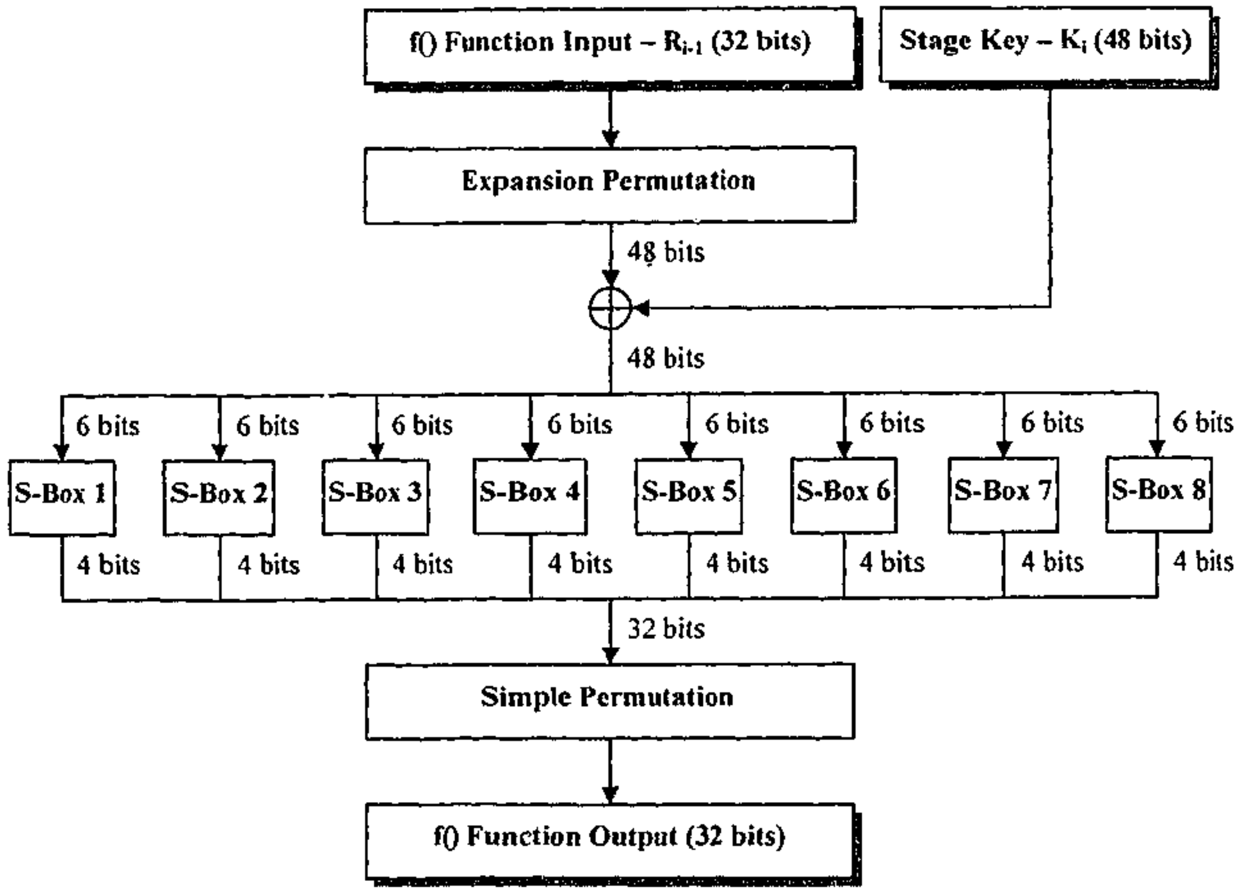


Figure B-8: DES Fiestel Network f() function

Because of the fact that the DES encryption algorithm forms a Feistel Network, decryption is extremely simple. The initial and final transforms undo their counterpart steps performed during the encryption process, the Feistel Network takes care of reversing the encryption stages as long as we feed the stage keys in the reverse order. In this case, K_{16} becomes the first stage key and K_1

becomes the final stage key used in the decryption process. The schedule of circular shifts selected in the encryption algorithm are chosen such that the final 56 bit key will be equal to the original key. As such, we can reverse the key schedule by performing a series of circular right shifts at each stage to counteract the left shifts used in the encryption process. Therefore, the only differences between the DES encryption and decryption algorithms are minor changes in the key scheduling algorithm, which can be implemented simply.

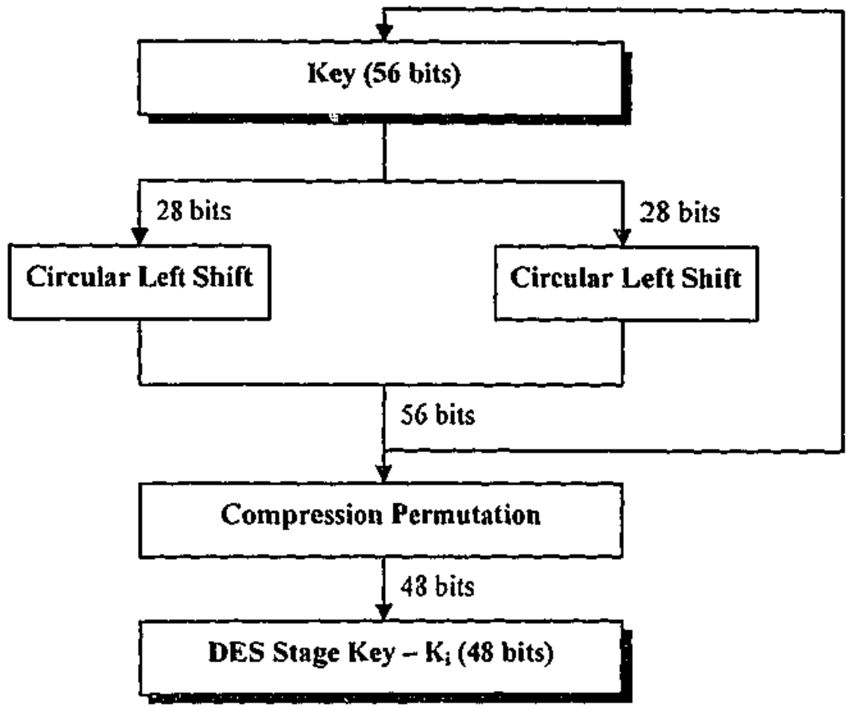


Figure B-9: DES Key Schedule

B.3.1.5 DES Variations

DES is an extremely good block cipher that has proven to be resistant to practically all statistical attacks since its introduction in 1976. In fact, the best cryptanalysis techniques have only reduced the complexity of an attack from the brute force technique of 2^{56} to the order of 2^{55}. This has proven the strength of the DES cipher as a whole but it is now being defeated, not to a deficiency in its design, but to the increasing computing power that is now available. Put simply, the key space afforded by DES, 56 bits, is too small and a brute force attack on a DES encrypted message is becoming possible. In fact, as long ago as 1993, it was calculated that a specialised machine employing brute force to crack a DES encrypted message in 3.5 hours could be built for one million dollars. Given the time that has passed since then it is reasonable to assume that the cost and speed of this machine would have improved greatly. There are a number of variations on DES which are discussed in following paragraphs, however, the American National Institute of Standards (NIST) has recently called for new algorithms to replace DES as the standard encryption algorithm.

Most variations on DES were originally proposed because of perceived fears of weaknesses in DES. These fears originally came about because the NSA was involved with the design of DES and artificially introduced weaknesses into the algorithm to make ciphertext easier to decode. Cryptanalysis of DES has not found any of these concerns to be proven. Most of the proposed variations are concerned with either changing the DES S-Boxes, the permutation modules, the key scheduling algorithm, or a mixture of these. Suggested changes in the S-Boxes involved changing their ordering, changing their transform functions, or selecting S-Box transforms based on the key. Proposed changes to permutation steps simply involved reordering the bit permutations and changes to the key scheduling algorithm are rather obvious. Most of these proposed changes statistically weakened the DES encryption algorithm, those that didn't also did not improve the strength of DES in any way. The one technique that did increase the strength utilised a completely different stage key for each cycle, these 16 48 bit keys were combined to form a single 768 bit private key used to encrypt the message. This extreme size of private key added a great deal of strength to the encryption algorithm at the expense of using a larger key.

Other DES variations that have been proposed involve using the existing DES algorithm more than once. Using DES twice on a block of plaintext, ie. encrypting the plaintext using one key and then encrypting it again using a second key, can be proven to add no more than a factor of 2 strength to a simple DES implementation. The most common multiple DES variant has been triple-DES, where the plaintext is encrypted with one key, decrypted with a second key and then re-encrypted with the first key again. This leads to a very strong encryption algorithm with a combined key length of 112 bits. One of the major drawbacks involved in using triple-DES is that processing time is tripled for both the encryption and decryption cycles.

DES is a very strong encryption algorithm that is losing its strength due to its relatively small key size. Many people advocate using a DES variant, even if it offers less statistical security, due to the cheapness in building a DES cracking machine. If speed of encryption/decryption is not a concern then triple-DES is a good solution. A new standard to replace DES is anticipated shortly and promises a much larger key space to protect against increases in computing power.

B.3.1.6 Recommended Key Lengths for Block Ciphers

When we look at recommended key lengths for Block Ciphers, the other factor we have to consider is how good is the encryption algorithm in the first place. The DES algorithm is considered very strong because a brute force attack is close to the best possible attack we can run on DES. In this section I discuss recommended key lengths for Block Ciphers, assuming that a brute force attack is the best option open to a cryptanalyst. While the 56 bit key employed by DES is inadequate for secure implementations today, a 112 bit key should still be secure in 40 years time.

The actual time taken to encrypt a plaintext block will vary from algorithm to algorithm, but since the time taken to employ brute force attack is exponentially dependant on the key length, the small constant factor that differentiates different algorithms is insignificant. Let us assume that we will attempt to decrypt the ciphertext using a unique hardware solution. Given a chip that can encrypt 10 million plaintext blocks a second, this chip could check each of the 2^{56} keys of a DES cipher in $2^{56} / 10^{7} = 7.2 \times 10^{9}$ seconds = 8.3×10^{4} days. This is still an extremely long period of time for a very fast encryption chip. However, the problem of decryption is very parallelisable in that we can have multiple encryption chips, each trying a different subset of keys. Consider a machine with one million of the aforementioned chips executing in parallel: it can check each of the 2^{56} keys of a DES cipher in $2^{56} / 10^{13} = 7200$ seconds = 2 hours. A 56 bit key can be easily cracked by such a dedicated machine. Even if the decryption phase took three times as long to perform, the time involved would triple to a still feasible 6 hours. Also, this solution is inherently scalable, so we could build a machine with two million such processors that could check all possible keys within one hour. Also, given that we do not know which key is the correct one, we will stumble across the correct key on average half way through the process, so the mean time to decrypt a DES encrypted message would halve again to 30 minutes.

Now continue to assume a hardware solution of one million processors capable of checking ten million keys per second. Also assume Moore's law continues to hold, ie. that computing power will double every eighteen months (increase by a factor of ten every five years). A series of results for the mean time taken to determine the correct decryption key is shown in Table B-4. Our 56 bit key will be decoded in an average time of 1 hour, while a 64 bit key yields a time of 10.7 days. A longer key of 128 bits requires over 10^{17} years to determine the key, providing very strong security.

Timeframe	56 Bits	64 Bits	80 Bits	112 Bits	128 Bits
Now	1 hr	10.7 days	1917 yrs	8×10^{12} yrs	5×10^{17} yrs
+ 5 years	6 mins	1.07 days	191.7 yrs	8×10^{11} yrs	5×10^{16} yrs
+ 10 years	36 s	2.6 hour	19.2 yrs	8×10^{10} yrs	5×10^{15} yrs
+ 15 years	3.6 s	15.4 minutes	1.9 yrs	8×10^{9} yrs	5×10^{14} yrs
+ 20 years	0.4 s	1.5 minutes	70 days	8×10^{8} yrs	5×10^{13} yrs
+ 25 years	40 ms	9.2 seconds	7 days	8×10^{7} yrs	5×10^{12} yrs
+ 30 years	4 ms	0.9 seconds	16.8 hrs	8×10^{6} yrs	5×10^{11} yrs
+ 35 years	0.4 ms	90 ms	1.7 hrs	8×10^{5} yrs	5×10^{10} yrs
+ 40 years	40 μs	9 ms	10 mins	8×10^{4} yrs	5×10^{9} yrs
+ 45 years	4 μs	0.9 ms	1 min	8000 yrs	5×10^{8} yrs
+ 50 years	0.4 μs	90 μs	6 s	800 yrs	5×10^{7} yrs

Table B-4 Block Cipher cracking times based on key length

In 50 years time, a 128 bit key will still require over 10^{7} or 50 million years of processing to determine the key, offering more than adequate levels of security. On the other hand, a 64 bit key will be cracked in 90 microseconds and an 80 bit key in 6 seconds. These figures are also valid in estimating a massively parallel effort involving a number of computers around the world. Assuming we could co-opt 300 million computers to test 1 million keys every second, the figures in Table B-4 would be out by a factor of 30 and 1.5 million years is still a long time to wait to crack a 128 bit key.

The actual choice of key length is not as important as ensuring that a brute force attack is the best attack on your Block Cipher. Given a good Block Cipher, a key length of 80 bits would be adequate for non sensitive data whilst a key length of at least 112 bits is required to protect sensitive data for the long term. In the case of dealing with multimedia on a network, the data is likely to be valuable for a small number of years, its value decreasing with time. If a Block Cipher is chosen, one with a key length of 80 bits would seem adequate, however, most new Block Ciphers being developed utilise key lengths of either 112 or 128 bits.

B.3.1.7 Amenability to Encryption of Streaming Multimedia

The question of how suitable Block Ciphers are in encryption of streaming multimedia is a more difficult question to answer than that of Public Key ciphers. Certainly, Block Ciphers are able to encrypt and decrypt data at the rates required of streaming MPEG-1 or MPEG-2. Indeed, a dedicated chip encrypting 1 million blocks a second could encrypt at a rate of about 60 Mb/s. However, it is also important to consider that a software system would not only encrypt/decrypt data at a slower rate, but also be required to have remaining CPU cycles to decode and display the video in real time. This implies that the amount of processing power available to decrypt the streaming data is minimal and the encryption scheme employed has to take this into account.

It is also interesting to note that unless the Block Cipher is running in CFB or CBC mode, the length of the ciphertext will not be equal to the length of the plaintext. Chapter 4 shows that the optimal technique involves partial encryption of the MPEG stream. As such, the desired system will need to encrypt the plaintext blocks using a grain size of 8 bits. If a Block Cipher is to be considered in either of the above two modes, then the decryption efficiency will be lowered by a factor of eight, causing a heavy load on the CPU cycle budget. If we are considering performing the encryption on the server in real time, then each stream being delivered will need to encrypted causing a potential high load on the server. The amount of CPU power required to perform this task becomes even more critical on the server that will be responsible for encrypting many streams of data. The calculations involving the CPU cycle budget become more complicated when looking at this issue.

It is certainly possible to employ Block Ciphers in an MPEG partial encryption scheme. However, requirements on maintaining equal plaintext length and total CPU cycles available to perform the process mean that a Block Cipher remains a difficult choice. Considering the CPU processing requirements of decoding and decrypting an MPEG stream, it would be more prudent to consider a Stream Cipher which runs at greater speeds than a Block Cipher.

B.3.2 Stream Ciphers

Stream ciphers are a class of encryption algorithm that try to emulate the properties of the One Time Pad encryption cipher. From Section B.1.3, a One Time Pad cipher uses a true random string of bits of the same length as the plaintext. This random string of bits is then combined with the plaintext via an XOR function to form the ciphertext. If the One Time Random Pad is truly random, then the resultant ciphertext will also be truly random and impossible to cryptanalyse. The same pad of random bits is available to the receiver of the message who simply reapplies the XOR function to the ciphertext to retrieve the original plaintext. This scheme is perfectly secure if the pad is both truly random and unique. This rules out practical use of this cipher due to difficulties in the distribution of the random pad or private key.

Stream Ciphers attempt to emulate the One Time Pad Cipher by using a psuedo-random bit generator to produce a random bit stream that can be regenerated by the receiver. The pad is now not truly random and therefore susceptible to cryptanalysis, but key distribution becomes simpler as the entire random string can be represented by a smaller key, which is used to key the random bit generator. As for a One Time Pad Cipher, the pseudo-random stream generated by the stream cipher is combined with the plaintext bitstream using XOR. The receiver at the other end has an identical random bit generator, the pseudo-random stream of which is XORed with the received ciphertext to retrieve the plaintext. Stream ciphers have long been used in military communications and their design is based heavily on mathematics. Unfortunately, while this means that the pseudo-random stream generally has good random properties, it also means that there is a wealth of mathematical knowledge that can be used to attack these ciphers.

Initial implementations of Stream Ciphers encrypted the plaintext one bit at a time. These implementations could be implemented extremely efficiently in hardware, especially where serial transfer of data over a communications link is common. However, software implementations are somewhat slower. Some stream ciphers can be modified to work one byte or word at a time where the size of the word is dependent on the bus size of the computer.

B.3.2.1 Principles of Stream Cipher Cryptology

A Stream Cipher is basically just a random number generator. Its security lies in two areas, the first of these being how good the random number generator is. What this means is that the closer the output stream resembles a truly random source, the better the cipher is. In order to understand why this is so, we need to look again at the concept of the One-Time Pad Cipher. If the random source is truly random, then the plaintext XORed with the random string will also be random and there are no patterns in the ciphertext to exploit. Since any stream of bits are just as possible as any other stream of the same length, it is possible to conclude that any stream of bits is a random stream, including the repetitive '101010101010...' cycle. While this stream is just as likely as any other from a true random stream generator, it does not qualify as being truly random as there is a simple representation of the same stream. This is also true of any output produced by a random stream generator as the random stream can be simply represented by an algorithm and a starting state value (key). There are many statistical tests that we can apply to streams of bits in order to determine how good the random number generator that produced these bits are.

There are many good random generators that are not cryptographically secure, leading to the second requirement for a good stream cipher. That is, given a random stream generator and a short string of random bits produced by that generator, how easy is it to determine the rest of the output from that random stream generator. This unpredictability is what protects us from a known plaintext attack. If a cryptanalyst knows a short segment of plaintext and the corresponding ciphertext, a simple XOR operation will reveal the random bits produced by the generator at that stage. Assuming that the cryptanalyst also knows the random generation algorithm utilised, we do not want them to be able to recreate the internal state of the generator and therefore produce the rest of the random bit stream.

Linear Congruential Generators are amongst the simplest of all random number generators where the next random number X_n is calculated from the previous random number X_{n-1} using the formula:

$$X_n = (aX_{n-1} + b) \bmod m \qquad \text{(B.21)}$$

In this case, the variables **a**, **b** and **m** are constants. Obviously, the random number produced will be in the range 0 to **m**-1 and if X_i is equal to X_j then X_{i+1} will be equal to X_{j+1}. There are rules for good choices of the constant **a**, b and **m** to ensure that the generator has a maximal period. That is the generator will cycle through all values between 0 and **m**-1 before repeating itself. If a random bit stream is required, or the random numbers are required in smaller groups of bits, then X_n can be broken into smaller groups and used in this fashion. Alternatively, a function may be applied to X_n in order to extract a random number of the required size. While Linear Congruential Generators make good random number generators, they are not cryptographically secure. Similarly neither are Quadratic Congruential Generators ($X_n = (aX_{n-1}{}^2 + bX_{n-1} + c) \bmod m$) nor Polynomial Congruential Generators in general.

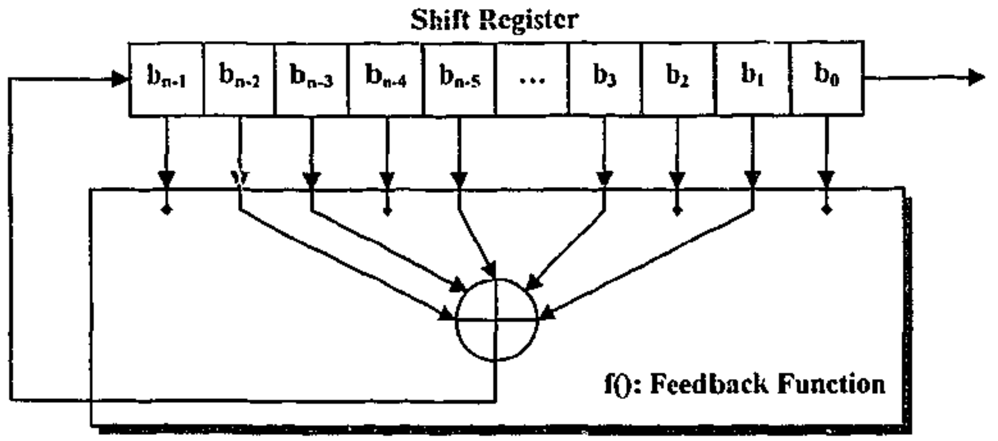


Figure B-10: Linear Feedback Shift Register Random Stream Generator

Linear Feedback Shift Registers (LFSR) are amongst the oldest types of technology used in Stream Ciphers, especially since they can be easily built in hardware and are generally very fast. A Linear Feedback Shift Register consists of two parts, a register of n bits and a feedback function f() which is the XOR of certain bits of the shift register. The output of an LFSR is the least significant, or the rightmost, bit of the shift register. When a random bit is required, the shift register is shifted by one bit to the right with the new leftmost bit being set to the output of the feedback function f(), an example of an LFRS can be seen in Figure B-10. As for Linear Congruential Generators, we must choose which bits will form the input to the feedback function, with particular choices ensuring we have maximal period of $2^n - 1$ output bits before the LFSR repeats itself. The security provided by a single LFSR is not high with a known plaintext of length 2n required to determine future random bits using the Berlekamp-Massey algorithm. While the mathematics and security of the LFSR are well understood, there are a number of more secure ciphers based on LFSR algorithms that use the LFSR in a non linear fashion or combine multiple LFSRs in a non linear fashion.

Additive Generators are more efficient than LFSR generators because they produce more than one random bit per cycle. Whilst they are not cryptographically secure in their own right, they can be combined to form more secure random number generators as in the FISH and PIKE stream ciphers. The state generator in an Additive Generator contains **n** **m**-bit words rather than **n** bits as for the LFSR. The function of an Additive Generator is exactly the same as that for the LFSR except that the feedback function is of the form:

$$f() = (X_a + X_b + \ldots + X_i) \bmod 2^{\alpha} \qquad \text{(B.22)}$$

Again, good choices for which word to use and what value of **n** to use ensures a maximal period random generator. If our additive generator uses **m** = 1 bit words and **α** = 1, then the generator degenerates to a simple LFSR.

S-Boxes in Stream Cipher design perform the same function as S-Boxes in Block Cipher design, they map a sequence of input bits to a sequence of output bits using a non-linear function. S-Boxes used in Stream Ciphers tend to be larger than those utilised in Block Cipher but in general they perform the same function. An interesting aside is the use of dynamic S-Boxes as in the RC4 encryption algorithm. A dynamic S-Box is one where the mapping function of the S-Box changes with each cycle of the cipher, thereby introducing further non linearity and extending the period of the random number generator. The RC4 stream cipher can be in one of about 2^{1700} internal states. While this doesn't ensure that the period of the cipher is 2^{1700} bytes, its non-linearity combined with this number of possible states ensures a cryptographically secure random stream generator.

Strongly related to LFSRs are Feedback with Carry Shift Registers, or FCSRs. These differ mainly in the feedback function f(), which employs the use of a carry register. Rather than performing an XOR on all the bits being tapped from the shift register, we add them together, along with the value in the carry register. The least significant bit of the result is then used to form the new bit for the shift register, the result is also right shifted through one bit to form the new value of the carry register. Using an FCSR requires more internal memory to store the carry register as well as slightly extra time to compute the feedback function, however the feedback function in an FCSR is less predictable than that of an LFSR. FCSRs are relatively new and there has been little cryptanalysis performed on these generators, however some mathematical work has been done and FCSRs can be partially analysed using a mathematical property called **2-adic** numbers. There have been some designs proposed that combine LFSR generators with FCSR generators, assuming that the add with carry confuses the algebraic properties of LFSR generators, and that XOR confuses the algebraic properties of FCSR generators.

Non-linear Feedback Shift Registers, or NLSRs, are also strongly related to LFSR and FCSR random stream generators, the main differences being that the feedback function f() is a more complicated non-linear function of the shift register. In this case, the feedback function would consist of a complicated series of XOR, AND, OR and ADD functions, all inter-operating on tapped bits from the shift register and intermediate results of these functions. The problem with NLSRs is that the feedback function is difficult to analyse and it is more difficult to design either a maximal period generator, or even a good random sequence generator. Some of the potential problems could be biases in output bits or runs of the same bit, sequence period length may depend on the key, and the generated sequence may begin randomly but eventually degenerate into a predictable output.

Many of these random stream generators can be combined in different ways to achieve a more cryptographically secure random stream generators, this is especially useful when considering Stream Ciphers built using Feedback Shift Registers. As mentioned previously, these generators can be built in hardware to both run quickly and using few components, combining a number of them together is still going to produce a fast, small random stream generator. These techniques are utilised less often with stream ciphers based around S-Boxes, mainly because these ciphers are generally implemented in software and combinations with other generators will merely slow down the generator.

What follows is a brief overview of the techniques that can be employed to combine random bit stream generators: Using Shift Registers to select outputs from other Shift Registers; using Shift Registers to clock other Shift Registers and using Shift Registers that clock themselves based on their output. All these techniques can be applied to LSFRs, FCSRs and NLSRs or a combination of these types of Shift Registers. All of the shift registers output a pseudo-random bit stream and can therefore be combined with each other. As for design of all types of encryption algorithms, the skill level involved in designing a secure cipher is high.

We can combine shift registers by using one register to select from the output of two or more separate shift registers. In a simple case we can combine three generators, labelled SR1, SR2 and SR3. If the output of SR1 is zero, then our final output is the output of SR2, otherwise we output SR3, this design can be seen in Figure B-11. This idea can be generalised to selecting from more than two shift registers, however, the security afforded is not that much greater than the individual generators. It is however possible to use this technique in a more complex combination of random bit generators.

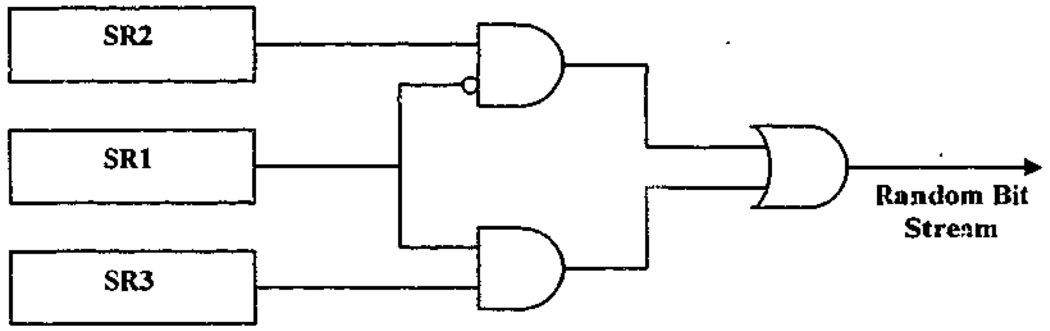


Figure B-11: Shift Register Selection Combination

We can combine Shift Registers by using the output of a shift register to clock another shift register. In this scenario, if the output of the first generator is a 1 bit, then we clock the second generator, otherwise we do not clock the second generator. We can combine a number of generators using this technique with the output bit being a function of the output of all the shift registers. Only some of the registers will actually shift on each cycle, making the output of the entire cipher more

difficult to predict. A variation of this technique uses shift registers that clock each other based on a clocking function.

Shift registers can also be programmed to clock themselves, to make their output more unpredictable. The simplest implementation of this technique uses the output of the shift register to determine how many times to clock the shift register, with the actual register being clocked multiple times for each cycle.

There are still other methods available to us when building actual Stream Ciphers from our building blocks. One of these involves using the output of multiple shift registers in a voting system to determine the final output bit. Another choice is to build a shrinking generator where two shift registers, SR1 and SR2, are used, for each cycle we continuously clock both registers until the output of SR1 is 1, the random bit is the output of SR2. This can also be reduced to using a single register and clocking it twice, using both the output bits to determine whether or not to repeat the cycle until an output bit is produced.

These techniques can be applied when combining random sequence generators to produce a more cryptographically secure random sequence generator. The speed at which random sequence generators can run, and their simplicity to build in hardware makes them practical for high bit rate applications. Also Stream Ciphers are more flexible than Block Ciphers in that the length of the ciphertext is identical to that of the plaintext.

B.3.2.2 Weaknesses and Practicalities

There are even more Stream Ciphers to choose from than Block Ciphers, and the choice is a difficult one to make. While we can easily say not to select a Stream Cipher based on a Linear Congruential Generator, there are still a number of choices available to us. Stream Ciphers based on Feedback Shift Registers are very well understood but their weaknesses are also well known. A common approach is to use a Shift Register based Stream Cipher where the key is reset regularly. This means that only a small portion of plaintext is encoded using a single key. Since there is little ciphertext associated with a single key, the cryptanalyst has less data to work with. Also, if a single key is broken, then only a small portion of plaintext becomes legible. We how have a more secure system overall whilst still having the convenience of a fast Stream Cipher.

As well as the Feedback Shift Register based Stream Ciphers, there are a number of other Random number generators that can be used as Stream Ciphers. These generators – examples include RC4 and SEAL – produce a series of random **n** bit numbers which are subsequently XORed with the plaintext to produce the ciphertext. While many Stream Ciphers are optimised for hardware implementations, these variants are specifically designed for optimal software implementations, at very high encryption rates when compared to Block Ciphers. Stream Ciphers offer good security, especially if they are only used to encrypt short spans of ciphertext and then re-keyed. However, one must be careful when selecting the encryption algorithm that it contains no known weaknesses.

B.3.2.3 Recommended Key Lengths for Stream Ciphers

Required key lengths for Stream Ciphers form a slightly more complex question than for Block Ciphers, this is because the concept of a key in a Stream Cipher is different than a key in a Block Cipher. In a Block Cipher, the key is integrated with the plaintext in order to produce a block of ciphertext. On the other hand, in a Stream Cipher, the key is used to set the initial internal state of the cipher. Since a Stream Cipher consists of a pseudo-random stream being XORed with the plaintext, the security lies in the generation of the random bit stream. This random bit stream is not dependant at all on the plaintext but instead on the current internal state of the random sequence generator. The length of the key is therefore the number of bits required to represent all possible internal states of the random sequence generator. A Stream Cipher consisting of two 32 bit Linear Feedback Shift Registers requires a 64 bit value to set the initial state of the two shift registers, whilst a Stream Cipher consisting of three 64 bit LFSRs requires a 192 bit key. On the other hand, a Stream Cipher like RC4 requires a minimum size 40 bit key to set its initial state.

Most Stream Ciphers based on Feedback Shift Registers are vulnerable to an attack better than a brute force attack, however this can be circumvented if we re-key the cipher regularly. In fact, the length of the key in this type of cipher is less important that the regularity of which the cipher is re-keyed. For a single LFSR, a simple attack requires only 2n bits of known plaintext, we can therefore state that the register should be re-keyed every **n** bits. This period can be made longer through the combination of more than one register, the optimal re-keying period should be uniquely calculated for the cipher that is being used.

The security offered by Stream Ciphers based on S-Boxes and other Block Cipher techniques depends on the actual algorithm chosen, but there are some for which a brute force attack is the currently best known attack. These ciphers offer a larger period utilising a somewhat smaller key. If the brute force attack is the best option available, then the recommended key lengths would be of the same order as those for Block Ciphers.

B.3.2.4 Amenability to Encryption of Streaming Multimedia

Stream Ciphers are perhaps the most easily amenable to the task of encrypting streaming multimedia. In fact, Stream Ciphers are suitable for many different types of application where speed of encryption is a prime consideration. While not generally considered to be as secure as Block Ciphers, Stream Ciphers have many other advantages apart from their speed. Stream Ciphers are the most flexible ciphers that can encrypt plaintext in block sizes as low as one bit which means that any block of plaintext can be encrypted without increasing its length.

A Stream Cipher is only as secure as its random bit generation algorithm. Shift register based Stream Ciphers are considered to be vulnerable unless the key is changed regularly. This tactic improves the strength of all Stream Ciphers as it gives the cryptanalyst less ciphertext per key to work with. The frequency of re-keying required and the amount of time required to re-key the Cipher can be of concern when regarding the overall speed of the Cipher, however, re-keying turns out to be a useful

feature when talking about streaming multimedia. Amongst the features of streaming media is the ability to seek to a different part of the stream, as well as fast playback. A seek feature would require the ability to start decoding the stream from set marker points within the stream, in the case of MPEG-1 at the beginning of each I-Frame. If we are employing a Stream Cipher, then we must be able to restart our random bit generator at each of these marker points. Restarting the Cipher with the same key reduces security as the same random bit stream is being used repetitively, however re-keying the Cipher at each marker point solves one of the security issues of Stream Ciphers as well as allowing a seek to any one of these marker points. This issue becomes more prevalent when considering the issue of fast playback as only select frames of the stream are transmitted, again re-keying the generator allows us to decrypt these segments of the total stream even when bits of the total ciphertext are missing.

A Block Cipher running in CBC mode can be considered to be a Stream Cipher, however this will generally run slower than most dedicated Stream Ciphers. Re-keying that would increase the strength of the Stream Cipher is required to provide interactive multimedia, and the flexibility provided by the Stream Cipher in maintaining ciphertext length makes these types of Ciphers ideal candidates for the task of encrypting streaming multimedia. Finally, the speed afforded by a good Stream Cipher means that the CPU time penalty due to encryption is minimised. While this is a minor problem on the client side where only one stream is being decrypted and viewed, it could make a large difference on the server side if we choose to employ a scheme whereby the stream is encrypted in real time.

B.4 Conclusion

This thesis recommends the use of a Private Key Stream Cipher when designing an encryption scheme for the purpose of streaming multimedia. Combined with a good partial encryption algorithm which will select some bits of the stream for encryption, a regularly re-keyed Stream Cipher will provide the required security for the streaming asset as well as accessibility to digital media features such as seeking and different speed playbacks. The choice of which Stream Cipher to employ, or indeed to use a Block Cipher in CBC mode, is an academic decision which will need to be based on a number of ideas. These include the speed of encryption, the time taken to re-key the Cipher, the required re-keying frequency of the Cipher, the required re-keying frequency of the multimedia stream and of course the strength of the Cipher in its own right.

Public Key Ciphers and Private Key Block Ciphers are not suitable for streaming multimedia. Block Ciphers have restrictions in that the ciphertext length has to be a multiple of the block size and are not as fast as Stream Ciphers. While Public Key Ciphers are too slow to apply to this streaming multimedia, they should be considered when solving the Key Management issue. In order to allow an authorised person to view an encrypted stream, they require the decryption key. Key Management algorithms using Public Key Ciphers will allow us to securely transmit the Private Key to the authorised viewer to allow them to decode the encrypted multimedia stream.

Appendix C
Source Code

In the accompanying CD to this thesis, you will find both the source code listings that implement the MPEG-1 Ciphers as well as compiled applications and modules that will allow you to use this software immediately. In this Appendix I will discuss and present relevant portions of the source that are responsible for implementing the Cipher design presented earlier in this thesis. Source code not shown or discussed in this Appendix is not concerned with the task of MPEG stream encryption or decryption, but rather with enabling the specified module or application to correctly function within the Microsoft Windows environment. Each section of this Appendix will discuss a different aspect of the source code listings. All source can be found in the *"\Development"* subdirectory on the provided CD, all applications can be built through the Microsoft Visual C++ Workspace file *"PhD Thesis Development.dsw"*.

C.1 ClassFactory and StreamCipherBase Classes

Two classes are defined in the two header files *"ClassFactory.h"* and *"StreamCipherBase.h"*. These two classes enable generic Cipher Parser modules to be built that can use a range of different base ciphers. This allows the same code-base to be used for both the prototype cipher and the more secure SEAL based cipher. A further advantage to this approach is that further experimentation with new cipher bases can be easily performed through inheritance from the base class **StreamCipherBase**. The **ClassFactory** class is a generic template which can be used in a variety of applications. It enables specific class instantiation at real-time, selecting one type from a range of registered classes. While this task can be performed in many ways, the class factory approach enables the addition of new class types to the factory, without any modification of any existing code. A short description of how to use the **ClassFactory** template is included in the source code listing.

Of more interest is the **StreamCipherBase** class, this class provides a base model for a Stream Cipher that can be used for encryption of an MPEG-1 bitstream. This class is pure, and as such cannot be directly instantiated – however it does provide a range of virtual methods that are intended to be overloaded. Note that the Stream Cipher can be used to generate two separate pseudorandom streams, one each for encrypting the Video and Audio Stream. The base class supports five virtual methods, the first two are used to respectively resynchronise the Video and Audio Cipher modules within the class instance given a unique 32-bit value, this value is not a secret key but instead a publicly known sequence number that can be used re-synchronise the pseudorandom stream. The next two methods respectively return pseudorandom bytes to encrypt the Video and Audio Streams, each call to these methods returns a single byte. The final virtual method is used to inform the cipher of the secret key to be used for encryption. While these two classes do not perform any of the encryption process in themselves, they are used to simplify the actual implementations discussed in the next section.

C.2 Stream Ciphers

In this section I will discuss the code that implements the **XORStreamCipher** and **SEALStreamCipher** classes. These classes are inherited from **StreamCipherBase** and are used to generate the pseudorandom streams as used in the prototype cipher design (Chapter 4) and the secure cipher design (Chapter 5) respectively.

C.2.1 XORStreamCipher Class

The **XORStreamCipher** class is implemented in the *"XORStreamCipher.cpp"* file, this class inherits from the **StreamCipherBase** base class and is used to implement the prototype cipher as designed in Chapter 4. The prototype cipher design is very simple, using a single 8-bit key which is used as the (not-so) pseudorandom value to be returned. This allows for a very simple implementation, the 8-bit key is stored in an internal member variable and returned with each call to either *GetNextRandomAudioByte()* or *GetNextRandomVideoByte()*, the resynchronisation methods are not implemented as there is no resynchronisation to occur with the prototype cipher. Also within this implementation file is the registration of the **XORStreamCipher** base with the Class Factory described in the previous section.

C.2.2 SEALStreamCipher Class

The **SEALStreamCipher** class is implemented in the *"SEALMacros.h"* and *"SEALStreamCipher.cpp"* files, like **XORStreamCipher**, this class also inherits from the **StreamCipherBase** base class and is used to implement the secure cipher as designed in Chapter 5. The implementation is broken into two parts, that of a basic SEAL Cipher (in *"SEALMacros.h"*), and its modification for the generation of synchronised pseudorandom byte streams (in *"SEALStreamCipher.cpp"*). The basic SEAL Cipher is primarily implemented through the macros defined in the *"SEALMacros.h"* header file. The core SEAL code is implemented as macros to improve the execution speed of the potentially time critical code. The macros as implemented perform parts of the SEAL task as described in [x] and [y]. The macros themselves do not produce a single pseudo-random byte output, but instead are designed to be used as per the original algorithm to produce an L kB pseudo-random string with the appropriate loop code. I will describe how these macros are used to produce a single pseudo-random byte when discussing the actual **SEALStreamCipher** class.

The first set of macros are used to define arrays to store the SEAL **T**, S and **R** lookup Tables. The first three macros define arrays of 32 bit words while the next three define the required offsets into these arrays to correctly define the Tables. As shown in the definition of the SEAL cipher in Figure 5-4 and Figure 5-5, the **T** and S Table are of fixed size while the size of the **R** Table is

dependant on the maximum length of pseudo-random string produced – also the number of cycles through the SEAL generation function outer loop.

The next set of macros defines the basic functions of rotation of 32 bit values to the right through a specified number of bits, followed by a set of which macros define the four functions and constants used in the SEAL $\mathbf{G}_{\mathbf{key}}$ function. Another macro, **SEAL_GKEY** is a straight implementation of $\mathbf{G}_{\mathbf{key}}$, taking as input a 160-bit secret key and a 32-bit selector/hash value. The 160-bit output hash value is stored in the desired location. This macro is called to fill the **T**, S and **R** Tables. Following the $\mathbf{G}_{\mathbf{key}}$ macro, there is a set of macros, one to fill an array with a continuous set of $\mathbf{G}_{\mathbf{key}}$ results, and three macros that use this preceding macro to fill the **T**, S and **R** Tables. These final macros will be used by the calling code to fill the arrays defined by the first set of macros in this header file.

The final set of macros perform the actual SEAL code, as shown in Figure 5-6 there are two loops required to produce a pseudo-random output string from SEAL, an inner loop which generates four 32-bit values for each of its 64 iterations, and an outer loop which sets some variables for the next cycle of the inner loop. Each of these two code segments have been implemented as separate macros, simplifying any SEAL code to containing two loops with one macro being called inside each loop. These macros take as input pointers to the Lookup Tables to be used in random string generation as well as state variables used to store calculations during each cycle. The inner loop macro also takes a pointer to an array where it can store its 16-bytes of pseudo-random output.

The SEAL macros can be combined to implement the SEAL cipher using the following steps, some sample code is also provided in Figure C-1:

- Define variables to store the state information of the SEAL Cipher.
- Define the Lookup Tables using the provided macros.
- Define pointers to the correct offsets into the Lookup Tables using the provided macros.
- Fill the Lookup Tables given the 160-bit secret key.
- Implement the basic inner and outer loop code.

It is however, important to consider how these macros can be used to generate a single byte of the pseudorandom streams produced by the SEAL Cipher – these modifications, along with resynchronisation, are implemented by the code in the file *"SEALStreamCipher.cpp"*. The purpose of the **SEALStreamCipher** class is to allow for the generation of a pseudo-random stream of bytes, therefore it will be necessary to turn the SEAL generation function inside out to produce a series of bytes rather than a complete string as per the original algorithm. It is also necessary to allow for resynchronisation, for the SEAL Cipher we will use the 32-bit resynchronisation value to select between one of the 2^{32} possible pseudorandom strings that SEAL can generate, remembering to reset to the beginning of the newly selected pseudorandom stream. Finally, we should remember that it is essential for **SEALStreamCipher** to generate two different pseudo-random streams, one for each of the Video and Audio Streams. Since the secret key for both streams is identical, and the SEAL Cipher does not modify the contents of the **R**, **S** and **T** Tables during execution, it is prudent to use the same set of tables for both ciphers – as such the class maintains a single instance of the **R**, **S** and **T** tables.

```
#define   OUTERLOOPS                         64
{
     uint32     A, B, C, D, N1, N2, N3, N4, index = 0;
     uint32     pui32Random[256 * OUTERLOOPS];

     SEAL_DEFINETTABLE(pui32TTable);
     SEAL_DEFINESTABLE(pui32STable);
     SEAL_DEFINERTABLE(pui32RTable, OUTERLOOPS);

     uint32      *pui32T = SEAL_TTABLEOFFSET(pui32TTable);
     uint32      *pui32S = SEAL_STABLEOFFSET(pui32STable);
     uint32      *pui32R = SEAL_RTABLEOFFSET(pui32RTable);

     SEAL_FILLTTABLE(pui32Key, pui32TTable);
     SEAL_FILLSTABLE(pui32Key, pui32STable);
     SEAL_FILLRTABLE(pui32Key, OUTERLOOPS, pui32RTable);

     for (int loop = 0; loop < OUTERLOOPS, loop++)
     {
          SEAL_OUTERLOOP(loop, n, pui32R, pui32T, A, B, C, D,
                         N1, N2, N3, N4);
          for (int count = 0; count < 64; count++)
          {
               SEAL_INNERLOOP(count, pui32S, pui32T, A, B, C, D,
                              N1, N2, N3, N4, pui32Output + index);
               index+= 4;
          }
     }
}
```

Figure C-1: SEAL Sample Implementation

The tables are created as member variables of the class and the sizes are defined at compile time, however the offsets into these tables are calculated and stored when the class is instantiated. When the ***SetKey()*** method is called, we need to populate the **R**, **S** and **T** tables based on the secret key, this is performed by calling the three **SEAL_FILLxTABLE** macros. At this stage we also resynchronise the ciphers, choosing by default to generate the pseudo-random string identified by the ***n*** resynchronisation value of **0**.

In order to reverse the implementation of the SEAL loops, we must permanently maintain the inner and outer loop counters within class variables, these are both reset to zero when the cipher is resynchronised. We turn the two SEAL loops inside-out through the use of *if-then* statements and look at the implementation of the basic algorithm in the inner loop. The inner loop produces four 32-bit pseudorandom words (or 16 bytes). The class maintains an array of 16 bytes that is filled with each call to the **SEAL_INNERLOOP** macro, as well as an index as to how many of these bytes have been returned in previous requests for a byte. If this index is zero, then the random byte array is empty and must be filled via a call to **SEAL_INNERLOOP** before a value can be returned. Once the array contains data – or the index value is non-zero, indicating the array contains partially valid data – we retrieve the next random value from the array and increment the index before returning. When the index becomes 16, we have run out of data in the array, the index is reset to zero so that when the next random byte is requested, **SEAL_INNERLOOP** is called again to fill the random array.

If the 16 byte random array must be filled, we first need to check if the outer loop code must be executed before the inner loop macro is called to fill the array. This is the case when the inner loop counter is 0 – at this stage we must execute the outer loop code and increment the outer loop counter prior to running the inner loop code. Once this is complete, we execute the inner loop code before incrementing the inner loop counter which resets to zero if it reaches 64. This ensures that the outer loop code is only executed once the inner loop code is executed 64 times.

This implementation is more efficient than generating the entire pseudorandom string and extracting bytes as required, a single video or audio frame may require only a small number of pseudorandom bytes and CPU time can be otherwise wasted generating random bytes that will never be used. This implementation generates the pseudo-random string in blocks of 16 bytes, minimising the number of non-utilised bytes, the average number of generated random bytes which will not be used for each resynchronisation value is 8.

C.3 Parsing and Encrypting the Video and Audio Streams

In this section I will discuss the code that implements the **MPEGVideoParser** and **MPEGAudioParser** classes. These classes implement the two stream parsers whose final design is outlined in Chapter 5 and utilise instances of either **XORStreamCipher** or **SEALStreamCipher** to generate bytes to XOR with the plaintext. These classes also implement the minor changes to the generic Stream Cipher to ensure against the generation of unviable Video and Audio Streams (e.g. False MPEG-1 Headers).

C.3.1 MPEGVideoParser Class

The **MPEGVideoParser** class is implemented in the ***"MPEGVideoParser.h"*** and ***"MPEGVideoParser.cpp"*** files, and is used to parse an MPEG-1 Video Stream, selecting the MacroBlock data to pass to the provided Stream Cipher instance to either encrypt or decrypt the Video Stream. This class implements the State Machine described in Figure 5-9 and consists of two major steps:

- The initialisation phase generates a lookup table that can determine the next state based on both the current state and the next byte in the stream to be parsed.
- The parsing phase will parse the stream in re-startable blocks, the current state being stored within the class instance member variables. The actions defined in the State Machine are implemented based on the current state as well as passing the necessary bytes through the cipher. All of the actions defined in the State Machine are implemented here, including the calculation of the Stream Cipher resyncrhonisation values and the subsequent resynchronisation of the cipher. It is here that the final modification to the basic SEAL and XOR Stream Ciphers

are performed – the cipher module returns a byte for XOR, its value is checked before determining whether to actually perform the XOR.

Note that the class implementation requires specification of a cipher module to use during the parsing process. This allows the same code to be used with both the SEAL based cipher and the basic prototype XOR based cipher. When parsing an MPEG-1 Video Stream, the output using the XOR module will be identical to if the original State Machine (Figure 4-5) was used – the modified state machine will perform more work by calculating the values *lGOPTimeStamp* and *lPictureCount*, but these will not change the eventual stream generated when using the XOR module.

Finally, this class can be used in different ways, when implementing DirectShow filters, the Cipher class is created by the filter along with the **MPEGVideoParser** class, the cipher is registered with the parser and the Video Stream is parsed directly after it has been extracted from the System Stream. On the other hand, when developing entire bitstream encryption applications, we are processing complete System Streams and therefore use the **MPEGSystemParser** class presented in the next section to modify the bitstream, this class is then responsible for generating the instances of the cipher class and **MPEGVideoParser** classes.

C.3.2 MPEGAudioParserClass

The **MPEGAudioParser** class is implemented in the *"MPEGAudioParser.h"* and *"MPEGAudioParser.cpp"* files, and is used to parse an MPEG-1 Audio Stream, selecting the compressed audio data to pass to the provided Stream Cipher instance to either encrypt or decrypt the Audio Stream. This class implements the State Machine described in Figure 5-11 and, like the **MPEGVideoParser** class, consists of the same two major implementation steps.

Also like the **MPEGVideoParser** class, the **MPEGAudioParser** class implementation requires specification of a cipher module to use during the parsing process – again allowing the same code to be used with both the SEAL based cipher and the basic prototype XOR based cipher. What is different however, is that the modified State Machine for the Audio Cipher (Figure 5-11) used in **MPEGAudioParser** will generate a slightly different stream as compared to the original State Machine (Figure 4-9) when using the XOR Cipher module. This is due to the extra two bytes of the data stream being left as plaintext for cipher resynchronisation purposes. While this means that software changes would be necessary in order to exactly reproduce the cipher designed in Chapter 4, it was deemed more useful to have a single code base that implemented both the prototype and secure ciphers. As such, the prototype XOR based cipher implemented in the source code uses the proposed simple cipher from Chapter 4, while using the more complex State Machine presented in Chapter 5.

Finally, the MPEGAudioParser class is designed to be used as a standalone module for encrypting and decrypting an MPEG-1 Audio Stream. Like the **MPEGVideoParser** implementation, it should be used directly within a DirectShow type application where the Video and Audio Streams

can be accessed after they have been de-multiplexed from the System Stream, but through the **MPEGSystemParser** class in an application that must process the System Stream directly.

C.4 Parsing and Encrypting the System Stream

In this section I will discuss the code that implements the **MPEGSystemParser** class. This class implements a simple parser for the MPEG-1 System Stream and passes the multiplexed streams contained within to instances of the **MPEGVideoParser** and **MPEGAudioParser** classes for actual encryption of the overall bitstream. The **MPEGSystemParser** class is not needed in applications where access to the contained Video and Audio Streams is available – such as in a DirectShow application – but is where only access to the System Stream is possible.

The implementation of the **MPEGSystemParser** class allows for processing of an intact MPEG-1 System Stream and the in-place encryption or decryption of the multiplexed Video and Audio Streams. The implementation is very similar to the **MPEGVideoParser** and **MPEGAudioParser** classes, whereby an initialisation phase builds a lookup table to implement the State Machine defined in Figure 4-1, and an execution phase which processes the System Stream in blocks, passing sub-blocks of Video or Audio Stream data to their respective parsers for processing.

The implementation is straightforward, the class creates a single instance of a cipher module, which is shared between the Video and Audio Stream parsers as well as single instances of a Video Parser and an Audio Parser. Ideally, multiple Video and Audio Stream parsers should be created for the potential case of multiple Video/Audio Streams multiplexed within the System Stream, but for testing purposes we assume that there is only one stream of each type. This means that the **MPEGSystemParser** class as shown will not correctly encrypt or decrypt a bitstream containing multiple Video or Audio Streams and that the code must be modified to support this extra functionality.

C.5 Listings

The following twenty pages contain printouts of the full listings of the files described above. These files form the part of the total supplied source code that implements the primary functionality of the encryption scheme developed in this thesis. Other source code is available on the included CD, but this code pertains to using the aforementioned classes to implement a working module or application and has more to do with user interface and system development than encryption of an MPEG-1 bitstream. Of course, the code included in the following listings is also available on the CD.

May 02, 03 9:12 **ClassFactory.h** Page 1/3

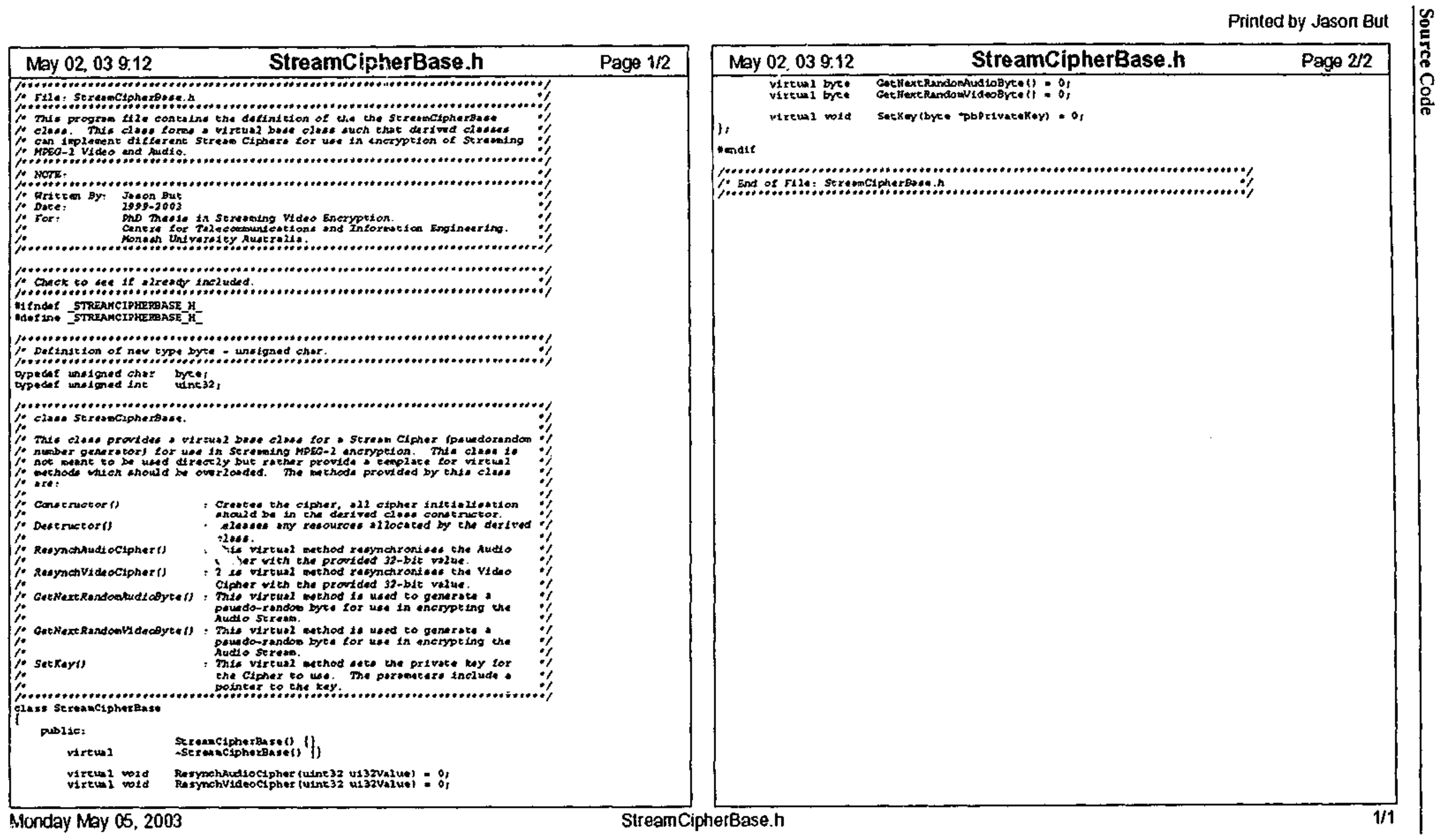
```
/****************************************************************************/
/* File: ClassFactory.h                                                     */
/****************************************************************************/
/* This header file creates some C++ templates to automate the creation of a */
/* range of derived classes.  The Class Factory will take a string identifier */
/* and create the correct derived class based on the value of the identifier. */
/****************************************************************************/
/* NOTE:                                                                    */
/****************************************************************************/
/* Written By:  Jason But                                                   */
/* Date:        1999-2003                                                   */
/* For:         PhD Thesis in Streaming Video Encryption.                   */
/*              Centre for Telecommunications and Information Engineering.  */
/*              Monash University Australia.                                */
/****************************************************************************/

/****************************************************************************/
/* Check to see if already included.                                        */
/****************************************************************************/
#ifndef _CLASSFACTORY_H_
#define _CLASSFACTORY_H_
#include <windows.h>

/****************************************************************************/
/* Include Standard C++ Library and Template Header Files.                  */
/****************************************************************************/
#include <map>
#include <string>

/****************************************************************************/
/* Class Template ClassFactory.                                             */
/*                                                                          */
/* This template actually implements the Class Factory.                     */
/* Within the class, two new types are defined:                             */
/*  fnCreate -  A type which defines a function that takes no parameters and */
/*              returns apointer to CreatedType.  This can be modified to   */
/*              accept parameters to use when constructing the class.       */
/*  dbClasses - A type which defines a map between the provided class       */
/*              'dentifiers and the corresponding class creation function.  */
/*                                                                          */
/* Within the class, there is one internal variable (FactoryDataBase) of type */
/* dbClasses which stores the lookup class identifiers and creation functions.*/
/*                                                                          */
/* The class constructor is private and must be accessed through static    */
/* methods.  No copy constructor (not even default) is provided so an       */
/* instance of ClassFactory cannot be copied.                               */
/*                                                                          */
/* The Public Member Methods are:                                           */
/* Instance()           : Returns the instance of the ClassFactory class.   */
/* RegisterCreateFunc() : Registers a class creation function with the factory*/
/* CreateClass()        : Creates an instance of a class, the type of which */
/*                        is determined by the provided identifier.         */
/*                                                                          */
/* Usage:                                                                   */
/*   Once all derived classes have been registered, instances of BaseC and  */
/*   DerivedC can be created using the commands:                            */
/*                                                                          */
/*   newBase = ClassFactory<BaseC>::Instance().create(0);                   */
/*   newDerived = ClassFactory<BaseC>::Instance().create(1);                */
/****************************************************************************/
template <class CreatedType>
class ClassFactory
{
    private:
        typedef             CreatedType * (*fnCreate)();
```

May 02, 03 9:12 **ClassFactory.h** Page 2/3

```
        typedef             std::map<long, fnCreate> dbClasses;

        dbClasses           FactoryDataBase;

                            ClassFactory() {}
                            ClassFactory(const ClassFactory&);

        ClassFactory        &operator=(const ClassFactory&);

    public:
        enum                Error { errClassNotRegistered };

        static ClassFactory *Instance();

        void                RegisterCreateFunc(const long &lIndex, fnCreate func);

        CreatedType *       CreateClass(const long &lIndex) const;
};

/****************************************************************************/
/* Class Template ClassFactory Implementation.                              */
/****************************************************************************/
/* ClassFactory<CreatedType> * Instance()                                   */
/*                                                                          */
/* This static member method creates a static member variable of the specific */
/* Factory type, it then returns a pointer to the created instance.         */
/****************************************************************************/
template <class CreatedType>
ClassFactory<CreatedType> *
ClassFactory<CreatedType>::Instance()
{
    static ClassFactory<CreatedType> cInstance;
    return &cInstance;
}

/****************************************************************************/
/* void RegisterCreateFunc(const long &lIndex, fnCreate func)               */
/*                                                                          */
/* This public member method registers a class creation function and the    */
/* corresponding identifier in the FactoryDataBase for later lookup.        */
/****************************************************************************/
template <class CreatedType>
void
ClassFactory<CreatedType>::RegisterCreateFunc(const long &lIndex, fnCreate func)
{
    FactoryDataBase[lIndex]=func;
}

/****************************************************************************/
/* CreatedType *CreateClass(const long &lIndex) const                       */
/*                                                                          */
/* This public member method will create an instance of the class identified */
/* by ClassID.  It looks for an entry for ClassID in FactoryDataBase, if it  */
/* exists the corresponding Create function is called and its value returned. */
/* Otherwise an exception is thrown.                                        */
/****************************************************************************/
template <class CreatedType>
CreatedType *
ClassFactory<CreatedType>::CreateClass(const long &lIndex) const
{
    dbClasses::const_iterator iIndex = FactoryDataBase.find(lIndex);

    if (iIndex == FactoryDataBase.end()) throw errClassNotRegistered;

    return (*iIndex).second();
```

May 02, 03 9:12 **ClassFactory.h** Page 3/3

```
}

/****************************************************************************/
/* Class Template RegisterInFactory.                                        */
/*                                                                          */
/* This templ'te class is used to allow for registration of the classes that */
/* the Class Factory can create.  An instance of this class should be created */
/* ONCE only for each class type that the Class Factory can create.  The    */
/* template parameters include the Base Class type, the derived Class type  */
/* and the type used as an identifier to determine whether the class should */
/* be created or not.                                                       */
/*                                                                          */
/* The Public Member Methods are:                                           */
/* Constructor      : Registers the class CreateInstance() method with the  */
/*                    corresponding ClassFactory.  A single instance should */
/*                    be created for each class type that must be registered. */
/* CreateInstance() : Creates an instance of the specified class.  Should not */
/*                    be called directly, instead being called within the   */
/*                    corresponding ClassFactory to actually create the class */
/*                    instance.                                             */
/*                                                                          */
/* Usage:                                                                   */
/*   Assuming a Base Class called BaseC and a Derived Class called DerivedC, */
/*   that are to be created with the string identifiers "Base" and "Derived", */
/*   they should be registered with the commands:                           */
/*                                                                          */
/*  RegisterInFactory<BaseC, BaseC> regBase("Base");                        */
/*  RegisterInFactory<BaseC, DerivedC> regDerived("Derived");               */
/****************************************************************************/
template <class BaseType, class CreatedType>
class RegisterInFactory
{
    public:
        RegisterInFactory(const long &lIndex)
        {
            ClassFactory<BaseType>::Instance()->RegisterCreateFunc(lIndex, CreateInstance);
        }

        static BaseType *CreateInstance()
        {
            return (BaseType *) (new CreatedType);
        }
};

#endif

/****************************************************************************/
/* End of File: classfactory.h                                              */
/****************************************************************************/
```



Printed by Jason But

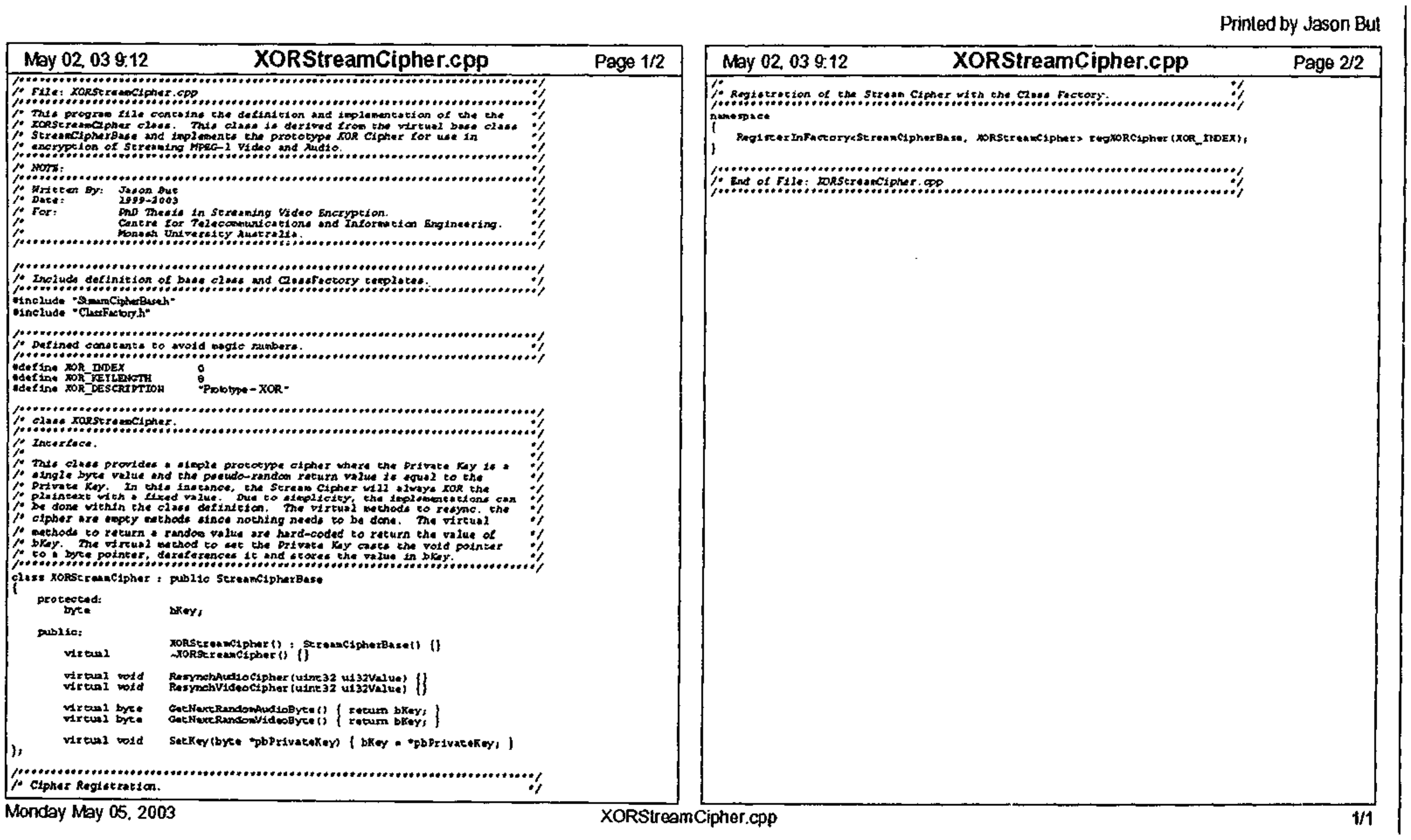
May 02, 03 9:12 **StreamCipherBase.h** Page 1/2

```
/******************************************************************************/
/* File: StreamCipherBase.h                                                   */
/******************************************************************************/
/* This program file contains the definition of the the StreamCipherBase     */
/* class.  This class forms a virtual base class such that derived classes    */
/* can implement different Stream Ciphers for use in encryption of Streaming */
/* MPEG-1 Video and Audio.                                                    */
/******************************************************************************/
/* NOTE:                                                                      */
/******************************************************************************/
/* Written By:  Jason But                                                     */
/* Date:        1999-2003                                                     */
/* For:         PhD Thesis in Streaming Video Encryption.                     */
/*              Centre for Telecommunications and Information Engineering.    */
/*              Monash University Australia.                                  */
/******************************************************************************/

/******************************************************************************/
/* Check to see if already included.                                          */
/******************************************************************************/
#ifndef _STREAMCIPHERBASE_H_
#define _STREAMCIPHERBASE_H_

/******************************************************************************/
/* Definition of new type byte - unsigned char.                               */
/******************************************************************************/
typedef unsigned char   byte;
typedef unsigned int    uint32;

/******************************************************************************/
/* class StreamCipherBase.                                                    */
/*                                                                            */
/* This class provides a virtual base class for a Stream Cipher (pseudorandom */
/* number generator) for use in Streaming MPEG-1 encryption.  This class is   */
/* not meant to be used directly but rather provide a template for virtual    */
/* methods which should be overloaded.  The methods provided by this class    */
/* are:                                                                       */
/*                                                                            */
/* Constructor()              : Creates the cipher, all cipher initialisation */
/*                              should be in the derived class constructor.   */
/* Destructor()               :  leases any resources allocated by the derived */
/*                              class.                                        */
/* ResynchAudioCipher()       :  is virtual method resynchronises the Audio   */
/*                               er with the provided 32-bit value.           */
/* ResynchVideoCipher()       : T is virtual method resynchronises the Video  */
/*                              Cipher with the provided 32-bit value.        */
/* GetNextRandomAudioByte()   : This virtual method is used to generate a     */
/*                              pseudo-random byte for use in encrypting the  */
/*                              Audio Stream.                                 */
/* GetNextRandomVideoByte()   : This virtual method is used to generate a     */
/*                              pseudo-random byte for use in encrypting the  */
/*                              Audio Stream.                                 */
/* SetKey()                   : This virtual method sets the private key for  */
/*                              the Cipher to use.  The parameters include a  */
/*                              pointer to the key.                           */
/******************************************************************************/
class StreamCipherBase
{
    public:
                        StreamCipherBase() {}
        virtual         ~StreamCipherBase() {}

        virtual void    ResynchAudioCipher(uint32 ui32Value) = 0;
        virtual void    ResynchVideoCipher(uint32 ui32Value) = 0;
```

May 02, 03 9:12 **StreamCipherBase.h** Page 2/2

```
        virtual byte    GetNextRandomAudioByte() = 0;
        virtual byte    GetNextRandomVideoByte() = 0;

        virtual void    SetKey(byte *pbPrivateKey) = 0;
};

#endif

/******************************************************************************/
/* End of File: StreamCipherBase.h                                            */
/******************************************************************************/
```

Monday May 05, 2003 StreamCipherBase.h 1/1



Printed by Jason But

May 02, 03 9:12 **XORStreamCipher.cpp** Page 1/2

```
/******************************************************************************/
/* File: XORStreamCipher.cpp                                                  */
/******************************************************************************/
/* This program file contains the definition and implementation of the the    */
/* XORStreamCipher class.  This class is derived from the virtual base class  */
/* StreamCipherBase and implements the prototype XOR Cipher for use in        */
/* encryption of Streaming MPEG-1 Video and Audio.                            */
/******************************************************************************/
/* NOTE:                                                                      */
/******************************************************************************/
/* Written By:  Jason But                                                     */
/* Date:        1999-2003                                                     */
/* For:         PhD Thesis in Streaming Video Encryption.                     */
/*              Centre for Telecommunications and Information Engineering.    */
/*              Monash University Australia.                                  */
/******************************************************************************/

/******************************************************************************/
/* Include definition of base class and ClassFactory templates.               */
/******************************************************************************/
#include "StreamCipherBase.h"
#include "ClassFactory.h"

/******************************************************************************/
/* Defined constants to avoid magic numbers.                                  */
/******************************************************************************/
#define XOR_INDEX           0
#define XOR_KEYLENGTH       8
#define XOR_DESCRIPTION     "Prototype - XOR"

/******************************************************************************/
/* class XORStreamCipher.                                                     */
/******************************************************************************/
/* Interface.                                                                 */
/*                                                                            */
/* This class provides a simple prototype cipher where the Private Key is a   */
/* single byte value and the pseudo-random return value is equal to the       */
/* Private Key.  In this instance, the Stream Cipher will always XOR the      */
/* plaintext with a fixed value.  Due to simplicity, the implementations can  */
/* be done within the class definition.  The virtual methods to resync. the   */
/* cipher are empty methods since nothing needs to be done.  The virtual      */
/* methods to return a random value are hard-coded to return the value of     */
/* bKey.  The virtual method to set the Private Key casts the void pointer    */
/* to a byte pointer, dereferences it and stores the value in bKey.           */
/******************************************************************************/
class XORStreamCipher : public StreamCipherBase
{
    protected:
        byte            bKey;

    public:
                        XORStreamCipher() : StreamCipherBase() {}
        virtual         ~XORStreamCipher() {}

        virtual void    ResynchAudioCipher(uint32 ui32Value) {}
        virtual void    ResynchVideoCipher(uint32 ui32Value) {}

        virtual byte    GetNextRandomAudioByte() { return bKey; }
        virtual byte    GetNextRandomVideoByte() { return bKey; }

        virtual void    SetKey(byte *pbPrivateKey) { bKey = *pbPrivateKey; }
};

/******************************************************************************/
/* Cipher Registration.                                                       */
```

May 02, 03 9:12 **XORStreamCipher.cpp** Page 2/2

```
/*                                                                            */
/* Registration of the Stream Cipher with the Class Factory.                  */
/******************************************************************************/
namespace
{
    RegisterInFactory<StreamCipherBase, XORStreamCipher> regXORCipher(XOR_INDEX);
}

/******************************************************************************/
/* End of File: XORStreamCipher.cpp                                           */
/******************************************************************************/
```

Monday May 05, 2003 XORStreamCipher.cpp 1/1

Printed by Jason But

May 02, 03 9:12 **SEALMacros.h** Page 1/5

```
/*****************************************************************************/
/* File: SEALMacros.h                                                        */
/*****************************************************************************/
/* This program file contains the definition and implementation of a range of */
/* macros that implement the SEAL 3.0 Stream Cipher.  Code is implemented as  */
/* macros rather than functions or class methods to ensure that the final code*/
/* contains no function calls (increasing optimisation).                     */
/*****************************************************************************/
/* NOTE:                                                                     */
/*****************************************************************************/
/* Written By:  Jason But                                                    */
/* Date:        1999-2003                                                    */
/* For:         PhD Thesis in Streaming Video Encryption.                    */
/*              Centre for Telecommunications and Information Engineering.   */
/*              Monash University Australia.                                 */
/*****************************************************************************/

/*****************************************************************************/
/* Check to see if already included.                                         */
/*****************************************************************************/
#ifndef _SEALMACROS_H_
#define _SEALMACROS_H_

/*****************************************************************************/
/* Definition of new type uint32 - unsigned int (32-bits).                   */
/*****************************************************************************/
typedef unsigned int    uint32;

/*****************************************************************************/
/* Macros to help in the generation of SEAL pseudo-random bytes.             */
/*****************************************************************************/
/* SEAL_DEFINETTABLE(), SEAL_DEFINESTABLE(), SEAL_DEFINERTABLE(),            */
/* SEAL_TTABLEOFFSET(), SEAL_STABLEOFFSET(), SEAL_RTABLEOFFSET()             */
/*                                                                           */
/* These macros allow us to define the arrays required to hold the T, S and R */
/* tables simply and without error.  Note that the size of the R table is    */
/* dependent on the maximum number of outer loops within the SEAL Cipher.    */
/* The other macros return an offset into the tables that point to the correct*/
/* locations of the beginning of the three arrays.  This is necessary as the */
/* S Table is offset 4 bytes (1 32-bit word) in and the R Table is offset 8  */
/* bytes (2 32-bit words) into the arrays as filled via calls to the macros  */
/* to fill these tables (SEAL_FILLxTABLE()).                                 */
/*****************************************************************************/
#define SEAL_DEFINETTABLE(pui32ArrayName)    uint32  pui32ArrayName[515];

#define SEAL_DEFINESTABLE(pui32ArrayName)    uint32  pui32ArrayName[260];

#define SEAL_DEFINERTABLE(pui32ArrayName, iMaxOuterLoops)               \
    uint32  pui32ArrayName[5 * ((4 * iMaxOuterLoops + 6) / 5)];

#define SEAL_TTABLEOFFSET(pui32TArray)       pui32TArray

#define SEAL_STABLEOFFSET(pui32SArray)       pui32SArray + 1

#define SEAL_RTABLEOFFSET(pui32RArray)       pui32RArray + 2

/*****************************************************************************/
/* ROTATE2(), ROTATE9(), ROTATE8(), ROTATE16()                               */
/* ROTATE24(), ROTATE27(), ROTATE31()                                        */
/*                                                                           */
/* These macros rotate a 32 bit value to the right through the number of bits */
/* as shown in the macro title.  Each required rotation is done seperately   */
/* rather than as a common to increase the final speed through use of less  */
/* calculations.                                                             */
/*****************************************************************************/
```

Monday May 05, 2003

May 02, 03 9:12 **SEALMacros.h** Page 2/5

```
#define ROTATE2(ui32)    (((ui32) >> 2) | ((ui32) << 30))

#define ROTATE9(ui32)    (((ui32) >> 9) | ((ui32) << 23))

#define ROTATE8(ui32)    (((ui32) >> 8) | ((ui32) << 24))

#define ROTATE16(ui32)   (((ui32) >> 16) | ((ui32) << 16))

#define ROTATE24(ui32)   (((ui32) >> 24) | ((ui32) << 8))

#define ROTATE27(ui32)   (((ui32) >> 27) | ((ui32) << 5))

#define ROTATE31(ui32)   (((ui32) >> 31) | ((ui32) << 1))

/*****************************************************************************/
/* SEAL_GKEYF1(), SEAL_GKEYF2(), SEAL_GKEYF3(), SEAL_GKEYF4(), SEAL_GKEYC1,  */
/* SEAL_GKEYC2, SEAL_GKEYC3, SEAL_GKEYC4                                     */
/*                                                                           */
/* These macros implement helper functions and helper constants as used by  */
/* the SEAL_GKEY() macro.                                                    */
/*****************************************************************************/
#define SEAL_GKEYF1(x, y, z)    (((x) & (y)) | ((~(x)) & (z)))

#define SEAL_GKEYF2(x, y, z)    ((x) ^ (y) ^ (z))

#define SEAL_GKEYF3(x, y, z)    (((x) & (y)) | ((x) & (z)) | ((y) & (z)))

#define SEAL_GKEYF4(x, y, z)    ((x) ^ (y) ^ (z))

#define SEAL_GKEYC1             0x5a827999

#define SEAL_GKEYC2             0x6ed9eba1

#define SEAL_GKEYC3             0x8f1bbcdc

#define SEAL_GKEYC4             0xca62c1d6

/*****************************************************************************/
/* SEAL_GKEY()                                                               */
/*                                                                           */
/* This macro calculates the GKey function as used when setting up the R, S */
/* and T tables while initialising the SEAL Cipher.  The macro takes two    */
/* input parameters (a 160-bit secret key and a 32-bit selection value) and */
/* returns a 160-bit output hash value, we return this in the pui32Result   */
/* array.                                                                    */
/*****************************************************************************/
#define SEAL_GKEY(pui32Key, ui32Selector, pui32Result)                  \
    {                                                                   \
        uint32      ui32H0, ui32H1, ui32H2, ui32H3, ui32H4;             \
        uint32      ui32A, ui32B, ui32C, ui32D, ui32E;                  \
        uint32      pui32X[80], ui32Temp, *pui32ResArray;               \
        int         iLoop;                                              \
                                                                        \
        pui32ResArray = pui32Result;                                    \
                                                                        \
        ui32A = ui32H0 = pui32Key[0];                                   \
        ui32B = ui32H1 = pui32Key[1];                                   \
        ui32C = ui32H2 = pui32Key[2];                                   \
        ui32D = ui32H3 = pui32Key[3];                                   \
        ui32E = ui32H4 = pui32Key[4];                                   \
                                                                        \
        pui32X[0] = ui32Selector;                                       \
        for (iLoop = 1; iLoop < 80; iLoop++)                            \
        {                                                               \
            pui32X[iLoop] = 0;                                          \
```

SEALMacros.h 1/3

Printed by Jason But

May 02, 03 9:12 **SEALMacros.h** Page 3/5

```
            if (iLoop >= 16)                                            \
                pui32X[iLoop] = ROTATE31(pui32X[iLoop - 3] ^            \
                                         pui32X[iLoop - 8] ^            \
                                         pui32X[iLoop - 14] ^           \
                                         pui32X[iLoop - 16]);           \
        }                                                               \
                                                                        \
        for (iLoop = 0; iLoop < 20; iLoop++)                            \
        {                                                               \
            ui32Temp = ROTATE27(ui32A) + SEAL_GKEYF1(ui32B, ui32C, ui32D) + \
                       ui32E + pui32X[iLoop] + SEAL_GKEYC1;             \
            ui32E = ui32D; ui32D = ui32C; ui32C = ROTATE2(ui32B);       \
            ui32B = ui32A; ui32A = ui32Temp;                            \
        }                                                               \
                                                                        \
        for (iLoop = 20; iLoop < 40; iLoop++)                           \
        {                                                               \
            ui32Temp = ROTATE27(ui32A) + SEAL_GKEYF2(ui32B, ui32C, ui32D) + \
                       ui32E + pui32X[iLoop] + SEAL_GKEYC2;             \
            ui32E = ui32D; ui32D = ui32C; ui32C = ROTATE2(ui32B);       \
            ui32B = ui32A; ui32A = ui32Temp;                            \
        }                                                               \
                                                                        \
        for (iLoop = 40; iLoop < 60; iLoop++)                           \
        {                                                               \
            ui32Temp = ROTATE27(ui32A) + SEAL_GKEYF3(ui32B, ui32C, ui32D) + \
                       ui32E + pui32X[iLoop] + SEAL_GKEYC3;             \
            ui32E = ui32D; ui32D = ui32C; ui32C = ROTATE2(ui32B);       \
            ui32B = ui32A; ui32A = ui32Temp;                            \
        }                                                               \
                                                                        \
        for (iLoop = 60; iLoop < 80; iLoop++)                           \
        {                                                               \
            ui32Temp = ROTATE27(ui32A) + SEAL_GKEYF4(ui32B, ui32C, ui32D) + \
                       ui32E + pui32X[iLoop] + SEAL_GKEYC4;             \
            ui32E = ui32D; ui32D = ui32C; ui32C = ROTATE2(ui32B);       \
            ui32B = ui32A; ui32A = ui32Temp;                            \
        }                                                               \
                                                                        \
        pui32ResArray[0] = ui32H0 + ui32A;                              \
        pui32ResArray[1] = ui32H1 + ui32B;                              \
        pui32ResArray[2] = ui32H2 + ui32C;                              \
        pui32ResArray[3] = ui32H3 + ui32D;                              \
        pui32ResArray[4] = ui32H4 + ui32E;                              \
    }

/*****************************************************************************/
/* SEAL_CONTINUOUSGKEY()                                                     */
/*                                                                           */
/* This macro fills an output array with consecutive results of calling the */
/* SEAL_GKEY macro.  The input parameters include a 160-bit key, the start  */
/* and end selection values for the loop and a pointer to the table to fill */
/* with the results.                                                         */
/*****************************************************************************/
#define SEAL_CONTINUOUSGKEY(pui32Key, iStart, iEnd, pui32Array)         \
    {                                                                   \
        uint32      *pui32Temp = pui32Array;                            \
        for (int iCount = iStart; iCount <= iEnd; iCount++, pui32Temp+= 5) \
            SEAL_GKEY(pui32Key, iCount, pui32Temp);                     \
    }

/*****************************************************************************/
/* SEAL_FILLTTABLE(), SEAL_FILLSTABLE(), SEAL_FILLRTABLE()                   */
/*                                                                           */
/* These macros are hardcoded to call the SEAL_CONTINUOUSGKEY() macro to    */
```

Monday May 05, 2003

May 02, 03 9:12 **SEALMacros.h** Page 4/5

```
/* correctly fill the T, S, and R arrays given the key and a pointer to the */
/* tables to fill.  Note that the S Array actually begins one 32-bit word   */
/* into the filled table and the R Array actually begins two 32-bit words   */
/* into the filled table.                                                    */
/*****************************************************************************/
#define SEAL_FILLTTABLE(pui32Key, pui32Array)                           \
    SEAL_CONTINUOUSGKEY(pui32Key, 0, 102, pui32Array)

#define SEAL_FILLSTABLE(pui32Key, pui32Array)                           \
    SEAL_CONTINUOUSGKEY(pui32Key, 819, 870, pui32Array)

#define SEAL_FILLRTABLE(pui32Key, iMaxOuterLoops, pui32Array)           \
    SEAL_CONTINUOUSGKEY(pui32Key, 1638,                                 \
                        1638 + ((4 * iMaxOuterLoops + 1) / 5), pui32Array)

/*****************************************************************************/
/* SEAL_SETADD(), SEAL_SETXOR(), SEAL_UPDADD(), SEAL_UPDXOR()                */
/*                                                                           */
/* These macros perform smaller functions used repetitively inside the SEAL */
/* pseudo-random number generation process.                                 */
/*****************************************************************************/
#define SEAL_SETADD(ui32Temp, ui32Param1, ui32Param2, pui32TArray)      \
    ui32Temp    = ui32Param1 & 0x7fc;                                   \
    ui32Param2 += pui32TArray[ui32Temp >> 2];                           \
    ui32Param1  = ROTATE9(ui32Param1);

#define SEAL_SETXOR(ui32Temp, ui32Param1, ui32Param2, pui32TArray)      \
    ui32Temp    = ui32Param1 & 0x7fc;                                   \
    ui32Param2 ^= pui32TArray[ui32Temp >> 2];                           \
    ui32Param1  = ROTATE9(ui32Param1);

#define SEAL_UPDADD(ui32Temp, ui32Param1, ui32Param2, pui32TArray)      \
    ui32Temp    = (ui32Temp + ui32Param1) & 0x7fc;                      \
    ui32Param2 += pui32TArray[ui32Temp >> 2];                           \
    ui32Param1  = ROTATE9(ui32Param1);

#define SEAL_UPDXOR(ui32Temp, ui32Param1, ui32Param2, pui32TArray)      \
    ui32Temp    = (ui32Temp + ui32Param1) & 0x7fc;                      \
    ui32Param2 ^= pui32TArray[ui32Temp >> 2];                           \
    ui32Param1  = ROTATE9(ui32Param1);

/*****************************************************************************/
/* SEAL_OUTERLOOP()                                                          */
/*                                                                           */
/* This macro performs the calculations as per the Outer Loop of the SEAL   */
/* psuedo-random generation algorithm.  It prepares the values of A, B, C, D, */
/* N1, N2, N3 and N4 based on the R and T tables calculated by the Private  */
/* Key, the current Resynchronisation value and the pass number through the */
/* outer loop.                                                               */
/*****************************************************************************/
#define SEAL_OUTERLOOP(iOuterLoopCount, ui32Sync, pui32RArray, pui32TArray, \
                       ui32A, ui32B, ui32C, ui32D, ui32N1, ui32N2, ui32N3,  \
                       ui32N4)                                          \
    {                                                                   \
        uint32      ui32Temp;                                           \
                                                                        \
        ui32A = ui32Sync ^ pui32RArray[iOuterLoopCount << 2];           \
        ui32B = ROTATE8(ui32Sync) ^ pui32RArray[(iOuterLoopCount << 2) + 1]; \
        ui32C = ROTATE16(ui32Sync) ^ pui32RArray[(iOuterLoopCount << 2) + 2]; \
        ui32D = ROTATE24(ui32Sync) ^ pui32RArray[(iOuterLoopCount << 2) + 3]; \
                                                                        \
        SEAL_SETADD(ui32Temp, ui32A, ui32B, pui32TArray);               \
        SEAL_SETADD(ui32Temp, ui32B, ui32C, pui32TArray);               \
        SEAL_SETADD(ui32Temp, ui32C, ui32D, pui32TArray);               \
        SEAL_SETADD(ui32Temp, ui32D, ui32A, pui32TArray);               \
```

SEALMacros.h 2/3

```
        SEAL_SETADD(ui32Temp, ui32A, ui32B, pui32TArray);                  \
        SEAL_SETADD(ui32Temp, ui32B, ui32C, pui32TArray);                  \
        SEAL_SETADD(ui32Temp, ui32C, ui32D, pui32TArray);                  \
        SEAL_SETADD(ui32Temp, ui32D, ui32A, pui32TArray);                  \
                                                                           \
        ui32N1 = ui32D; ui32N2 = ui32B; ui32N3 = ui32A; ui32N4 = ui32C;    \
                                                                           \
        SEAL_SETADD(ui32Temp, ui32A, ui32B, pui32TArray);                  \
        SEAL_SETADD(ui32Temp, ui32B, ui32C, pui32TArray);                  \
        SEAL_SETADD(ui32Temp, ui32C, ui32D, pui32TArray);                  \
        SEAL_SETADD(ui32Temp, ui32D, ui32A, pui32TArray);                  \
    }

/****************************************************************************/
/* SEAL_INNERLOOP()                                                         */
/*                                                                          */
/* This macro performs the calculations as per the Inner Loop of the SEAL   */
/* psuedo-random generation algorithm.  It updates the values of A, B, C, D, */
/* N1, N3, N3 and N4 based on the S and T tables calculated by the Private  */
/* Key and the pass number through the inner loop.  The resultant 16 bytes of */
/* psuedo-random string are stored in the pui32Output array.                */
/****************************************************************************/
#define SEAL_INNERLOOP(iInnerLoopCount, pui32SArray, pui32TArray, ui32A,   \
                       ui32B, ui32C, ui32D, ui32N1, ui32N2, ui32N3, ui32N4, \
                       pui32Output)                                        \
    {                                                                      \
        uint32      ui32Temp1, ui32Temp2;                                  \
                                                                           \
        SEAL_SETADD(ui32Temp1, ui32A, ui32B, pui32TArray); ui32B ^= ui32A; \
        SEAL_SETXOR(ui32Temp2, ui32B, ui32C, pui32TArray); ui32C += ui32B; \
        SEAL_UPDADD(ui32Temp1, ui32C, ui32D, pui32TArray); ui32D ^= ui32C; \
        SEAL_UPDXOR(ui32Temp2, ui32D, ui32A, pui32TArray); ui32A += ui32D; \
        SEAL_UPDXOR(ui32Temp1, ui32A, ui32B, pui32TArray);                 \
        SEAL_UPDADD(ui32Temp2, ui32B, ui32C, pui32TArray);                 \
        SEAL_UPDXOR(ui32Temp1, ui32C, ui32D, pui32TArray);                 \
        SEAL_UPDADD(ui32Temp2, ui32D, ui32A, pui32TArray);                 \
                                                                           \
        pui32Output[0] = ui32B + pui32S[iInnerLoopCount << 2];             \
        pui32Output[1] = ui32C ^ pui32S[(iInnerLoopCount << 2) + 1];       \
        pui32Output[2] = ui32D + pui32S[(iInnerLoopCount << 2) + 2];       \
        pui32Output[3] = ui32A ^ pui32S[(iInnerLoopCount << 2) + 3];       \
                                                                           \
        if (iInnerLoopCount & 0x01)                                        \
        {                                                                  \
            ui32A+= ui32N1; ui32B+= ui32N2; ui32C^= ui32N1; ui32D^= ui32N2; \
        } else                                                             \
        {                                                                  \
            ui32A+= ui32N3; ui32B+= ui32N4; ui32C^= ui32N3; ui32D^= ui32N4; \
        }                                                                  \
    }

#endif

/****************************************************************************/
/* End of File: SEALMacros.h                                                */
/****************************************************************************/
```

```
/****************************************************************************/
/* File: SEALStreamCipher.cpp                                               */
/****************************************************************************/
/* This program file contains the definition and implementation of the the  */
/* SEALStreamCipher class.  This class is derived from the virtual base class */
/* StreamCipherBase and implements the prototype SEAL Cipher for use in     */
/* encryption of Streaming MPEG-1 Video and Audio.                          */
/****************************************************************************/
/* NOTE:                                                                    */
/****************************************************************************/
/* Written By:  Jason But                                                   */
/* Date:        1999-2003                                                   */
/* For:         PhD Thesis in Streaming Video Encryption.                   */
/*              Centre for Telecommunications and Information Engineering.  */
/*              Monash University Australia.                                */
/****************************************************************************/

/****************************************************************************/
/* Include definition of base class and ClassFactory templates.             */
/****************************************************************************/
#include "StreamCipherBase.h"
#include "ClassFactory.h"

/****************************************************************************/
/* Include macros for implementation of the SEAL algorithm.                 */
/****************************************************************************/
#include "SEALMacros.h"

/****************************************************************************/
/* Defined constants to avoid magic numbers.                                */
/*                                                                          */
/* The first three macros define information presented to the code using this */
/* cipher.  The last macro defines the maximal length of a pseudo-random    */
/* string produced by a given synchronisation code, in this case 64kB.  The */
/* cipher implementation will repeat the string if more 65536 random bytes  */
/* are required.                                                            */
/****************************************************************************/
#define SEAL_INDEX          1
#define SEAL_KEYLENGTH      160
#define SEAL_DESCRIPTION    "SEAL"
#define SEAL_MAXOUTERLOOPS  64

/****************************************************************************/
/* class SEALStreamCipher.                                                  */
/****************************************************************************/
/* Interface.                                                               */
/*                                                                          */
/* This class provides a SEAL cipher which can be used to provide two       */
/* seperate psuedo-random streams as required for MPEG-1 Video and Audio    */
/* stream encryption.  Are number of protected member variables are defined */
/* to store SEAL state information during random-string generation.  The    */
/* overloaded resyncuronisation methods simply reset some state variables to */
/* ensure consistent pseudo-random string generation.  Implementation details */
/* for other virtual methods are decribed in the implementation comment     */
/* blocks.                                                                  */
/****************************************************************************/
class SEALStreamCipher : public StreamCipherBase
{
    protected:
        // R, S and T tables used to generate pseudo-random values
        SEAL_DEFINETTABLE(pui32TTable);
        SEAL_DEFINESTABLE(pui32STable);
        SEAL_DEFINERTABLE(pui32RTable, SEAL_MAXOUTERLOOPS);

        uint32          *pui32R, *pui32S, *pui32T;
```

```
        // Loop counters and variable storage for generating Audio pseudo-random bytes
        int             iAudioByteCount, iAudioInnerCount, iAudioOuterCount;
        uint32          ui32AudioA, ui32AudioB, ui32AudioC, ui32AudioD;
        uint32          ui32AudioN1, ui32AudioN2, ui32AudioN3, ui32AudioN4;
        uint32          ui32AudioSyncValue, pui32AudioRandomBlock[4];

        // Loop counters and variable storage for generating Audio pseudo-random bytes
        int             iVideoByteCount, iVideoInnerCount, iVideoOuterCount;
        uint32          ui32VideoA, ui32VideoB, ui32VideoC, ui32VideoD;
        uint32          ui32VideoN1, ui32VideoN2, ui32VideoN3, ui32VideoN4;
        uint32          ui32VideoSyncValue, pui32VideoRandomBlock[4];

    public:
                        SEALStreamCipher();
        virtual         ~SEALStreamCipher() {}

        virtual void    ResynchAudioCipher(uint32 ui32Value)
        {
            ui32AudioSyncValue = ui32Value;
            iAudioByteCount = iAudioInnerCount = iAudioOuterCount = 0;
        }
        virtual void    ResynchVideoCipher(uint32 ui32Value)
        {
            ui32VideoSyncValue = ui32Value;
            iVideoByteCount = iVideoInnerCount = iVideoOuterCount = 0;
        }

        virtual byte    GetNextRandomAudioByte();
        virtual byte    GetNextRandomVideoByte();

        virtual void    SetKey(byte *pbPrivateKey);
};

/****************************************************************************/
/* Cipher Registration.                                                     */
/*                                                                          */
/* Registration of the Stream Cipher with the Class Factory.                */
/****************************************************************************/
namespace
{
    RegisterInFactory<StreamCipherBase, SEALStreamCipher> regSEALCipher(SEAL_INDEX);
}

/****************************************************************************/
/* Implementation.                                                          */
/****************************************************************************/
/* Constructor.                                                             */
/*                                                                          */
/* This is the constructor for the SEALStreamCipher class, we obtain and    */
/* store the offsets of the T, S and R tables for the SEAL cipher.          */
/****************************************************************************/
SEALStreamCipher::SEALStreamCipher() : StreamCipherBase()
{
    pui32T = SEAL_TTABLEOFFSET(pui32TTable);
    pui32S = SEAL_STABLEOFFSET(pui32STable);
    pui32R = SEAL_RTABLEOFFSET(pui32RTable);
}

/****************************************************************************/
/* void SetKey(byte *pbPrivateKey)                                          */
/*                                                                          */
/* This method is called to reset the cipher with a new Private Key.  For the */
/* SEAL Cipher this involves first recalculating the values of the T, S and R */
/* tables and then resynchronising the ciphers private internal values.  The */
```

May 02, 03 9:12 **SEALStreamCipher.cpp** Page 3/4

```
/* tables are filled via calls to the SEAL_FILLxTABLE() macros, the ciphers    */
/* are resynchronised via calls to the ResychXXXCipher() methods.              */
/*******************************************************************************/
void
SEALStreamCipher::SetKey(byte *pbPrivateKey)
{
    uint32      *pui32Key = (uint32 *)pbPrivateKey;

    SEAL_FILLTTABLE(pui32Key, pui32TTable);

    SEAL_FILLSTABLE(pui32Key, pui32STable);

    SEAL_FILLRTABLE(pui32Key, SEAL_MAXOUTERLOOPS, pui32RTable);

    ResynchAudioCipher(0);
    ResynchVideoCipher(0);
}

/*******************************************************************************/
/* byte GetNextRandomAudioByte()                                               */
/*                                                                             */
/* This virtual method is called to return the next pseudo-random output byte */
/* produced by the SEAL Cipher for encryption of the audio stream.  The SEAL  */
/* algorithm is usually presented as two loops producing a large array filled */
/* with pseudo-random values.  Since we need the output just a byte at a time,*/
/* we need to turn the algorithm inside out.  We maintain the loop counters   */
/* in class member variables.                                                  */
/*  - Starting with the innermost loop, this produces 4 32-bit pseudo-random  */
/*    values which are stored in the pui32AudioRandomBlock array.  If our     */
/*    position counter into this array is not zero, then the array contains   */
/*    random values we can return.  We obtain this indexed value from the     */
/*    array and increment the position counter.  If this counter is now 16,   */
/*    we have passed the end of the array and so reset it to zero.            */
/*  - In the initial condition the random array position counter is zero, it  */
/*    is also zero when we have consumed all entries in the array.  This      */
/*    indicates that we have to calculate a new set of pseudo-random values.  */
/*    If the Inner Loop counter is not zero then we are nominally inside the  */
/*    inner loop of the SEAL algorithm.  We execute the code for the inner    */
/*    loop via a call to the SEAL_INNERLOOP() macro, this will fill the       */
/*    pui32AudioRandomBlock array.  Finally, we increment the inner loop      */
/*    counter, if it is equal to 64 we should terminate the inner loop for    */
/*    the next pass so we reset the inner loop counter to zero.  Now that the */
/*    array of random values contains data we can return the first byte and  */
/*    increments the position counter as per above.                           */
/*  - In the initial condition the inner loop counter is zero, it is also     */
/*    zero when we complete all 64 cycles of the inner loop.  In this case we */
/*    are nominally inside the Outer Loop of the SEAL algorithm.  We execute  */
/*    the code for the outer loop via a call to the SEAL_OUTERLOOP() macro,   */
/*    initialising variables for the inner loop.  We also increment the outer */
/*    loop counter, if it is equal to SEAL_MAXOUTERLOOPS, we reset this       */
/*    counter to zero.  This indicates that the next time that the inner loop */
/*    counter is zero, the generation of random bytes will reset to produce   */
/*    the same sequence as long as the key is not reset.  The number of bytes */
/*    generated before the sequence is repeated is equal to SEAL_MAXOUTERLOOPS*/
/*    * 1024.  Once the outer loop is initialised we then proceed to execute  */
/*    the inner loop code as outlined above.                                  */
/*******************************************************************************/
byte
SEALStreamCipher::GetNextRandomAudioByte()
{
    byte    bReturnValue;

    if (iAudioByteCount == 0)
    {
        if (iAudioInnerCount == 0)
```

May 02, 03 9:12 **SEALStreamCipher.cpp** Page 4/4

```
        {
            SEAL_OUTERLOOP(iAudioOuterCount, ui32AudioSyncValue, pui32R, pui32T,
                           ui32AudioA, ui32AudioB, ui32AudioC, ui32AudioD, ui32AudioN1,
                           ui32AudioN2, ui32AudioN3, ui32AudioN4);

            iAudioOuterCount++;
            if (iAudioOuterCount == SEAL_MAXOUTERLOOPS) iAudioOuterCount = 0;
        }

        SEAL_INNERLOOP(iAudioInnerCount, pui32S, pui32T, ui32AudioA, ui32AudioB,
                       ui32AudioC, ui32AudioD, ui32AudioN1, ui32AudioN2, ui32AudioN3,
                       ui32AudioN4, pui32AudioRandomBlock);

        iAudioInnerCount++;
        iAudioInnerCount &= 0x3f;
    }
    bReturnValue = ((byte *) pui32AudioRandomBlock)[iAudioByteCount++];
    iAudioByteCount &= 0x0f;
    return bReturnValue;
}

/*******************************************************************************/
/* byte GetNextRandomVideoByte()                                               */
/*                                                                             */
/* This virtual method is called to return the next pseudo-random output byte */
/* produced by the SEAL Cipher for encryption of the video stream.  The code  */
/* is identical to that for GetNextRandomAudioByte() except that a different  */
/* set of state member variables are used.                                    */
/*******************************************************************************/
byte
SEALStreamCipher::GetNextRandomVideoByte()
{
    byte    bReturnValue;

    if (iVideoByteCount == 0)
    {
        if (iVideoInnerCount == 0)
        {
            SEAL_OUTERLOOP(iVideoOuterCount, ui32VideoSyncValue, pui32R, pui32T,
                           ui32VideoA, ui32VideoB, ui32VideoC, ui32VideoD, ui32VideoN1,
                           ui32VideoN2, ui32VideoN3, ui32VideoN4);

            iVideoOuterCount++;
            if (iVideoOuterCount == SEAL_MAXOUTERLOOPS) iVideoOuterCount = 0;
        }

        SEAL_INNERLOOP(iVideoInnerCount, pui32S, pui32T, ui32VideoA, ui32VideoB,
                       ui32VideoC, ui32VideoD, ui32VideoN1, ui32VideoN2, ui32VideoN3,
                       ui32VideoN4, pui32VideoRandomBlock);

        iVideoInnerCount++;
        iVideoInnerCount &= 0x3f;
    }
    bReturnValue = ((byte *) pui32VideoRandomBlock)[iVideoByteCount++];
    iVideoByteCount &= 0x0f;
    return bReturnValue;
}

/*******************************************************************************/
/* End of File: SEALStreamCipher.cpp                                           */
/*******************************************************************************/
```

May 02, 03 9:12 **MPEGVideoParser.h** Page 1/2

```
/*******************************************************************************/
/* File: MPEGVideoParser.h                                                     */
/*******************************************************************************/
/* This program file contains the definition of the the MPEGVideoParser class.*/
/* This class can process an Mpeg Video Stream for the purposes of either     */
/* encryption or decryption.  the class requires access to a fully            */
/* synchronised stream cipher.                                                 */
/*******************************************************************************/
/* NOTE:                                                                       */
/*       This file contains some Conditional code which is compiled only if   */
/*       the CALC_CIPHER_STATS macro is defined during compile time.  This is */
/*       true when compiling the 'MPEG1 Cipher' application but not the 'MPEG1 */
/*       Cipher Filter' DirectShow filter.  This code compiles statistics     */
/*       during the cipher process.                                            */
/*******************************************************************************/
/* Written By:  Jason But                                                      */
/* Date:        1999-2003                                                      */
/* For:         PhD Thesis in Streaming Video Encryption.                      */
/*              Centre for Telecommunications and Information Engineering.    */
/*              Monash University Australia.                                   */
/*******************************************************************************/

/*******************************************************************************/
/* Check to see if already included.                                           */
/*******************************************************************************/
#ifndef _MPEGVIDEOPARSER_H_
#define _MPEGVIDEOPARSER_H_

/*******************************************************************************/
/* Forward definition of StreamCupherBase class.                               */
/*******************************************************************************/
class StreamCipherBase;

/*******************************************************************************/
/* Number of states in the MPEG Audio Parser State Machine.                    */
/*******************************************************************************/
#define VIDEO_NUM_STATES        21

/*******************************************************************************/
/* Conditional code.                                                           */
/*                                                                             */
/* The following code is compiled only if the CALC_CIPHER_STATS macro is     */
/* defined during compile time.  This is true when compiling the 'MPEG1      */
/* Cipher' application but not the 'MPEG1 Cipher Filter' DirectShow filter.  */
/* The macros defined in this section define the variables used to maintain  */
/* statistics for the parsing module, otherwise the macros resolve to no code.*/
/*******************************************************************************/
#ifdef CALC_CIPHER_STATS

#define DEFINE_VARIABLES()      long    lBytesProcessed, lBytesSelected, lBytesEncrypted

#else

#define DEFINE_VARIABLES()

#endif

/*******************************************************************************/
/* class MPEGVideoParser.                                                      */
/*                                                                             */
/* This class will parse an MPEG-1 Video Stream in blocks, selecting bytes    */
/* from the stream to pass to the cipher.  The cipher is a modified stream    */
/* cipher pointed to by pcCipher - which produces pseudo-random bytes - the   */
/* modifications ensure against creation of false MPEG-1 Headers, details of  */
/* the modifications can be found in the corresponding source code file.  The */
```

May 02, 03 9:12 **MPEGVideoParser.h** Page 2/2

```
/* public member methods of this class are:                                   */
/*                                                                             */
/* Constructor          : Creates and initialises the state machine used to   */
/*                        parse the binary stream.  Registers the provided    */
/*                        cipher to be used to generated random values.       */
/* ResetStateMachine()  : Resets the State Machine to the initial state, used */
/*                        during indexed playback to reset the module to ensure*/
/*                        correct parsing of the newly indexed stream.        */
/* ParseBuffer()        : Parses a block of data from the Video Stream, the   */
/*                        state is remembered at the end of the data block so */
/*                        that processing can continue on the next provided   */
/*                        data block.                                         */
/*******************************************************************************/
class MPEGVideoParser
{
    protected:
        int                 iState;
        int                 ppiStateLookup[VIDEO_NUM_STATES][256];
        long                lGOPTimeStamp, lPictureCount;
        StreamCipherBase    *pcCipher;

        void                InitialiseLookupTable();

    public:
                            MPEGVideoParser();
        void                RegisterCipher(StreamCipherBase *pcStreamCipher) { pcCipher = pcStreamCi
pher; }
        void                ResetStateMachine();
        void                ParseBuffer(byte *pbBuffer, long lBufferSize);

        DEFINE_VARIABLES();
};

/*******************************************************************************/
/* Un-define macros that are no longer needed.                                 */
/*******************************************************************************/
#undef  DEFINE_VARIABLES

#endif

/*******************************************************************************/
/* End of File: MPEGVideoParser.h                                              */
/*******************************************************************************/
```

```
/*****************************************************************************/
/* File: MPEGVideoParser.cpp                                                 */
/*****************************************************************************/
/* This program file contains the implementation of the the MPEGVideoParser  */
/* class.  This class can process an Mpeg Video Stream for the purposes of   */
/* either encryption or decryption.  the class requires access to a fully    */
/* synchronised stream cipher.                                               */
/*****************************************************************************/
/* NOTE:                                                                     */
/*      This file contains some Conditional code which is compiled only if   */
/*      the CALC_CIPHER_STATS macro is defined during compile time.  This is */
/*      true when compiling the 'MPEG1 Cipher' application but not the 'MPEG1 */
/*      Cipher Filter' DirectShow filter.  This code compiles statistics     */
/*      during the cipher process.                                           */
/*****************************************************************************/
/* Written By:  Jason But                                                    */
/* Date:        1999-2003                                                    */
/* For:         PhD Thesis in Streaming Video Encryption.                    */
/*              Centre for Telecommunications and Information Engineering.   */
/*              Monash University Australia.                                 */
/*****************************************************************************/

/*****************************************************************************/
/* Include Standard C Libraries.                                             */
/*****************************************************************************/
#include <stdlib.h>

/*****************************************************************************/
/* Definition of StreamCipherBase class.                                     */
/*****************************************************************************/
#include "StreamCipherBase.h"

/*****************************************************************************/
/* Include class header file.                                                */
/*****************************************************************************/
#include "MPEGVideoParser.h"

/*****************************************************************************/
/* Conditional code.                                                         */
/*                                                                           */
/* The following code is compiled only if the CALC_CIPHER_STATS macro is     */
/* defined during compile time.  This is true when compiling the 'MPEG1      */
/* Cipher' application but not the 'MPEG1 Cipher Filter' DirectShow filter.  */
/* The macros defined in this section either reset or update statistic       */
/* counters for the parsing module, otherwise the macros resolve to no code. */
/*****************************************************************************/
#ifdef CALC_CIPHER_STATS

#define RESET_VARIABLES()           lBytesProcessed = lBytesSelected = lBytesEncrypted = 0
#define INC_PROCESSED(lAmount)      lBytesProcessed+= lAmount
#define INC_SELECTED()              lBytesSelected++
#define INC_ENCRYPTED()             lBytesEncrypted++

#else

#define RESET_VARIABLES()
#define INC_PROCESSED(lAmount)
#define INC_SELECTED()
#define INC_ENCRYPTED()

#endif

/*****************************************************************************/
/* MPEG Video Stream Encryption                                              */
/*                                                                           */
```

```
/* Partial Selection of Video Stream for Encryption.                         */
/*                                                                           */
/* An MPEG-1 Video Stream can be broken into a sequence of successive binary */
/* blocks called Sequences, each Sequence can be further sub-divided into a  */
/* series of Groups of Pictures (GOPs).  Each GOP is further constructed of a */
/* series of Pictures which are in turn made of a series of Slices which are */
/* formed by a group of MacroBlocks.  The header for each of these constructs */
/* is always byte aligned and begins with the three byte sequence 00-00-01.  */
/* The encryption algorithm outlined in my thesis selects the Slice payload  */
/* for encryption.  However we must also ensure that we do not create any    */
/* false MPEG-1 Headers nor modify any existing headers.  This is done by not */
/* encrypting any byte that has either of the values 0x00 or 0x01.  We also */
/* ensure that no byte encrypts to either of these two values.  Outlined     */
/* is a State Machine which will parse an MPEG-1 Video Stream and select     */
/* bytes for encryption.                                                     */
/*                                                                           */
/*   STATE      INPUT        NEXT STATE    ACTION                            */
/* ------------------------------------------------------------------------- */
/*    1-1       0x00         1-2           .                                 */
/*              default      1-1           .                                 */
/*                                                                           */
/*    1-2       0x00         1-3           .                                 */
/*              default      1-1           .                                 */
/*                                                                           */
/*    1-3       0x00         1-3           .                                 */
/*              0x01         1-4           .                                 */
/*              default      1-1           .                                 */
/*                                                                           */
/*    1-4       0x00         1-9           (Picture Header).                 */
/*              0xb8         1-5           (GOP Header).                     */
/*              0x01-0xaf    2-6           (Slice Header).                   */
/*              default      1-1           .                                 */
/*                                                                           */
/*    1-5       default      1-6           INPUT & 0x1f = 5 MSB of Timestamp.*/
/*                                                                           */
/*    1-6       default      1-7           Next 8 Bits of Timestamp.         */
/*                                                                           */
/*    1-7       default      1-8           Next 8 Bits of Timestamp.         */
/*                                                                           */
/*    1-8       default      1-1           INPUT & 0x80 = LSB of Timestamp.  */
/*                                                                           */
/*    1-9       default      1-10          8 MSB Bits of Picture Count.      */
/*                                                                           */
/*    1-10      default      1-1           INPUT & 0xc0 = 2 LSB of Pic Count */
/*                                         Resynchronise Stream Cipher.      */
/*                                                                           */
/*    2-1       1xxx-xxxx    2-2           .                                 */
/*              0xxx-xxxx    3-1           .                                 */
/*                                                                           */
/*    2-2       x1xx-xxxx    2-3           .                                 */
/*              x0xx-xxxx    3-1           .                                 */
/*                                                                           */
/*    2-3       xx1x-xxxx    2-4           .                                 */
/*              xx0x-xxxx    3-1           .                                 */
/*                                                                           */
/*    2-4       xxx1-xxxx    2-5           .                                 */
/*              xxx0-xxxx    3-1           .                                 */
/*                                                                           */
/*    2-5       xxxx-1xxx    2-6           .                                 */
/*              xxxx-0xxx    3-1           .                                 */
/*                                                                           */
/*    2-6       xxxx-x1xx    2-7           .                                 */
/*              xxxx-x0xx    3-1           .                                 */
/*                                                                           */
/*    2-7       xxxx-xx1x    2-8           .                                 */
```

```
/*              xxxx-xx0x    3-1           .                                 */
/*                                                                           */
/*    2-8       xxxx-xxx1    2-1           .                                 */
/*              xxxx-xxx0    3-1           .                                 */
/*                                                                           */
/*    3-1       0x00         3-2           Pass INPUT through cipher.        */
/*              default      3-1           Pass INPUT through cipher.        */
/*                                                                           */
/*    3-2       0x00         3-3           Pass INPUT through cipher.        */
/*              default      3-1           Pass INPUT through cipher.        */
/*                                                                           */
/*    3-3       0x00         3-3           Pass INPUT through cipher.        */
/*              0x01         1-4           Pass INPUT through cipher.        */
/*              default      3-1           Pass INPUT through cipher.        */
/* ------------------------------------------------------------------------- */
/*                                                                           */
/* Resynchronisation occurs at the end of processing each Picture Header,    */
/* this ensures that the pseudo random stream generated is correct for       */
/* decryption during indexed and high-speed playback.  Details of how the    */
/* modified stream cipher does not produce any false MPEG-1 headers is       */
/* outlined in comments for the ParseBuffer() method.                        */
/*****************************************************************************/

/*****************************************************************************/
/* Definition of State Names for human readable purposes.                    */
/*****************************************************************************/
#define STATE_1_1        0
#define STATE_1_2        1
#define STATE_1_3        2
#define STATE_1_4        3
#define STATE_1_5        4
#define STATE_1_6        5
#define STATE_1_7        6
#define STATE_1_8        7
#define STATE_1_9        8
#define STATE_1_10       9
#define STATE_2_1        10
#define STATE_2_2        11
#define STATE_2_3        12
#define STATE_2_4        13
#define STATE_2_5        14
#define STATE_2_6        15
#define STATE_2_7        16
#define STATE_2_8        17
#define STATE_3_1        18
#define STATE_3_2        19
#define STATE_3_3        20

/*****************************************************************************/
/* Constructor.                                                              */
/*                                                                           */
/* This is the constructor for the MPEGVideoParser class.  We initialise the */
/* State Machine Lookup Table before resetting the current state.            */
/*****************************************************************************/
MPEGVideoParser::MPEGVideoParser() : pcCipher(NULL)
{
    InitialiseLookupTable();
    ResetStateMachine();
}

/*****************************************************************************/
/* void InitialiseLookupTable()                                              */
/*                                                                           */
/* This method is called to initialise the lookup table to confer with the   */
/* state machine described above.                                            */
```

```
/*****************************************************************************/
void
MPEGVideoParser::InitialiseLookupTable()
{
    for (int iInput = 0x00; iInput <= 0xff; iInput++)
    {
        ppiStateLookup[STATE_1_1][iInput] = (iInput)?STATE_1_1:STATE_1_2;
        ppiStateLookup[STATE_1_2][iInput] = (iInput)?STATE_1_1:STATE_1_3;
        ppiStateLookup[STATE_1_3][iInput] = (iInput)?STATE_1_1:STATE_1_3;
        ppiStateLookup[STATE_1_4][iInput] = (iInput <= 0xaf)?STATE_2_6:STATE_1_1;
        ppiStateLookup[STATE_1_5][iInput] = STATE_1_6;
        ppiStateLookup[STATE_1_6][iInput] = STATE_1_7;
        ppiStateLookup[STATE_1_7][iInput] = STATE_1_8;
        ppiStateLookup[STATE_1_8][iInput] = STATE_1_1;
        ppiStateLookup[STATE_1_9][iInput] = STATE_1_10;
        ppiStateLookup[STATE_1_10][iInput] = STATE_1_1;

        ppiStateLookup[STATE_2_1][iInput] = (iInput & 0x80)?STATE_2_2:STATE_3_1;
        ppiStateLookup[STATE_2_2][iInput] = (iInput & 0x40)?STATE_2_3:STATE_3_1;
        ppiStateLookup[STATE_2_3][iInput] = (iInput & 0x20)?STATE_2_4:STATE_3_1;
        ppiStateLookup[STATE_2_4][iInput] = (iInput & 0x10)?STATE_2_5:STATE_3_1;
        ppiStateLookup[STATE_2_5][iInput] = (iInput & 0x08)?STATE_2_6:STATE_3_1;
        ppiStateLookup[STATE_2_6][iInput] = (iInput & 0x04)?STATE_2_7:STATE_3_1;
        ppiStateLookup[STATE_2_7][iInput] = (iInput & 0x02)?STATE_2_8:STATE_3_1;
        ppiStateLookup[STATE_2_8][iInput] = (iInput & 0x01)?STATE_2_1:STATE_3_1;

        ppiStateLookup[STATE_3_1][iInput] = (iInput)?STATE_3_1:STATE_3_2;
        ppiStateLookup[STATE_3_2][iInput] = (iInput)?STATE_3_1:STATE_3_3;
        ppiStateLookup[STATE_3_3][iInput] = (iInput)?STATE_3_1:STATE_3_3;
    }
    ppiStateLookup[STATE_1_3][0x01] = STATE_1_4;
    ppiStateLookup[STATE_1_4][0x00] = STATE_1_9;
    ppiStateLookup[STATE_1_4][0xb8] = STATE_1_5;
    ppiStateLookup[STATE_3_3][0x01] = STATE_1_4;
}

/*****************************************************************************/
/* void ResetStateMachine()                                                  */
/*                                                                           */
/* This method allows us to reset the state machine to the initial state of  */
/* STATE_1_1.                                                                */
/*****************************************************************************/
void
MPEGVideoParser::ResetStateMachine()
{
    iState = STATE_1_1;
    RESET_VARIABLES();
}

/*****************************************************************************/
/* void ParseBuffer(byte *pbBuffer, long lBufferSize)                        */
/*                                                                           */
/* This method parses a buffer containing a part of an MPEG Video Stream and */
/* passes the relevant bytes from the buffer to the cipher.  The method takes */
/* a pointer to the buffer and the length of that buffer.                    */
/* The partial selection procedure is outlined above, when the state machine */
/* is in States 3-1, 3-2 or 3-3, the bytes must be encrypted.  The cipher is */
/* a modified stream cipher, it will not generate a 0x00 or 0x01 byte, nor   */
/* will it modify an existing 0x00 or 0x01 byte, as these bytes are used as  */
/* markers for the State Machine to locate MPEG-1 Header Codes.  This is     */
/* achieved by checking the existing value (shifted right one bit) prior to  */
/* performing the XOR, if the shifted byte is 0x00 (original value of 0x00 or */
/* 0x01), or the generated random byte value shifted right through one bit   */
/* (original value of bRandom or bRandom XOR 0x01), the XOR is not performed. */
/* The first check ensures we do not change any existing 0x00 or 0x01 marker */
```

May 02, 03 9:12 **MPEGVideoParser.cpp** Page 5/5

```
/* bytes and the second ensures we do not create a false MPEG-1 Header Code.  */
/*  - bRandom ^ bRandom = 0x00                                                */
/*  - bRandom ^ 0x01 ^ bRandom = 0x01                                         */
/*                                                                            */
/* When the State Machine is in States 1-5 through 1-8, we are processing the */
/* timecode within a GOP Header and we store this value in the class variable */
/* lGOPTimeStamp.  When the value is fully read, it is set in the cipher.     */
/* When the State Machine is in States 1-9 and 1-10, we are processing the    */
/* Picture Count value in the Picture header and we store this value in the   */
/* class variable lPictureCount.  When the value is fully read, it is set in  */
/* the cipher and the stream cipher must be reset for synchronisation         */
/* purposes, ensuring production of the correct sequence of random bytes for  */
/* the following Slice payloads.                                              */
/* audio frame and are about to complete processing the header, the next byte */
/* will form part of the payload and must be passed through the cipher.  For  */
/* resynchronisation purposes, we must reset the stream cipher random number  */
/* generator to ensure we produce the correct sequence of random bytes.       */
/* After the cipher process is complete, we must update the current State,   */
/* this is done using the State Machine Lookup Table.                         */
/******************************************************************************/
void
MPEGVideoParser::ParseBuffer(byte *pbBuffer, long lBufferSize)
{
    byte    bTemp, bRandom, *pbInput, *pbEndBuffer = pbBuffer + lBufferSize;

    if (pcCipher == NULL) return;

    for(pbInput = pbBuffer; pbInput != pbEndBuffer; pbInput++)
    {
        switch (iState)
        {
            case STATE_1_5:     lGOPTimeStamp = (*pbInput & 0x1f) << 27; break;
            case STATE_1_6:     lGOPTimeStamp+= *pbInput << 19; break;
            case STATE_1_7:     lGOPTimeStamp+= *pbInput << 11; break;
            case STATE_1_8:     lGOPTimeStamp+= (*pbInput & 0x80) << 3; break;
            case STATE_1_9:     lPictureCount = *pbInput << 2; break;
            case STATE_1_10:    lPictureCount+= (*pbInput & 0xc0) >> 6;
                                pcCipher->ResynchVideoCipher(lGOPTimeStamp + lPictureCount);
                                break;
            case STATE_3_1:
            case STATE_3_2:
            case STATE_3_3:     bRandom = pcCipher->GetNextRandomVideoByte();
                                INC_SELECTED();
                                bTemp = (*pbInput) >> 1;
                                if ((bTemp) && (bTemp != (bRandom >> 1)))
                                {
                                    *pbInput ^= bRandom;
                                    INC_ENCRYPTED();
                                }
        }
        iState = ppiStateLookup[iState][*pbInput];
    }
    INC_PROCESSED(lBufferSize);
}

/******************************************************************************/
/* End of File: MPEGVideoParser.cpp                                           */
/******************************************************************************/
```

May 02, 03 9:12 **MPEGAudioParser.h** Page 1/2

```
/******************************************************************************/
/* File: MPEGAudioParser.h                                                    */
/******************************************************************************/
/* This program file contains the definition of the the MPEGAudioParser class.*/
/* This class can process an Mpeg Audio Stream for the purposes of either     */
/* encryption or decryption.  the class requires access to a fully            */
/* synchronised stream cipher.                                                */
/******************************************************************************/
/* NOTE:                                                                      */
/*      This file contains some Conditional code which is compiled only if    */
/*      the CALC_CIPHER_STATS macro is defined during compile time.  This is  */
/*      true when compiling the 'MPEG1 Cipher' application but not the 'MPEG1 */
/*      Cipher Filter' DirectShow filter.  This code compiles statistics      */
/*      during the cipher process.                                            */
/******************************************************************************/
/* Written By:  Jason But                                                     */
/* Date:        1999-2003                                                     */
/* For:         PhD Thesis in Streaming Video Encryption.                     */
/*              Centre for Telecommunications and Information Engineering.    */
/*              Monash University Australia.                                  */
/******************************************************************************/

/******************************************************************************/
/* Check to see if already included.                                          */
/******************************************************************************/
#ifndef _MPEGAUDIOPARSER_H_
#define _MPEGAUDIOPARSER_H_

/******************************************************************************/
/* Forward definition of StreamCipherBase class.                              */
/******************************************************************************/
class StreamCipherBase;

/******************************************************************************/
/* Number of states in the MPEG Audio Parser State Machine.                   */
/******************************************************************************/
#define AUDIO_NUM_STATES        8

/******************************************************************************/
/* Conditional code.                                                          */
/*                                                                            */
/* The following code is compiled only if the CALC_CIPHER_STATS macro is      */
/* defined during compile time.  This is true when compiling the 'MPEG1       */
/* Cipher' application but not the 'MPEG1 Cipher Filter' DirectShow filter.   */
/* The macros defined in this section define the variables used to maintain   */
/* statistics for the parsing module, otherwise the macros resolve to no code.*/
/******************************************************************************/
#ifdef CALC_CIPHER_STATS

#define DEFINE_VARIABLES()      long    lBytesProcessed, lBytesSelected, lBytesEncrypted

#else

#define DEFINE_VARIABLES()

#endif

/******************************************************************************/
/* class MPEGAudioParser.                                                     */
/*                                                                            */
/* This class will parse an MPEG-1 Audio Stream in blocks, selecting bytes    */
/* from the stream to pass to the cipher.  The cipher is a modified stream    */
/* cipher pointed to by pcCipher - which produces pseudo-random bytes - the   */
/* modifications ensure against creation of false MPEG-1 Audio Headers,       */
/* details of the modifications can be found in the corresponding source code */
```

May 02, 03 9:12 **MPEGAudioParser.h** Page 2/2

```
/* file.  The public member methods of this class are:                        */
/*                                                                            */
/* Constructor          : Creates and initialises the state machine used to   */
/*                        parse the binary stream.  Registers the provided    */
/*                        cipher to be used to generated random values.       */
/* ResetStateMachine()  : Resets the State Machine to the initial state, used */
/*                        during indexed playback to reset the module to ensure*/
/*                        correct parsing of the newly indexed stream.        */
/* ParseBuffer()        : Parses a block of data from the Audio Stream, the   */
/*                        state is remembered at the end of the data block so */
/*                        that processing can continue on the next provided   */
/*                        data block.                                         */
/******************************************************************************/
class MPEGAudioParser
{
    protected:
        int                 iState;
        int                 ppiStateLookup[AUDIO_NUM_STATES][256];
        long                lResyncValue;
        StreamCipherBase    *pcCipher;

        void                InitialiseLookupTable();

    public:
                            MPEGAudioParser();
        void                RegisterCipher(StreamCipherBase *pcStreamCipher) { pcCipher = pcStreamCi
pher; }
        void                ResetStateMachine();
        void                ParseBuffer(byte *pbBuffer, long lBufferSize);

        DEFINE_VARIABLES();
};

/******************************************************************************/
/* Un-define macros that are no longer needed.                                */
/******************************************************************************/
#undef  DEFINE_VARIABLES

#endif

/******************************************************************************/
/* End of File: MPEGAudioParser.h                                             */
/******************************************************************************/
```

Printed by Jason But

May 02, 03 9:12 **MPEGAudioParser.cpp** Page 1/4

```
/****************************************************************************/
/* File: MPEGAudioParser.cpp                                                */
/****************************************************************************/
/* This program file contains the implementation of the the MPEGAudioParser */
/* class.  This class can process an Mpeg Audio Stream for the purposes of  */
/* either encryption or decryption.  the class requires access to a fully   */
/* synchronised stream cipher.                                              */
/****************************************************************************/
/* NOTE:                                                                    */
/*      This file contains some Conditional code which is compiled only if  */
/*      the CALC_CIPHER_STATS macro is defined during compile time.  This is */
/*      true when compiling the 'MPEG1 Cipher' application but not the 'MPEG1 */
/*      Cipher Filter' DirectShow filter.  This code compiles statistics    */
/*      during the cipher process.                                          */
/****************************************************************************/


/****************************************************************************/
/* Written By:  Jason But                                                   */
/* Date:        1999-2003                                                   */
/* For:         PhD Thesis in Streaming Video Encryption.                   */
/*              Centre for Telecommunications and Information Engineering.  */
/*              Monash University Australia.                                */
/****************************************************************************/

/****************************************************************************/
/* Include Standard C Libraries.                                            */
/****************************************************************************/
#include <stdlib.h>

/****************************************************************************/
/* Definition of StreamCipherBase class.                                    */
/****************************************************************************/
#include "StreamCipherBase.h"

/****************************************************************************/
/* Include class header file.                                               */
/****************************************************************************/
#include "MPEGAudioParser.h"

/****************************************************************************/
/* Conditional code.                                                        */
/*                                                                          */
/* The following code is compiled only if the CALC_CIPHER_STATS macro is    */
/* defined during compile time.  This is true when compiling the 'MPEG1     */
/* Cipher' application but not the 'MPEG1 Cipher Filter' DirectShow filter. */
/* The macros defined in this section either reset or update statistic      */
/* counters for the parsing module, otherwise the macros resolve to no code. */
/****************************************************************************/
#ifdef CALC_CIPHER_STATS

#define RESET_VARIABLES()          lBytesProcessed = lBytesSelected = lBytesEncrypted = 0
#define INC_PROCESSED(lAmount)     lBytesProcessed+= lAmount
#define INC_SELECTED()             lBytesSelected++
#define INC_ENCRYPTED()            lBytesEncrypted++

#else

#define RESET_VARIABLES()
#define INC_PROCESSED(lAmount)
#define INC_SELECTED()
#define INC_ENCRYPTED()

#endif

/****************************************************************************/
```

May 02, 03 9:12 **MPEGAudioParser.cpp** Page 2/4

```
/* MPEG Audio Stream Encryption                                             */
/*                                                                          */
/* Partial Selection of Audio Stream for Encryption.                        */
/*                                                                          */
/* An MPEG-1 Audio Stream can be broken into a sequence of successive Audio */
/* Frames, each with a four byte Audio Frame Header.  The header is always  */
/* byte aligned and the first twelve bits are always set.  This allows for  */
/* easy location of the header.  The encryption algorithm outlined in my    */
/* thesis selects the audio payload for encryption.  However we must also   */
/* ensure that we do not create any false Audio Headers nor modify any      */
/* existing headers.  This is done by not encrypting any byte that has the  */
/* value 0xff (a byte aligned value of eight set bits) nor any byte following */
/* this byte.  We also ensure that no byte encrypts to the value 0xff.      */
/* Outlined below is a State Machine which will parse an MPEG-1 Audio Stream */
/* and select bytes for encryption.                                         */
/*                                                                          */
/*   STATE      INPUT       NEXT STATE    ACTION                            */
/* ------------------------------------------------------------------------ */
/*    1-1       1111-1111   1-2           .                                 */
/*              default     1-1           .                                 */
/*                                                                          */
/*    1-2       1111-xxxx   1-3           .                                 */
/*              default     1-1           .                                 */
/*                                                                          */
/*    1-3       default     1-4           .                                 */
/*                                                                          */
/*    1-4       default     2-1           .                                 */
/*                                                                          */
/*    2-1       default     2-2           INPUT = 2nd 8 LSB of Resync Value */
/*                                                                          */
/*    2-2       default     3-1           INPUT = 8 LSB of Resync Value     */
/*                                        Resynchronise Stream Cipher.      */
/*                                                                          */
/*    3-1       1111-1111   3-2           Pass INPUT through cipher.        */
/*              default     3-1           Pass INPUT through cipher.        */
/*                                                                          */
/*    3-2       1111-xxxx   1-3           .                                 */
/*              default     3-1           .                                 */
/* ------------------------------------------------------------------------ */
/*                                                                          */
/* Resynchronisation occurs after obtaining a 16-bit resynchronisation value */
/* in states 2-1 and 2-2.  This value is added to 0xffff0000 to produce a 32 */
/* bit resynchronisation value.  This ensures that the pseudo random stream */
/* generated is correct for decryption during indexed playback.  Details of */
/* the modified stream cipher to not produce any false headers is outlined in */
/* comments for the ParseBuffer() method.                                   */
/****************************************************************************/

/****************************************************************************/
/* Definition of State Names for human readable purposes.                   */
/****************************************************************************/
#define STATE_1_1       0
#define STATE_1_2       1
#define STATE_1_3       2
#define STATE_1_4       3
#define STATE_2_1       4
#define STATE_2_2       5
#define STATE_3_1       6
#define STATE_3_2       7

/****************************************************************************/
/* Constructor.                                                             */
/*                                                                          */
/* This is the constructor for the MPEGAudioParser class.  We initialise the */
/* State Machine Lookup Table before resetting the current state.           */
```

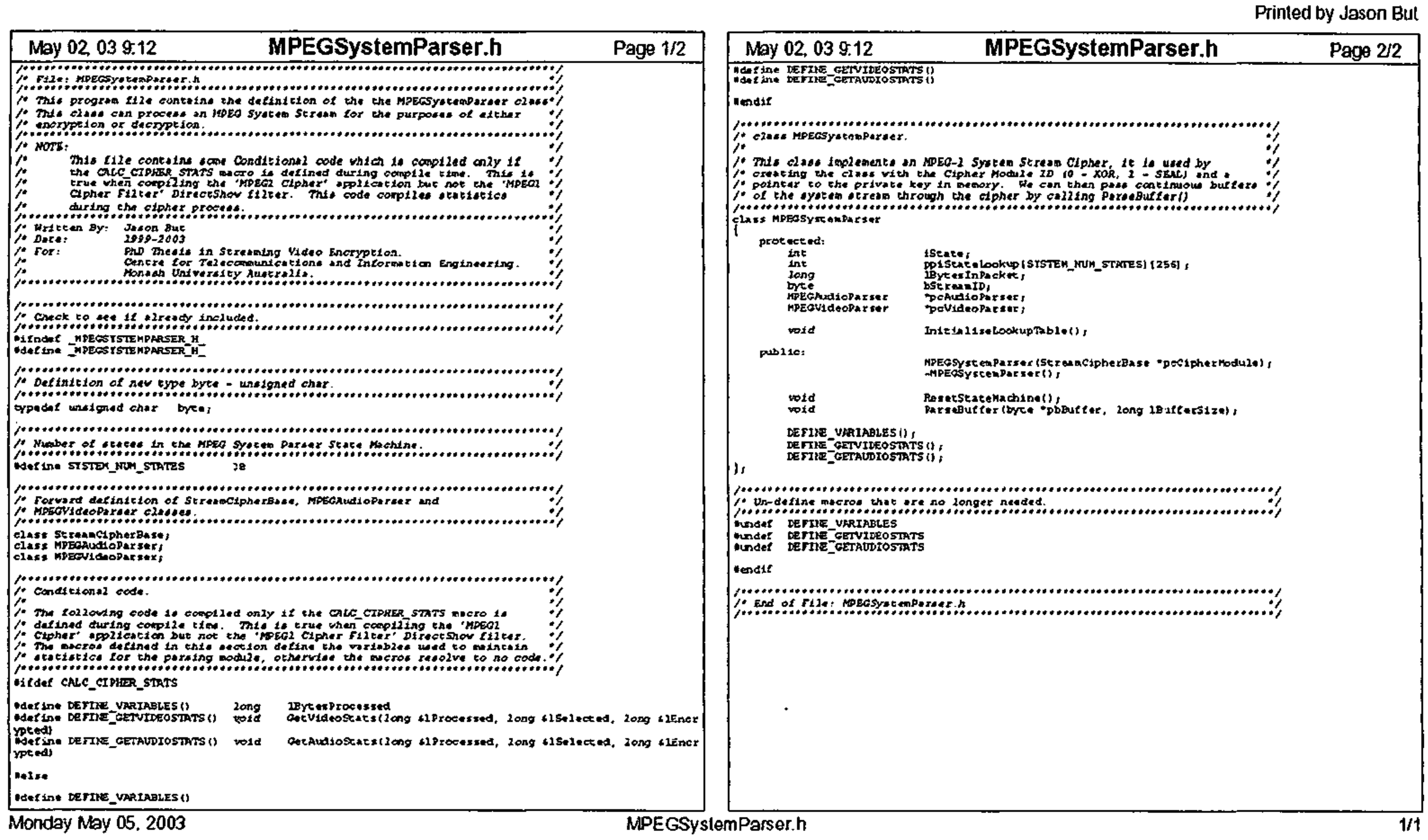
Printed by Jason But

May 02, 03 9:12 **MPEGAudioParser.cpp** Page 3/4

```
/****************************************************************************/
MPEGAudioParser::MPEGAudioParser() : pcCipher(NULL)
{
    InitialiseLookupTable();
    ResetStateMachine();
}

/****************************************************************************/
/* void InitialiseLookupTable()                                             */
/*                                                                          */
/* This method is called to initialise the lookup table to confer with the  */
/* state machine described above.                                           */
/****************************************************************************/
void
MPEGAudioParser::InitialiseLookupTable()
{
    for (int iInput = 0x00; iInput <= 0xff; iInput++)
    {
        ppiStateLookup[STATE_1_1][iInput] = (iInput == 0xff)?STATE_1_2:STATE_1_1;
        ppiStateLookup[STATE_1_2][iInput] = ((iInput & 0xf0) == 0xf0)?STATE_1_3:STATE_1_1;
        ppiStateLookup[STATE_1_3][iInput] = STATE_1_4;
        ppiStateLookup[STATE_1_4][iInput] = STATE_2_1;
        ppiStateLookup[STATE_2_1][iInput] = STATE_2_2;
        ppiStateLookup[STATE_2_2][iInput] = STATE_3_1;
        ppiStateLookup[STATE_3_1][iInput] = (iInput == 0xff)?STATE_3_2:STATE_3_1;
        ppiStateLookup[STATE_3_2][iInput] = ((iInput & 0xf0) == 0xf0)?STATE_1_3:STATE_3_1;
    }
}

/****************************************************************************/
/* void ResetStateMachine()                                                 */
/*                                                                          */
/* This method allows us to reset the state machine to the initial state of */
/* STATE_1_1.                                                               */
/****************************************************************************/
void
MPEGAudioParser::ResetStateMachine()
{
    iState = STATE_1_1;
    RESET_VARIABLES();
}

/****************************************************************************/
/* void ParseBuffer(byte *pbBuffer, long lBufferSize)                       */
/*                                                                          */
/* This method parses a buffer containing a part of an MPEG Audio Stream and */
/* passes the relevant bytes from the buffer to the cipher.  the method takes */
/* a pointer to the buffer and the length of that buffer.                   */
/* The partial selection procedure is outlined in the above, when the state */
/* machine is in State 3-1, the bytes must be encrypted.  The cipher is a   */
/* modified stream cipher, it will not generate a 0xff byte nor will it     */
/* modify an existing 0xff byte as these bytes are used as markers for the  */
/* State Machine.  This is achieved by checking the existing value prior to */
/* performing the XOR, if the byte is 0xff or the generated random byte value */
/* XORed with 0xff, the XOR is not performed.  The first check ensures we do */
/* not remove any existing 0xff marker bytes and the second ensures we do not */
/* create a false 0xff marker byte (bRandom ^ 0xff ^ bRandom = 0xff).       */
/* When the State Machine is in State 2-1, we have located the start of an  */
/* audio frame and are about to complete processing the header, the next two */
/* bytes are used to determine a cipher resynchronisation value, the bytes  */
/* that then follow form part of the payload and must be passed through the */
/* cipher.  For resynchronisation purposes, once this resynchronisation value */
/* is computed, we must reset the stream cipher random number generator to  */
/* ensure we produce the correct sequence of random bytes.  After processing */
/* a byte we must update the current State, this is done using the State    */
```

May 02, 03 9:12 **MPEGAudioParser.cpp** Page 4/4

```
/* Machine Lookup Table.                                                    */
/****************************************************************************/
void
MPEGAudioParser::ParseBuffer(byte *pbBuffer, long lBufferSize)
{
    byte    bRandom, *pbInput, *pbEndBuffer = pbBuffer + lBufferSize;

    if (pcCipher == NULL) return;

    for(pbInput = pbBuffer; pbInput != pbEndBuffer; pbInput++)
    {
        switch (iState)
        {
            case STATE_2_1: lResyncValue = *pbInput << 8; break;
            case STATE_2_2: lResyncValue+= *pbInput;
                            pcCipher->ResynchAudioCipher(0xffff0000 + lResyncValue);
                            break;
            case STATE_3_1: bRandom = pcCipher->GetNextRandomAudioByte();
                            INC_SELECTED();
                            if ((*pbInput != 0xff) && (((*pbInput) ^ bRandom) != 0xff))
                            {
                                *pbInput^= bRandom;
                                INC_ENCRYPTED();
                            }
        }
        iState = ppiStateLookup[iState][*pbInput];
    }
    INC_PROCESSED(lBufferSize);
}

/****************************************************************************/
/* End of File: MPEGAudioParser.cpp                                         */
/****************************************************************************/
```



Printed by Jason But

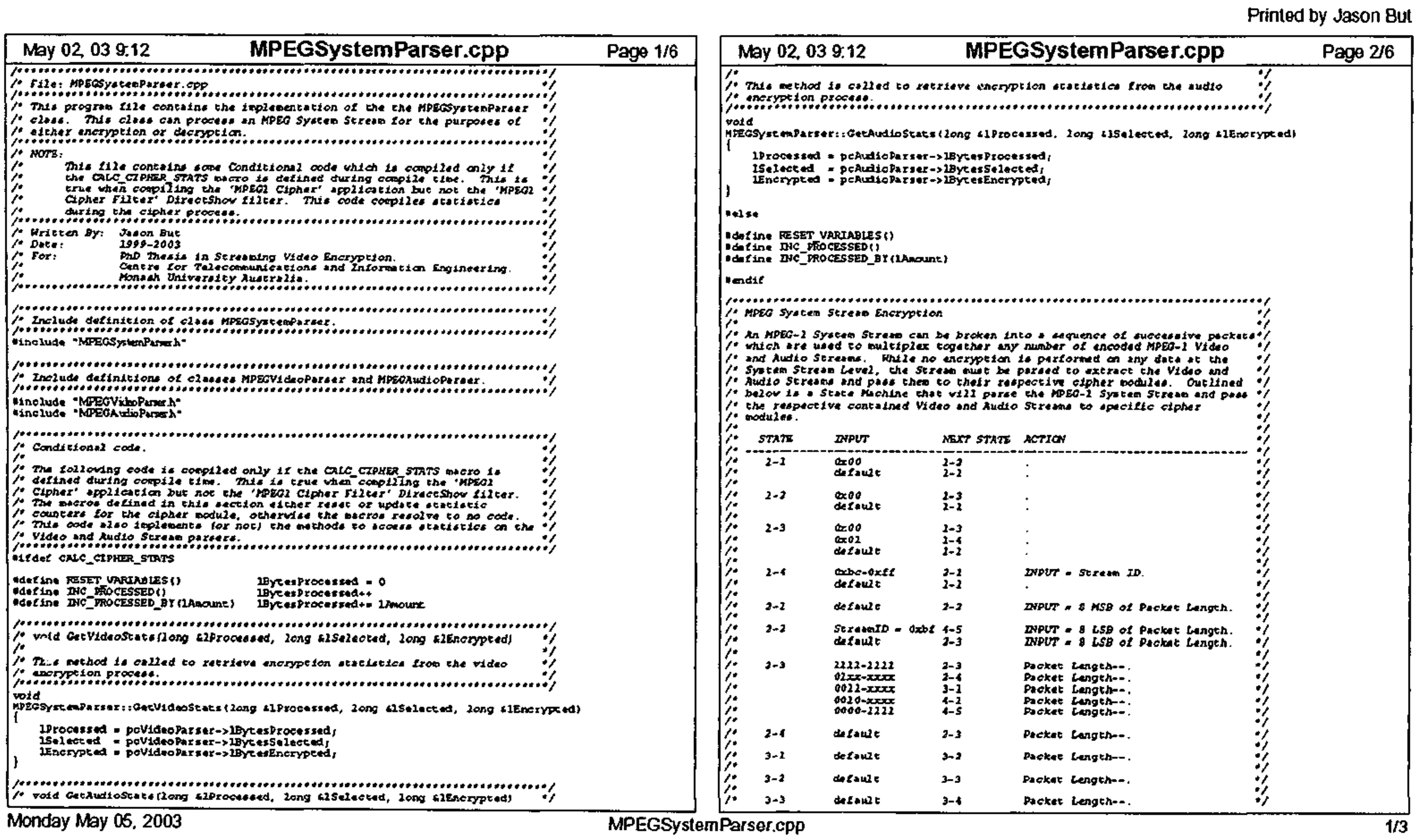
May 02, 03 9:12 **MPEGSystemParser.h** Page 1/2

```
/***************************************************************************/
/* File: MPEGSystemParser.h                                                */
/***************************************************************************/
/* This program file contains the definition of the the MPEGSystemParser class*/
/* This class can process an MPEG System Stream for the purposes of either  */
/* encryption or decryption.                                               */
/***************************************************************************/
/* NOTE:                                                                   */
/*       This file contains some Conditional code which is compiled only if */
/*       the CALC_CIPHER_STATS macro is defined during compile time.  This is */
/*       true when compiling the 'MPEG1 Cipher' application but not the 'MPEG1 */
/*       Cipher Filter' DirectShow filter.  This code compiles statistics  */
/*       during the cipher process.                                        */
/***************************************************************************/
/* Written By:  Jason But                                                  */
/* Date:        1999-2003                                                  */
/* For:         PhD Thesis in Streaming Video Encryption.                  */
/*              Centre for Telecommunications and Information Engineering. */
/*              Monash University Australia.                               */
/***************************************************************************/

/***************************************************************************/
/* Check to see if already included.                                       */
/***************************************************************************/
#ifndef _MPEGSYSTEMPARSER_H_
#define _MPEGSYSTEMPARSER_H_

/***************************************************************************/
/* Definition of new type byte - unsigned char.                            */
/***************************************************************************/
typedef unsigned char   byte;

/***************************************************************************/
/* Number of states in the MPEG System Parser State Machine.               */
/***************************************************************************/
#define SYSTEM_NUM_STATES       38

/***************************************************************************/
/* Forward definition of StreamCipherBase, MPEGAudioParser and             */
/* MPEGVideoParser classes.                                                */
/***************************************************************************/
class StreamCipherBase;
class MPEGAudioParser;
class MPEGVideoParser;

/***************************************************************************/
/* Conditional code.                                                       */
/*                                                                         */
/* The following code is compiled only if the CALC_CIPHER_STATS macro is   */
/* defined during compile time.  This is true when compiling the 'MPEG1    */
/* Cipher' application but not the 'MPEG1 Cipher Filter' DirectShow filter. */
/* The macros defined in this section define the variables used to maintain */
/* statistics for the parsing module, otherwise the macros resolve to no code.*/
/***************************************************************************/
#ifdef CALC_CIPHER_STATS

#define DEFINE_VARIABLES()      long    lBytesProcessed
#define DEFINE_GETVIDEOSTATS()  void    GetVideoStats(long &lProcessed, long &lSelected, long &lEncr
ypted)
#define DEFINE_GETAUDIOSTATS()  void    GetAudioStats(long &lProcessed, long &lSelected, long &lEncr
ypted)

#else

#define DEFINE_VARIABLES()
```

May 02, 03 9:12 **MPEGSystemParser.h** Page 2/2

```
#define DEFINE_GETVIDEOSTATS()
#define DEFINE_GETAUDIOSTATS()

#endif

/***************************************************************************/
/* class MPEGSystemParser.                                                 */
/*                                                                         */
/* This class implements an MPEG-1 System Stream Cipher, it is used by     */
/* creating the class with the Cipher Module ID (0 - XOR, 1 - SEAL) and a  */
/* pointer to the private key in memory.  We can then pass continuous buffers */
/* of the system stream through the cipher by calling ParseBuffer()        */
/***************************************************************************/
class MPEGSystemParser
{
   protected:
        int                 iState;
        int                 ppiStateLookup[SYSTEM_NUM_STATES][256];
        long                lBytesInPacket;
        byte                bStreamID;
        MPEGAudioParser     *pcAudioParser;
        MPEGVideoParser     *pcVideoParser;

        void                InitialiseLookupTable();

   public:
                            MPEGSystemParser(StreamCipherBase *pcCipherModule);
                            ~MPEGSystemParser();

        void                ResetStateMachine();
        void                ParseBuffer(byte *pbBuffer, long lBufferSize);

        DEFINE_VARIABLES();
        DEFINE_GETVIDEOSTATS();
        DEFINE_GETAUDIOSTATS();
};

/***************************************************************************/
/* Un-define macros that are no longer needed.                             */
/***************************************************************************/
#undef  DEFINE_VARIABLES
#undef  DEFINE_GETVIDEOSTATS
#undef  DEFINE_GETAUDIOSTATS

#endif

/***************************************************************************/
/* End of File: MPEGSystemParser.h                                         */
/***************************************************************************/
```

Monday May 05, 2003 MPEGSystemParser.h 1/1



Printed by Jason But

May 02, 03 9:12 **MPEGSystemParser.cpp** Page 1/6

```
/***************************************************************************/
/* File: MPEGSystemParser.cpp                                              */
/***************************************************************************/
/* This program file contains the implementation of the the MPEGSystemParser */
/* class.  This class can process an MPEG System Stream for the purposes of */
/* either encryption or decryption.                                        */
/***************************************************************************/
/* NOTE:                                                                   */
/*       This file contains some Conditional code which is compiled only if */
/*       the CALC_CIPHER_STATS macro is defined during compile time.  This is */
/*       true when compiling the 'MPEG1 Cipher' application but not the 'MPEG1 */
/*       Cipher Filter' DirectShow filter.  This code compiles statistics  */
/*       during the cipher process.                                        */
/***************************************************************************/
/* Written By:  Jason But                                                  */
/* Date:        1999-2003                                                  */
/* For:         PhD Thesis in Streaming Video Encryption.                  */
/*              Centre for Telecommunications and Information Engineering. */
/*              Monash University Australia.                               */
/***************************************************************************/

/***************************************************************************/
/* Include definition of class MPEGSystemParser.                           */
/***************************************************************************/
#include "MPEGSystemParser.h"

/***************************************************************************/
/* Include definitions of classes MPEGVideoParser and MPEGAudioParser.     */
/***************************************************************************/
#include "MPEGVideoParser.h"
#include "MPEGAudioParser.h"

/***************************************************************************/
/* Conditional code.                                                       */
/*                                                                         */
/* The following code is compiled only if the CALC_CIPHER_STATS macro is   */
/* defined during compile time.  This is true when compiling the 'MPEG1    */
/* Cipher' application but not the 'MPEG1 Cipher Filter' DirectShow filter. */
/* The macros defined in this section either reset or update statistic     */
/* counters for the cipher module, otherwise the macros resolve to no code. */
/* This code also implements (or not) the methods to access statistics on the */
/* Video and Audio Stream parsers.                                         */
/***************************************************************************/
#ifdef CALC_CIPHER_STATS

#define RESET_VARIABLES()              lBytesProcessed = 0
#define INC_PROCESSED()                lBytesProcessed++
#define INC_PROCESSED_BY(lAmount)      lBytesProcessed+= lAmount

/***************************************************************************/
/* void GetVideoStats(long &lProcessed, long &lSelected, long &lEncrypted)   */
/*                                                                         */
/* This method is called to retrieve encryption statistics from the video  */
/* encryption process.                                                     */
/***************************************************************************/
void
MPEGSystemParser::GetVideoStats(long &lProcessed, long &lSelected, long &lEncrypted)
{
    lProcessed = pcVideoParser->lBytesProcessed;
    lSelected  = pcVideoParser->lBytesSelected;
    lEncrypted = pcVideoParser->lBytesEncrypted;
}

/***************************************************************************/
/* void GetAudioStats(long &lProcessed, long &lSelected, long &lEncrypted)   */
```

May 02, 03 9:12 **MPEGSystemParser.cpp** Page 2/6

```
/*                                                                         */
/* This method is called to retrieve encryption statistics from the audio  */
/* encryption process.                                                     */
/***************************************************************************/
void
MPEGSystemParser::GetAudioStats(long &lProcessed, long &lSelected, long &lEncrypted)
{
    lProcessed = pcAudioParser->lBytesProcessed;
    lSelected  = pcAudioParser->lBytesSelected;
    lEncrypted = pcAudioParser->lBytesEncrypted;
}

#else

#define RESET_VARIABLES()
#define INC_PROCESSED()
#define INC_PROCESSED_BY(lAmount)

#endif

/***************************************************************************/
/* MPEG System Stream Encryption                                           */
/*                                                                         */
/* An MPEG-1 System Stream can be broken into a sequence of successive packets*/
/* which are used to multiplex together any number of encoded MPEG-1 Video  */
/* and Audio Streams.  While no encryption is performed on any data at the  */
/* System Stream Level, the Stream must be parsed to extract the Video and  */
/* Audio Streams and pass them to their respective cipher modules.  Outlined */
/* below is a State Machine that will parse the MPEG-1 System Stream and pass */
/* the respective contained Video and Audio Streams to specific cipher     */
/* modules.                                                                */
/*                                                                         */
/*   STATE      INPUT           NEXT STATE  ACTION                         */
/* ----------------------------------------------------------------------- */
/*    1-1       0x00            1-2         .                              */
/*              default         1-1         .                              */
/*                                                                         */
/*    1-2       0x00            1-3         .                              */
/*              default         1-1         .                              */
/*                                                                         */
/*    1-3       0x00            1-3         .                              */
/*              0x01            1-4         .                              */
/*              default         1-1         .                              */
/*                                                                         */
/*    1-4       0xbc-0xff       2-1         INPUT = Stream ID.             */
/*              default         1-1         .                              */
/*                                                                         */
/*    2-1       default         2-2         INPUT = 8 MSB of Packet Length. */
/*                                                                         */
/*    2-2       StreamID = 0xbf 4-5         INPUT = 8 LSB of Packet Length. */
/*              default         2-3         INPUT = 8 LSB of Packet Length. */
/*                                                                         */
/*    2-3       1111-1111       2-3         Packet Length--.               */
/*              01xx-xxxx       2-4         Packet Length--.               */
/*              0011-xxxx       3-1         Packet Length--.               */
/*              0010-xxxx       4-1         Packet Length--.               */
/*              0000-1111       4-5         Packet Length--.               */
/*                                                                         */
/*    2-4       default         2-3         Packet Length--.               */
/*                                                                         */
/*    3-1       default         3-2         Packet Length--.               */
/*                                                                         */
/*    3-2       default         3-3         Packet Length--.               */
/*                                                                         */
/*    3-3       default         3-4         Packet Length--.               */
```

Monday May 05, 2003 MPEGSystemParser.cpp 1/3

```
/*                                                                    */
/*    3-4      default        3-5       Packet Length--.              */
/*                                                                    */
/*    3-5      default        4-1       Packet Length--.              */
/*                                                                    */
/*    4-1      default        4-2       Packet Length--.              */
/*                                                                    */
/*    4-2      default        4-3       Packet Length--.              */
/*                                                                    */
/*    4-3      default        4-4       Packet Length--.              */
/*                                                                    */
/*    4-4      default        4-5       Packet Length--.              */
/*                                                                    */
/*    4-5      0xc0 <ID< 0xdf 1-1       Next PacketLength bytes passed to */
/*                                      MPEG Audio Stream Cipher.     */
/*             0xe0 <ID< 0xef 1-1       Next PacketLength bytes passed to */
/*                                      Mpeg Video Stream Cipher.     */
/*             default        1-1       Next PacketLength bytes ignored.  */
/* ------------------------------------------------------------------ */
/*                                                                    */
/* The actual cipher is created and managed by the System Stream cipher as */
/* the single cipher is used by both the Audio and Video Ciphers Parsers.  */
/* this module also allows setting the cipher private key.            */
/**********************************************************************/

/**********************************************************************/
/* Definition of State Names for human readable purposes.             */
/**********************************************************************/
#define STATE_1_1       0
#define STATE_1_2       1
#define STATE_1_3       2
#define STATE_1_4       3
#define STATE_2_1       4
#define STATE_2_2       5
#define STATE_2_3       6
#define STATE_2_4       7
#define STATE_3_1       8
#define STATE_3_2       9
#define STATE_3_3       10
#define STATE_3_4       11
#define STATE_3_5       12
#define STATE_4_1       13
#define STATE_4_2       14
#define STATE_4_3       15
#define STATE_4_4       16
#define STATE_4_5       17

/**********************************************************************/
/* Constructor.                                                       */
/*                                                                    */
/* This is the constructor for the MPEGSystemParser class.  We first create */
/* instances of the  MPEG Audio Stream and Video Stream Parsers.  We then  */
/* register the provided cipher module with both parsing classes.  Finally we */
/* initialise the State Machine Lookup Table before resetting the current  */
/* state.                                                             */
/**********************************************************************/
MPEGSystemParser::MPEGSystemParser(StreamCipherBase *pcCipherModule)
{
    pcAudioParser = new MPEGAudioParser;
    pcVideoParser = new MPEGVideoParser;

    pcAudioParser->RegisterCipher(pcCipherModule);
    pcVideoParser->RegisterCipher(pcCipherModule);

    InitialiseLookupTable();
```

```
    ResetStateMachine();
}

/**********************************************************************/
/* Destructor.                                                        */
/*                                                                    */
/* This is the destructor for the MPEGSystemParser class.  We release any */
/* dynamically allocated memory for our parsing classes.              */
/**********************************************************************/
MPEGSystemParser::~MPEGSytemParser()
{
    delete pcVideoParser;
    delete pcAudioParser;
}

/**********************************************************************/
/* void InitialiseLookupTable()                                       */
/*                                                                    */
/* This method is called to initialise the lookup table to confer with the */
/* state machine described above.                                     */
/**********************************************************************/
void
MPEGSystemParser::InitialiseLookupTable()
{
    for (int iInput = 0x00; iInput <= 0xff; iInput++)
    {
        ppiStateLookup[STATE_1_1][iInput] = (iInput)?STATE_1_1:STATE_1_2;
        ppiStateLookup[STATE_1_2][iInput] = (iInput)?STATE_1_1:STATE_1_3;
        ppiStateLookup[STATE_1_3][iInput] = (iInput)?STATE_1_1:STATE_1_3;
        ppiStateLookup[STATE_1_4][iInput] = (iInput >= 0xbc)?STATE_2_1:STATE_1_1;

        ppiStateLookup[STATE_2_1][iInput] = STATE_2_2;
        ppiStateLookup[STATE_2_2][iInput] = STATE_2_3;

        if ((iInput & 0xc0) == 0x40)
            ppiStateLookup[STATE_2_3][iInput] = STATE_2_4;
        else if ((iInput & 0xf0) == 0x30)
            ppiStateLookup[STATE_2_3][iInput] = STATE_3_1;
        else if ((iInput & 0xf0) == 0x20)
            ppiStateLookup[STATE_2_3][iInput] = STATE_4_1;

        ppiStateLookup[STATE_2_4][iInput] = STATE_2_3;

        ppiStateLookup[STATE_3_1][iInput] = STATE_3_2;
        ppiStateLookup[STATE_3_2][iInput] = STATE_3_3;
        ppiStateLookup[STATE_3_3][iInput] = STATE_3_4;
        ppiStateLookup[STATE_3_4][iInput] = STATE_3_5;
        ppiStateLookup[STATE_3_5][iInput] = STATE_4_1;

        ppiStateLookup[STATE_4_1][iInput] = STATE_4_2;
        ppiStateLookup[STATE_4_2][iInput] = STATE_4_3;
        ppiStateLookup[STATE_4_3][iInput] = STATE_4_4;
        ppiStateLookup[STATE_4_4][iInput] = STATE_4_5;
        ppiStateLookup[STATE_4_5][iInput] = STATE_4_5;
    }
    ppiStateLookup[STATE_1_3][0x01] = STATE_1_4;
    ppiStateLookup[STATE_2_3][0xff] = STATE_2_3;
    ppiStateLookup[STATE_2_3][0x0f] = STATE_4_5;
}

/**********************************************************************/
/* void ResetStateMachine()                                           */
/*                                                                    */
/* This method allows us to reset the state machine to the initial state of */
/* STATE_2_2.                                                         */
```

```
/**********************************************************************/
void
MPEGSystemParser::ResetStateMachine()
{
    iState = STATE_1_1;
    RESET_VARIABLES();
}

/**********************************************************************/
/* void ParseBuffer(byte *pbBuffer, long lBufferSize)                 */
/*                                                                    */
/* This method parses a buffer containing a part of an MPEG System Stream and */
/* passes the relevant bytes from the buffer to either the Audio or Video  */
/* parsing/cipher modules.  The method takes a pointer to the buffer and the */
/* length of that buffer.                                             */
/* The parsing is performed using the State Machine outlined above, when the */
/* state machine is in State 4-5, we have hit the payload and must pass this */
/* payload to the Video or Audio modules.  Since the Packet payload may be */
/* longer (or shorter) that the remaining buffer, we need to be able to break */
/* up the payload into two seperate calls for when the next time ParseBuffer()*/
/* is called.  the class member variable lBytesInPacket (which counts the  */
/* remaining number of bytes in the packet is decremented according to how */
/* many bytes we actually process - we only leave this state when this value */
/* decrements to zero.                                                */
/* While in the remaining states, we step through the State machine as normal */
/* however we also need to calculate the values of lBytesInPacket and of   */
/* bStreamID (the ID code of the multiplexed stream).  In some states, we  */
/* also need to decrement lBytesInPacket as we process bytes within the packet*/
/* Header.                                                            */
/**********************************************************************/
void
MPEGSystemParser::ParseBuffer(byte *pbBuffer, long lBufferSize)
{
    long    lCryptAmount;
    byte    *pbInput = pbBuffer;
    byte    *pbEndBuffer = pbBuffer + lBufferSize;

    while (pbInput != pbEndBuffer)
    {
        if (iState == STATE_4_5)
        {
            lCryptAmount = ((pbEndBuffer - pbInput) < lBytesInPacket)?(pbEndBuffer - pbInput):lBytes
InPacket;
            if ((bStreamID >= 0xc0) && (bStreamID <= 0xdf))
                pcAudioParser->ParseBuffer(pbInput, lCryptAmount);
            else if ((bStreamID >= 0xe0) && (bStreamID <= 0xef))
                pcVideoParser->ParseBuffer(pbInput, lCryptAmount);
            else INC_PROCESSED_BT(lCryptAmount);

            pbInput+= lCryptAmount;
            lBytesInPacket-= lCryptAmount;
            if (lBytesInPacket == 0)
                iState = STATE_1_1;
        } else
        {
            switch (iState)
            {
                case STATE_1_1:
                case STATE_1_2:
                case STATE_1_3:     break;
                case STATE_1_4:     bStreamID = *pbInput; break;
                case STATE_2_1:     lBytesInPacket = *pbInput << 8; break;
                case STATE_2_2:     lBytesInPacket+= *pbInput; break;
                default:            lBytesInPacket--; break;
            }
```

```
                iState = ((iState == STATE_2_2) && (bStreamID == 0xbf))?STATE_4_5:(ppiStateLookup[iState
]{*pbInput]);
                pbInput++;
                INC_PROCESSED();
            }
        }
}

/**********************************************************************/
/* End of File: MPEGSystemParser.cpp                                  */
/**********************************************************************/
```

C.6 Applications

The source code for the applications and complete modules on the CD will not be described in detail here, the bulk of this code contains necessary calls to create either a Windows based GUI application or a module that correctly converses with the Microsoft DirectShow environment. What follows is a brief description of each application and its functionality.

C.6.1 MPEG-1 File Encryption

The most basic application included on the CD is one that can be used to both encrypt and decrypt MPEG-1 files on disk. This application – *"MPEG1 Cipher.exe"* – does not allow real-time decryption and playback but rather is used for two tasks:

- Perform a file encryption so that execution time and therefore performance can be measured.
- Encrypt an MPEG-1 file for later installation onto a Streaming Video Server.

This application is a Windows application whereby a source and destination file can be selected via standard Windows "File Open" and "File Save" Dialog Boxes. Other options include a selection of which cipher to use – no cipher (simple file copy), the prototype cipher and the secure cipher – and the key to use when processing the file. During testing this application was used for a number of purposes, these included ensuring the encryption process was reversible, measuring the execution speed of the cipher and creating encrypted MPEG-1 Streams to install onto Streaming Servers for functionality testing.

C.6.2 DirectShow MPEG-1 Cipher Filter

The core utility included on the CD is the Microsoft DirectShow Filter – *"MPEG1 Cipher Filter.dll"* – which is a streaming media compatible module that can be used with all DirectShow compatible applications, including the DirectShow Filter Graph Editor and MediaPlayer. The filter provides two input pins, accepting one MPEG-1 Video and one MPEG-1 Audio Stream, and two output pins, providing the modified streams after passing through the cipher. The filter also provides a properties box where the cipher type – prototype or secure – and the private key can be selected. As well as manual selection of the cipher and key, a programming interface is provided where the cipher type and key can be selected for the filter. The filter has been designed to function where data is presented on only one of its two input pins, allowing processing of a video only stream.

The Filter can be used in one of two ways, it was used when testing the functionality of real-time decryption and playback through the DirectShow Filter Graph Editor, an encrypted file was passed through the filter prior to decoding and display, allowing the functionality to be visually verified as well as the performance to be measured against the playback of the plaintext file. The filter is also utilised in numerous playback applications that stream video from a streaming server, testing real-time decryption and playback in a variety of different playback modes.

C.6.3 DirectShow Stream Playback Application

This sample application can be used to stream a video from either the Microsoft NetShow Theatre Streaming Video Server or an SGI MediaBase 3.1 Streaming Video Server using DirectShow. Both of these streaming servers come with DirectShow enabled source filters, allowing a DirectShow Filter Graph to be constructed to stream, decode and playback video installed on the servers. This application – *"StreamCipher.exe"* – allows the user to select which type of server to use, the name of the video installed on the server, selection of the prototype or secure cipher filter in decrypting the streaming video (or no cipher filter for plaintext playback) as well as the key to use to decrypt the video. Once the video is streaming, control is provided to allow the user to seek to any point in the video during playback, as well as providing pause and high-speed playback functionality. In essence, this application is a special version of the Microsoft MediaPlayer application that inserts the DirectShow Cipher Filter described in the previous section into the Filter Graph, thereby allowing the real-time decryption and playback of streaming video.

It would be possible to use the Filter Graph Editor to achieve real-time decryption and playback, but this option would preclude the options of testing functionality during seek and high-speed playback modes. This application allows verification that the cipher design correctly functions during the various digital playback modes not supported by the Filter Graph Editor. This application was used to verify the complete functionality of the designed cipher, both prototype and secure, in all playback modes when streaming from either of the two supported Streaming Video Servers. Note that the DirectShow Source Filter provided for the MediaBase server does not provide support for the high-speed playback modes – fast forward and fast rewind.

C.6.4 MediaBase Playback Application

This application – *"SGIStreamCipher.exe"* – also allows a user to stream video from the SGI MediaBase 3.1 Streaming Video Server. While the MediaBase source filter only provides standard playback and seek functionality – it does not support high-speed playback modes even though the streaming server does, it is possible to access these playback modes through the MediaBase software development SDK. This library allows a programmer to request and retrieve a stream from the streaming server, then decoding and displaying this stream within their own application. Since this requires development of an MPEG-1 decoding module, it would be too difficult to develop a complete streaming video client playback application that incorporates the MPEG-1 cipher filter described in this thesis.

A different approach was taken instead, a Windows Application was developed that would allow the user to specify a MediaBase server and an encrypted MPEG-1 file installed on that server, the user would also specify which type of cipher to use and the key. The user would then be presented with a video control panel, allowing general playback, pause, full seek functionality and playback in high-speed modes. However, instead of decoding and displaying the resultant stream, the application would decrypt the streamed video and save it to disk in a format that could later be read and decoded for display. If all the saved files are playable using a standard MPEG-1 video player, then we can show that the decryption algorithm correctly decrypted the encrypted video stream and resulted in a viable MPEG-1 Stream for real-time decoding and playback, only that the final step was not implemented due to the complexity of the development effort. This application was used to further verify the compatibility of the cipher with various existing streaming video platforms. While it is difficult to perform any performance tests using this application, it can be used to prove that not only can the MediaBase server successfully stream an encrypted MPEG-1 bitstream, but that it can be successfully decrypted ready for decoding and playback in all supported playback modes. Verification of real-time decryption and playback of indexed normal speed playback can be achieved with the *"StreamCipher.exe"* application. Some final notes regarding the *"SGIStreamCipher.exe"* application:

- Positional updates of the progress bar to indicate the current playback position of the video being streamed are only approximate due to this value usually being obtained from the MPEG-1 Decoder. The displayed playback position is especially incorrect during high-speed playback modes.

- Streaming cannot commence until the video has been opened on the server. Since this takes place in a second thread, there is a synchronisation issue that must be addressed so that the primary thread can commence streaming. As this is a sample application only, this has been achieved in a cumbersome way, the primary thread will open a Windows Message Box prompting the user to hit OK to set up the slider bar and commence streaming while the secondary thread will open a Windows Message Box informing the user that the stream has been opened on the server and that streaming can commence. In order for the application to function correctly, the user must close the Message Box from the secondary thread prior to closing that on the primary thread.

C.6.5 Other Applications

Also included is a small application, "Byte Count.exe", this application will read any file – expected to be an MPEG-1 bitstream – and count the distribution of different byte values within the bitstream. The displayed results include a raw count and subsequent proportions of the 0x00 and 0x01 bytes within the file. Also displayed is the mean count of the remaining byte values and the standard deviation. This application is used to show the high proportion of 0x00 and 0x01 bytes within an MPEG-1 bitstream as well as the relatively even distribution of remaining byte values.

Appendix D

Experimental Results

In this Appendix I will provide a complete description of the experiments performed with respect to the work presented in this Thesis. This information is expanded from that presented in Chapter 4 and Chapter 5 and contains the following information:

- **Input File Information** – General information about all the test MPEG-1 bitstreams used for experimental and testing purposes.

- **Experimental Procedure** – Description of the reason for each individual test or experiment and an outline of the steps taken such that the experiment can be reproduced.

- **Other Input Conditions** – Other inputs required to execute the test, in particular the secret key used in the cipher.

- **Expected and Actual Output** – The expected output and results of the performed tests.

D.1 Input Files

To ensure that the proposed MPEG-1 Ciphers function correctly, it is important to subject the ciphers to a wide range of MPEG-1 bitstreams. While it is unfeasible to perform the tests on all conceivable MPEG-1 bitstreams, it is possible to choose a subset of representative bitstreams for testing purposes. Ideally, any MPEG-1 bitstream would be suitable since the MPEG-1 Standard defines both the bitstream and the decoder, any bitstream that complies with the bitstream definition and can be successfully decoded by an MPEG-1 decoder should be as good as any other. However, it is not always possible to be certain of this.

For testing of the cipher designed within this Thesis, six MPEG-1 compatible bitstreams have been selected, three of these bitstreams have been selected due to their widespread use as test streams in other work involving MPEG(Qiao and Nahrstedt, 1997; Shi and Bhargava, 1998a; Tang, 1996; Anderson, 1990; Sikora, 1997). The other three streams have been selected for their combination of video and audio, as well as the fact that they have been encoded using different MPEG-1 encoders. By widening the range of MPEG-1 encoders that have been used, we can begin to simulate a real-world scenario where the bitstreams to be protected can come from numerous sources.

The full details of the selected bitstreams are outlined in Table D-1. The reasons for choosing these particular bitstreams are:

- **tennis.mpg** – This is a standard video sequence in widespread use for research involving MPEG video. Unfortunately, this bitstream is an MPEG-1 Video Stream only and does not contain any audio information. Due to its widespread use as a test bitstream, it can be assumed to fully comply with the MPEG-1 standard. Also, its high bitrate ensures a clear picture after decoding, any errors in the bitstream are likely to be immediately obvious and not lost amid any noise in the picture.

- **flowg.mpg** – This is also a standard video sequence chosen for the same reasons as tennis.mpg.

- **us.mpg** – This video sequence is an MPEG-1 Video Stream of a 24 second clip from the movie "Under Seige". While not used as often as tennis.mpg and flowg.mpg by researchers, this clip was chosen because it best demonstrates the type of content the cipher will be seeking to protect – footage requiring copyright protection. Like the previous two clips, this file also does not contain any audio information.

Filename	Stream Type	Size (bytes)	Duration	Source
tennis.mpg	MPEG-1 Video Stream	1,246,001	0:04	Standard MPEG-1 test sequence, obtained from http://peipa.essex.ac.uk/ipa/src/formats/mpeg/stanford
flowg.mpg	MPEG-1 Video Stream	2,819,836	0:04	Standard MPEG-1 test sequence, obtained from http://peipa.essex.ac.uk/ipa/src/formats/mpeg/stanford
us.mpg	MPEG-1 Video Stream	2,078,802	0:24	Standard MPEG-1 test sequence, obtained from http://peipa.essex.ac.uk/ipa/src/formats/mpeg/stanford
Chicken.mpg	MPEG-1 System Stream	33,663,140	2:40	Produced and encoded by Microsoft Engineers
Monash Nursing.mpg	MPEG-1 System Stream	91,539,915	4:28	Encoded at Monash Uni. using the Siemens Eikona MPEG-1 Encoder
Diablo2_5.mpg	MPEG-1 System Stream	169,236,004	9:01	Encoded by at Monash Uni. using the Optibase MPEG Fusion MPEG-1 Encoder

Table D-1 MPEG-1 Test File Details

- **Chicken.mpg** – This MPEG-1 System Stream was animated and encoded by Microsoft software engineers and can be found on the CD containing the initial release of the Microsoft NetShow Theatre Streaming Server software. This bitstream was chosen since it contained both Video and Audio streams, and was produced by a third party (I had no input into the generation of the bitstream). Also, while short, this animated feature is the sort of content that the copyright owner may want to protect.

- **Monash Nursing.mpg** – Monash Nursing is an advertisement made for the Nursing School of Monash University. The MPEG-1 bitstream was encoded using the Siemens Eikona MPEG-1 Encoder (developed collaboratively by Siemens and Monash University). The original source was content was high quality Beta? And the overall bitstream was encoded at approximately 2.7Mb/s. This bit rate produces an extremely high-quality video as output from an MPEG-1 decoder. This sequence was chosen for two reasons, one to test the cipher with a different commercial MPEG-1 encoder, and two, to test the performance of the cipher on what can be considered a high bit for an MPEG-1 stream (MPEG-1 being primarily designed for bit rates of 1.2Mb/s to 1.5Mb/s

- **Diablo2_5.mpg** – This video sequence was initially encoded by Monash University while working on a collaborative project with SGI and Cinemedia on developing a Video-on-Demand trial. The initial video was provided by Cinemedia and a short sequence was encoded at a variety of bit rates for testing purposes. This sample was encoded at 2.5Mb/s using the Optibase MPEG Fusion hardware based MPEG-1 Encoder. This sample was chosen due to the intermixing of video and audio, the high bit rate, again a different encoder being tested, and finally this material is an excerpt from a real movie – therefore providing real content for testing the cipher.

D.2 Proportion of Stream Selected for Encryption

The MPEG-1 File encryption program produces a series of statistics, part of which includes the number of bytes that have been selected for encryption. These results are tabulated below – see Table D-2 – for each of the test files used. The results were obtained by selecting each of the test files in turn and passing them through the encryption program. The actual cipher type chosen – prototype or SEAL – is irrelevant as the numbers of bytes selected for encryption remain constant. Once the encryption of the file is complete, a window appears containing some statistics, the statistics of interest are – ***Total Bytes, System Stream Bytes, Video Stream Bytes, Video Stream Selected, Audio Stream Bytes*** and ***Audio Stream Selected.*** The experiment is repeatable, choosing different ciphers and/or keys, the results are consistent. Some salient points include:

- The entire MPEG-1 System Stream information is left as plaintext, however, as can be seen from the results, this information makes up a small portion (less than 4%) of the overall stream.

- A high proportion of the Video Stream is selected for encryption. This shows that although there are many layers of encoding information within the MPEG-1 Video Stream headers, the majority (> 99%) of the bitstream consists of Macroblocks and actual video data. This shows that there is no great benefit with respect to minimising the proportion of bytes encrypted using this technique, rather the benefits arise in the encrypted stream being compatible with existing video streaming products.

- Similarly, a high proportion of the Audio Stream is selected for encryption. This is to be expected as the Audio Frame Headers are small (four bytes) and the remaining data is entirely encrypted.

Statistic	tennis	flowg	us	Chicken	Monash Nursing	Diablo2_5
Total Bytes	1,246,001	2,819,836	2,078,802	33,663,140	91,539,915	169,236,004
System Stream Bytes	0	0	0	322,717 (0.96%)	1,289,380 (1.41%)	5,387,976 (3.18%)
Video Stream Bytes	1,246,001	2,819,836	2,078,802	30,779,588	83,812,500	150,851,588
Audio Stream Bytes	0	0	0	2,560,835	6,438,035	12,996,440
Video Stream Selected	1,239,953 (99.5%)	2,813,788 (99.8%)	2,049,416 (98.6%)	30,542,212 (99.2%)	83,500,150 (99.6%)	150,153,682 (99.5%)
Audio Stream Selected	0	0	0	2,528,722 (98.7%)	6,385,721 (99.2%)	12,888,252 (99.2%)
Total Selected	1,239,953 (99.5%)	2,813,788 (99.8%)	2,049,416 (98.6%)	33,070,934 (98.2%)	89,885,871 (98.2%)	163,041,934 (96.3%)

Table D-2 Proportions of Bitstreams Selected for Encryption

- The proportion of the overall bitstream selected for encryption was high (> 96%). This leads to the conclusion that the cipher must be able to operate at bit-rates approaching the average stream bit-rate, while at the same time accommodating the CPU demands of the MPEG-1 decoder to allow for real-time streaming, decryption and playback.

D.3 Prototype Cipher

In this section I will discuss the tests performed to verify the viability and functionality of the prototype cipher designed in Chapter 4. These tests will be performed as a series of four steps, the first of which is to verify the repeatability of the cipher, that is, will the cipher produce the same encrypted bitstream every time given the same input bitstream and key. The second series of tests are set up to prove the reversibility of the cipher, that is, given an encrypted bitstream and the correct key, is the original bitstream reproduced. This test also implies showing that an incorrect bitstream is produced if the key used for decryption is incorrect. The third series of tests are to check the performance of the cipher, to determine the approximate CPU load required by the cipher as well as real-time decryption and playback tests to verify that an encrypted stream can be decrypted and decoded in real-time. The final series of tests are to confirm the primary goal of this new cipher, this involves installation on a range of Streaming Video Servers of an encrypted MPEG-1 file and then streaming and playing back the file in a variety of modes. The experimental procedures and results are outlined in the following sub-sections.

D.3.1 Testing Repeatability

The first set of tests involve verifying the repeatability of the cipher – or, will the cipher produce the same encrypted bitstream every time given the same input bitstream and key. The procedure for this test is outlined below, the same steps are repeated for each test file. The results are tabulated in Table D-3.

- The MPEG-1 test file is encrypted using the encryption program on the CD with the following settings.
    - The test file is selected for input.
    - The chosen cipher is the prototype cipher with the key – `0xff`
    - For the three test files *"tennis.mpg"*, *"flowg.mpg"* and *"us.mpg"*, select Video Stream only as these files are not valid MPEG-1 System Streams.
- The encryption is performed three times, each time with a different output filename being selected.
- Each of the three output files are compared (using the Windows file comparison program fc) with the original file to ensure that the files differ – the original stream has been modified.
- Each of the three output files are compared against each other to ensure that the files are exactly the same – the original stream is modified consistently.

Statistic	tennis	flowg	us	Chicken	Monash Nursing	Diablo2_5
Video Stream Selected	1,239,953	2,813,788	2,049,416	30,542,212	83,500,150	150,153,682
Video Stream Encrypted	1,180,554 (95.2%)	2,678,449 (95.2%)	1,956,573 (95.5%)	25,794,268 (84.5%)	78,447,200 (93.9%)	141,834,369 (94.5%)
Audio Stream Selected	0	0	0	2,528,722	6,385,721	12,888,252
Audio Stream Encrypted	0	0	0	2,461,338 (97.3%)	6,193,239 (97.0%)	12,731,394 (98.8%)
Three Output Files Equal	☑	☑	☑	☑	☑	☑

Table D-3 Encryption Statistics and Repeatability of Encryption Process – Prototype Cipher

As can be seen from the tabulated results, all tests in this case were successful. The three resultant files from each input test file were all equal, this indicates that the process of applying the key to the input bitstream is predictable and able to produce consistent results, allowing the safe assumption that it is possible to consistently reverse the results to obtain the original bitstream. Looking at the proportion of the bytes actually encrypted against the number of bytes selected for encryption. The difference in these two numbers indicates the number of bytes in the original bitstream that were selected for encryption but were left unmodified. This occurs in one of two scenarios:

- A byte from the Video Stream selected for encryption was either 0x00, 0x01, 0xfe or 0xff.

- A byte from the Audio Stream selected for encryption was either 0xff or 0x00.

While the presence of unencrypted bytes is to be expected, the numbers displayed here are higher than would be expected if the bitstream had a purely random distribution of bytes (the percentage not encrypted should be $^4/_{256}$ or approx. 1.56%). When considering these numbers it is important to look at two different aspects, the design of the cipher and how the program calculates the presented statistics, and the original input bitstream itself.

In the first case, the calculation of the bytes selected for encryption and actually encrypted include all bytes processed while in Stages 3-1, 3-2 and 3-3 of the Video Cipher State Machine, when encountering the MPEG-1 header code (0x00-0x00-0x01) to exit back to State 1-4, these three bytes are determined as selected for encryption and subsequently not encrypted by the software as written. This means that for every slice in the MPEG-1 Video Stream, the number of bytes determined as selected for encryption is three higher than what it should be. Similarly for the Audio Stream, although to a lesser extent, the number of bytes determined as being selected for encryption is too high by one for each Audio Frame Header in the stream. Unfortunately, it is difficult to determine the number of Slices or Audio Frames in a bitstream without further processing, however it is obvious that even taking this into account the proportion of bytes actually encrypted to bytes selected for encryption is too low.

Statistic	tennis	flowg	us	Chicken	Monash Nursing	Diablo2_5
Total Bytes	1,246,001	2,819,836	2,078,802	33,663,140	91,539,915	169,236,004
Number Bytes – 0x00	26,767 (2.15%)	63,276 (2.24%)	51,400 (2.47%)	4,207,299 (12.50%)	3,412,828 (3.73%)	5,288,316 (3.12%)
Number Bytes – 0x01	17,719 (1.42%)	46,043 (1.63%)	33,274 (1.60%)	489,028 (1.45%)	1,374,212 (1.50%)	2,160,343 (1.28%)
Expected Number for Purely Random Stream	4,867 (0.39%)	11,015 (0.39%)	8,120 (0.39%)	131,497 (0.39%)	357,578 (0.39%)	661,078 (0.39%)
Remaining Bytes – μ	4,730.4 (0.38%)	10,671.3 (0.38%)	7,850.9 (0.38%)	114,042.0 (0.34%)	341,546.7 (0.37%)	636,958.1 (0.38%)
Remaining Bytes – σ	1,680.9 (0.135%)	5,304.0 (0.188%)	2,615.3 (0.126%)	42,733.0 (0.127%)	138,249.1 (0.151%)	339,558.7 (0.201%)

Table D-4 Byte Count Distribution in Input Streams

The second case involves having a look at the input bitstreams before encryption. Passing these files through a simple filter which calculates the number of 0x00 and 0x01 bytes in these streams is a simple matter, the results are shown in Table D-4. These results show that the frequency of these bytes is higher than expected for a purely random bitstream, since these bytes are not encrypted in a Video Stream, they indicate an expected discrepancy in the proportion of bytes actually encrypted as opposed to those selected for encryption. Note that the test bitstreams which include System Stream data contain a higher proportion of 0x00 bytes on average than a Video only stream.

Looking at the remaining byte values, the mean number of bytes is close to the expected value and the standard deviation shows that while there are occasional peaks and troughs in the byte distribution count, the majority of the counts are much smaller than the count for 0x00 and 0x01 bytes. The higher instances of certain bytes (apart from 0x00 and 0x01) are independent of the MPEG bitstream format, occurring at different byte values for each test stream. Taken with the previous issue of how these statistics are calculated, they explain the difference in observed values from what would normally be expected.

D.3.2 Testing Reversibility

The reversibility of the encryption procedure is one of the most important aspects of any cipher. The cipher is of no practical use unless the encryption process can be completely reversed through the application of the selected secret key. The purposes of the tests outlined here are to prove that if the same key used during encryption is applied to the encrypted bitstream, then the original MPEG-1 bitstream can be re-obtained. Similarly, we wish to verify that if an incorrect key is applied to the encrypted bitstream, then the original bitstream is not obtained, but rather a different, invalid MPEG-1 bitstream is obtained instead. If this series of tests succeed, we can then be confident of proceeding to develop a real-time decryption module that allows playback of the decrypted bitstream simultaneously with the decryption procedure.

While there are 256 possible keys that can be applied using the prototype cipher, it is not feasible to perform a complete test using all possible keys – therefore, a subset of four keys will be used for testing purposes, these four keys are:

- **0xff** – This is the most obvious key to use, forcing each bit in a byte being encrypted to change its value.
- **0x00** – Looking at the rules of the cipher, this key will result in no bytes in the MPEG-1 bitstream being modified due to encryption, meaning that the encrypted bitstream is identical to the original bitstream. Indeed, using this key on an MPEG-1 file and comparing the input and output files verifies that this key has no effect in modifying the bitstream. It is however useful to use this key for testing purposes as it can be used to prove that a double application of this key will still return the original bitstream. Also, using this key shows that it cannot decrypt a sequence encrypted with a different key.
- **0x4a** – This key corresponds to the ASCII code for the character 'J'. This key was primarily selected as it forms the initial of my first name. For the final two test keys, a value had to be chosen and this approach allows selection of one of the remaining 254 possible keys.
- **0x42** – This key corresponds to the ASCII code for the character 'B', selected as it forms the initial of my surname.

In order to test the reversibility of the cipher given the correct key, each test MPEG-1 file was encrypted using each of the four test keys, resulting in 24 encrypted files (6 of these files were equal to the test files, the result of encryption with the key 0x00). These 24 encrypted files were then decrypted using each of the four test keys, resulting in 96 different output files. Each of these resultant files was then compared with their corresponding original bitstreams. The expected results being:

- For the case where the decryption key was equal to the encryption key, the final output file would be equal to the original bitstream.
- For any other case – the two keys being different – the final output file would differ from the original bitstream.

		Decryption Key			
Filename	Encryption Key	0xff	0x00	0x4a	0x42
tennis.mpg	0xff	☑	☒	☒	☒
	0x00	☒	☑	☒	☒
	0x4a	☒	☒	☑	☒
	0x42	☒	☒	☒	☑
flowg.mpg	0xff	☑	☒	☒	☒
	0x00	☒	☑	☒	☒
	0x4a	☒	☒	☑	☒
	0x42	☒	☒	☒	☑
us.mpg	0xff	☑	☒	☒	☒
	0x00	☒	☑	☒	☒
	0x4a	☒	☒	☑	☒
	0x42	☒	☒	☒	☑
Chicken.mpg	0xff	☑	☒	☒	☒
	0x00	☒	☑	☒	☒
	0x4a	☒	☒	☑	☒
	0x42	☒	☒	☒	☑
Monash Nursing.mpg	0xff	☑	☒	☒	☒
	0x00	☒	☑	☒	☒
	0x4a	☒	☒	☑	☒
	0x42	☒	☒	☒	☑
Diablo2_5.mpg	0xff	☑	☒	☒	☒
	0x00	☒	☑	☒	☒
	0x4a	☒	☒	☑	☒
	0x42	☒	☒	☒	☑

Table D-5 Comparison of Decrypted File with Original File given (En/De)cryption Key Pair

In each case, all 96 output files were passed through an MPEG-1 decoder to determine visually whether the resultant bitstream could be decoded and if any of the output was recognisable.

The results from this test are shown in Table D-5. These results confirm the functionality of the cipher in reversing the encryption process, the original bitstream can only be retrieved if the correct key is used for decryption. Visual confirmation indicated that these files were able to be decoded and played back normally. The same approach verified that the decoder could not playback or display any of the resultant output files that were decrypted using an incorrect key.

D.3.3 Performance Testing

While it is well and good to verify the functionality of the MPEG-1 cipher, it is also important to confirm it's viability for real-time decryption and decoding. In this subsection, experiments are described that measure the CPU load required to perform the decryption of an encrypted file as well as the load required to decode and display the original MPEG-1 bitstream on a test computer. A final series of tests will measure CPU loads while decryption and playback occur simultaneously on the test machine. The purpose of these tests is to verify that the CPU requirements of the cipher as designed are minimal and that the vast number of modern machines can easily perform the decryption of the bitstream as well as concurrently decoding and playing back the decrypted stream for a user.

Feature	Test Platform 1	Test Platform 2
CPU Type	Pentium II (233 Mhz)	Pentium 4 (1.6 GHz)
RAM	384 MB	256 MB
Operating System	Windows 2000	Windows 2000
Free Hard Disk Space	1.2 GB	7.1 GB
NIC Type	Intel 82558 Integrated Ethernet	3Com Etherlink 10/100 PCI - 3C905C-TX
Network Connection	Switched 10BaseT	Switched 10BaseT
Graphics Card	Cirrus Logic Laguna 5465 (4MB)	NVidia GeForce2 MX-100/200 (32MB)
Sound Hardware	Sound Blaster 16	Avance AV97 Audio (Onboard Sound)

Table D-6 Test Platform Specifications

Two different computers were used as test machines to measure the performance of the cipher, the details of these machines are outlined in Table D-6. The first of these machines is an older Pentium II computer – the purpose of using this type of machine is to verify that the cipher is capable of running on older hardware and does not require the use of the latest computers to perform well. The second test machine is a more modern computer, a Pentium 4, representing the type of computing hardware most users are likely to have on their desktop, as well as the hardware likely to be used in any new "black-box" device being built to support Video-on-Demand. Of course, the results obtained throughout the entire series of experiments are entirely dependant on the platform being used for testing. As such, these results should only be used as a guide to indicate probable performance, due the abundance of different parameters available on a PC test system and the fact that only Windows boxes were used for testing, the best that can be provided is a guide to how effective the cipher is performance-wise.

The first test measures the time required to encrypt (or decrypt – the procedure is the same) an MPEG-1 file on disk. Since the time required to execute this function is also dependant on time spent reading the source file from disk and writing the resultant file to disk, this must be taken into account. Also of issue is the fact that the encryption application is executing on a multi-tasking operating system where we cannot ensure that 100% of CPU resources are allocated to the task at hand.

The encryption application offers an option to perform a file copy – this procedure uses the exact same code as for encryption to load each byte of the test file and then write the modified data to disk, except that in this case the data is not modified. The purpose of this feature is to measure the time taken for the non-encryption code within the application to execute. This allows the determination of resources required to perform the data load and write procedures, the idea being that this value can be subtracted from the calculation of resources to perform data load, encryption and write functions to obtain an approximation of the resources required for data encryption only.

For test purposes, time required for a procedure to execute is hand timed using the clock on the computer, the time is manually noted when the procedure is started and again when it is ended, this allows precision of time required for execution to the order of 1 second. While this may not appear overly accurate, the difficulty in determining the actual CPU resources granted by the multi-tasking OS means that all values calculated are approximates only. In order to determine the CPU resources, the Windows Performance Tool (Located in Control Panel|Administration Tools) is used to display the CPU load as a percentage. The description of Processor Time in this application is:

> *"Processor Time is the percentage of time that the processor is executing a non-Idle thread. This counter was designed as a primary indicator of processor activity. It is calculated by measuring the time that the processor spends executing the thread of the Idle process in each sample interval, and subtracting that value from 100%. (Each processor has an Idle thread which consumes cycles when no other threads are ready to run). It can be viewed as the percentage of the sample interval spent doing useful work. This counter displays the average percentage of busy time observed during the sample interval. It is calculated by monitoring the time the service was inactive, and then subtracting that value from 100%."*

By restarting the tool at the beginning of the test and pausing at the completion, we can easily obtain an average CPU load from one of the text information areas on the screen. If our tests are performed on a system performing no other primary tasks, hopefully the majority of this reported figure can be attributed to the MPEG encryption application. While some cycles will be used to perform other Operating System tasks, by purposely not executing other applications, we can hope to minimise this effect. Unfortunately it is not possible to directly measure this value, as such the CPU load reported is only approximate. The procedure outlined below allows us to calculate an approximate determination of CPU resources required for encryption, remembering that the measured values have a reasonable margin for error.

- Perform the file copy procedure on each test file three times – noting the time (in seconds) and average CPU load required to perform the three procedures.
- From these figures we can determine the a figure for the time the file copy could run had 100% CPU resources been possible to allocate to the task – time spent idling waiting for the disk to seek is not counted in CPU time for applications, therefore this time is that spent processing functions that read and write data blocks to files before and after calling the disk driver. The figure of processing time at 100% load is obtained by multiplying the measured time by the measured average CPU load during execution.
- Since the previous step is performed three times for each test file, we can average these three results – in an attempt to minimise measurement errors – and obtain an average value for processing time required to copy a file.
- The three previous steps are repeated while actually encrypting the file – for test purposes, the selected encryption key is 0xff. Again for each test file we can determine an average value for the processing time required to read, encrypt and write the modified file back to disk.
- Subtracting the result from step 3 from the value obtained in step 4 will provide an average value for the processing time required to perform the encryption only on the test bitstream.

Statistic	Chicken	Monash Nursing	Diablo2_5
Copy Test 1 – (Time * Load = Proc. Time)	10s * 15.3% = 1.530s	30s * 13.4% = 4.020s	85s * 10.5% = 8.925s
Copy Test 2	11s * 15.3% = 1.683s	30s * 13.3% = 3.990s	89s * 10.2% = 9.078s
Copy Test 3	10s * 15.1% = 1.510s	33s * 11.8% = 3.894s	80s * 11.1% = 8.880s
Average Copy Time	**1.5743s**	**3.9680s**	**8.9610s**
Cipher Test 1 – (Time * Load = Proc. Time)	10s * 59.9% = 5.990s	27s * 54.8% = 14.796s	88s * 29.3% = 25.784s
Cipher Test 2	9s * 61.5% = 5.535s	26s * 58.7% = 15.262s	95s * 27.5% = 26.125s
Cipher Test 3	10s * 58.8% = 5.880s	28s * 52.3% = 14.644s	87s * 29.4% = 25.578s
Average Cipher Time	**5.8017s**	**14.9007s**	**25.8290s**
Cipher Only Time	**4.2274s**	**10.9327s**	**16.8680s**
Approximate CPU Load at Real-Time	2.64%	4.08%	3.12%
Approximate Cipher Processing Rate	63.7 Mb/s	67.0 Mb/s	80.3 Mb/s

Table D-7 Basic Performance Results on Test Platform 1

- Once we have determined the approximate time at 100% CPU load to encrypt the file, we can combine this value with the average bitrate of the test file to obtain two values, one the approximate CPU load (in %) required to encrypt or decrypt the data in real-time, and the other to determine the maximum processing rate of the cipher itself on the test platform.

The results from this experiment are listed in Table D-7 and Table D-8, note that there are no listed results for the first three test files "tennis.mpg", "flowg.mpg" and "us.mpg" as the time measurements for encrypting these files were approximately 1 second. This small measurement meant the error in the measured CPU load was much larger and any calculated results could not be used as an accurate determination of the performance of the cipher.

The results for both platforms indicate that the cipher is very fast, requiring between 2.5 and 4 % of available CPU cycles on the Pentium II and less than 1 % of available CPU cycles on the Pentium 4. The variability in these results is due to the differing average bitrates of the encoded video streams, indeed while there are only three test files, we can see the approximate cipher processing rate is fairly consistent. The highest rate test file – at 2.7 Mb/s – was "Monash Nursing.mpg", the results indicate that this rate requires just over 4% of available CPU cycles on a Pentium II and an extremely small 0.55% of cycles on the faster Pentium 4. This bitrate is considered high for an MPEG-1 encoded stream. If software decoding and display on a Pentium II requires less than 96% of available CPU cycles, we should be able to decrypt and decode the stream in real-time. An issue could arise when considering extending the cipher to support MPEG-2 streams at rate of about 10 Mb/s – this would require 4 times the number of free CPU cycles, however it is extremely unlikely that a Pentium II would have the processing power to decode a 10 Mb/s MPEG-2 stream, even without encryption.

Statistic	Chicken	Monash Nursing	Diablo2_5
Copy Test 1 – (Time * Load = Proc. Time)	5s * 9.7% = 0.485s	13s * 12.1% = 1.573s	29s * 7.1% = 2.059s
Copy Test 3	5s * 10.3% = 0.515s	11s * 12.8% = 1.408s	28s * 8.8% = 2.464s
Copy Test 2	4s * 13% = 0.520s	12s * 11.3% = 1.356s	29s * 8.4% = 2.436s
Average Copy Time	**0.5067s**	**1.4457s**	**2.3197s**
Cipher Test 1 – (Time * Load = Proc. Time)	4s * 33.8% = 1.352s	11s * 28.8% = 3.168s	28s * 17.9% = 5.012s
Cipher Test 2	5s * 31.2% = 1.560s	10s * 29.1% = 2.910s	27s *20.4% = 5.508s
Cipher Test 3	4s * 41.2% = 1.648s	10s * 27.0% = 2.700s	28s * 19.5% = 5.460s
Average Cipher Time	**1.5200s**	**2.9260s**	**5.3267s**
Cipher Only Time	**1.0133s**	**1.4803s**	**3.0070s**
Approximate CPU Load at Real-Time	0.633%	0.552%	0.556%
Approximate Cipher Processing Rate	265.8 Mb/s	494.7 Mb/s	450.2 Mb/s

Table D-8 Basic Performance Results on Test Platform 2

These results show the approximate number of CPU cycles required to perform decryption of the MPEG-1 test streams, however the real test is to determine whether or not the decryption can be performed in real-time, allowing the decrypted stream to be decoded and displayed to the user. In the following test I propose to take the encrypted files and play them back using the DirectShow (Microsoft, 2001b; Microsoft, 2001a) decryption filter developed as part of this work. The DirectShow Filter Graph editor will be used to assemble a filter graph that reads the encrypted files, de-multiplexes the MPEG-1 System Stream using the MPEG-1 System Stream Filter, decrypts the encrypted Video and Audio Streams using the decryption filter before decoding and displaying the decrypted streams using the generic MPEG-1 Video and Audio Stream Filters as well as the generic Video and Audio Renderer Filters. The Filter graph used to achieve this is shown in Figure D-1.

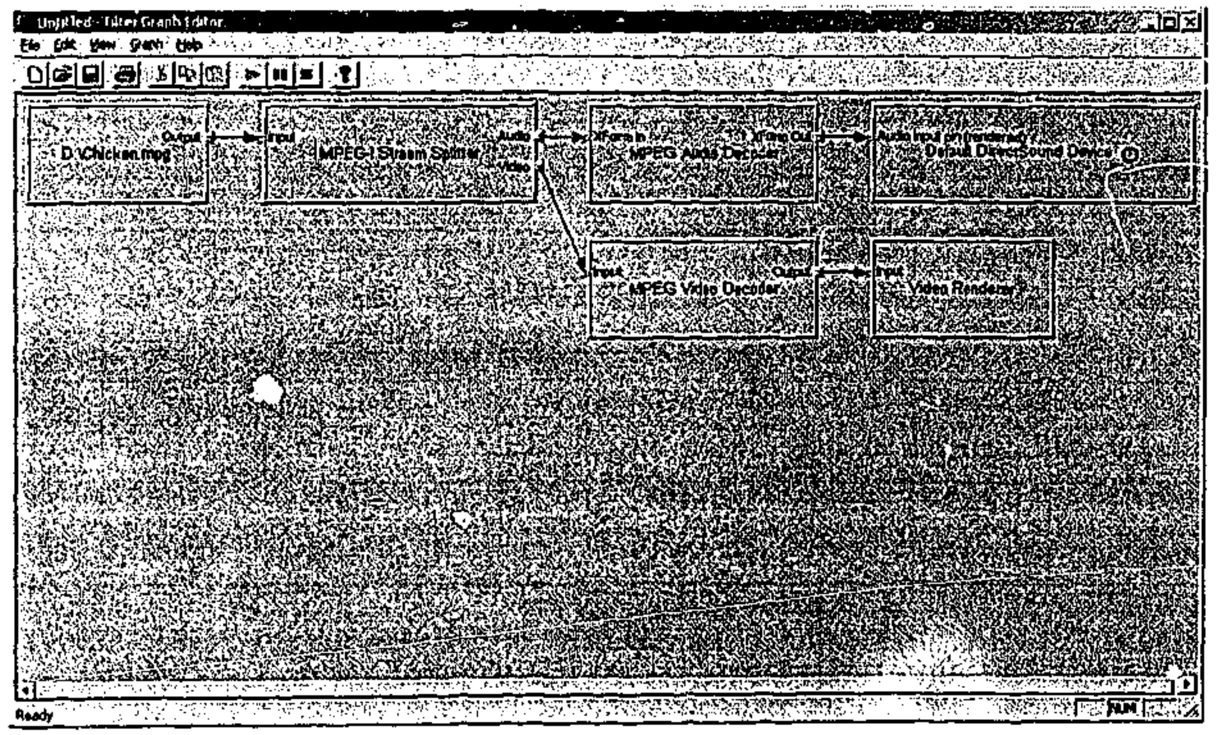


Figure D-1: DirectShow Filter Graph for Encrypted Video Playback from NetShow Theatre

Like for the previous test, we would like to determine an approximate value for the CPU resources required to decrypt each encrypted stream. This would provide a figure which we could compare to those from Table D-7 and Table D-8. The procedure here is slightly different, primarily because processing always occurs in real-time, the execution time is equal to the duration of the encoded MPEG-1 System Stream, also decryption is only ever performed in real-time, not at the cipher processing rate as determined from the previous experiment. As such, in this case we are only interested in the system average CPU load during playback only versus the load during decryption and playback. The procedure is outlined below:

- Directly playback of the plaintext MPEG-1 stream – noting the CPU load required to perform the procedure. Repeat this step three times, then obtain an average value for the CPU cycle requirements. Playback is performed full screen unless the test platform is unable to perform this task with available computing resources, in this case playback is performed at the default size of the encoded video stream.

- Using the Filter Graph Editor, playback the encrypted MPEG-1 stream – noting the CPU load required to perform the procedure. Again, repeat three times, then obtain an average value for the CPU cycle requirements. Playback is again performed full screen.

- Subtracting the result from step 1 from the value obtained in step 2 will provide an average value for the CPU processing requirements required to perform the decryption only on the encrypted MPEG-1 file during playback.

- Compare the value obtained in step 3 with those previously obtained.

Statistic	tennis#	flowg#	us#	Chicken #	Monash Nursing#	Diablo2_5#
Plaintext Playback – CPU Loads	47.8% 44.2% 49.6%	60.0% 56.7% 60.8%	41.7% 40.1% 42.1%	87.6 % 87.5% 87.9%	56.3% 56.2% 55.9%	59.3% 59.5% 60.0%
Playback – Average Load	**47.20%**	**59.17%**	**41.30%**	**87.67%**	**56.13%**	**59.60%**
Encrypted Playback – CPU Loads	51.2% 50.6% 48.6%	64.7% 65.8% 64.1%	43.7% 43.1% 43.8%	90.3% 89.7% 90.4%	61.5% 61.3% 62.1%	63.3% 63.5% 63.5%
Decryption and Playback – Average Load	**50.13%**	**64.87%**	**43.53%**	**90.13%**	**61.63%**	**63.43%**
Cipher Only - CPU Load	**2.93%**	**5.70%**	**2.23%**	**2.46%**	**5.50%**	**3.83%**
Previous Prediction	**N/A**	**N/A**	**N/A**	**2.64%**	**4.08%**	**3.12%**

# – Test platform was unable to playback these files full screen.

Table D-9 Real-time Decryption and Playback Performance Results on Test Platform 1

The results of this test can be found in Table D-9 and Table D-10. While real-time decryption and full-screen playback was not always possible, tests indicate that full-screen playback of the plaintext stream alone on the same platform was not possible – wherever full-screen playback was possible, it was always possible to perform real-time decryption and full-screen playback. However, this will not always be the case, there will be some scenarios where full-screen playback is just barely possible, with the remaining CPU cycles not being sufficient to provide real-time decryption. The promising outcome of these experiments however show that the CPU load required to perform the decryption is a small fraction of that required to perform the decoding of the MPEG-1 stream. On most modern desktop computers running a Pentium 4 processor or similar, real-time decryption and playback is well within the realms of possibility, with a large amount of processor time remaining available, indeed we can speculate that it may be possible to perform real-time decryption and playback on a modern computer of a much higher bitrate MPEG-2 stream if the cipher is extended as suggested in Chapter 6. The results also show that while a slower and older Pentium II based platform may struggle to provide real-time decryption and decoding, it would suffer the same problems in simple decoding and playback anyway. However, this type of platform would have no problems in performing real-time decryption if decoding and playback was handled by a hardware based MPEG decoder as would be likely in any black-box application based on Pentium II like hardware.

Statistic	tennis	flowg	us	Chicken	Monash Nursing	Diablo2_5
Plaintext Playback – CPU Loads	11.7% 12.2% 11.9%	17.6% 15.6% 17.2%	8.5% 8.4% 7.4%	11.9% 12.1% 12.3%	15.4% 14.9% 15.2%	15.3% 14.8% 15.0%
Playback – Average Load	**11.93%**	**16.80%**	**8.1%**	**12.10%**	**15.17%**	**15.03%**
Encrypted Playback – CPU Loads	12.5% 13.9% 13.3%	17.8% 18.3% 18.0%	9.0% 8.4% 9.4%	12.8% 13.0% 12.7%	16.1% 16.0% 15.8%	15.7% 15.3% 15.7%
Decryption and Playback – Average Load	**13.23%**	**18.03%**	**8.93%**	**12.83%**	**15.97%**	**15.57%**
Cipher Only - CPU Load	**1.30%**	**1.23%**	**0.83%**	**0.73%**	**0.80%**	**0.54%**
Previous Prediction	**N/A**	**N/A**	**N/A**	**0.633%**	**0.552%**	**0.556%**

Table D-10 Real-time Decryption and Playback Performance Results on Test Platform 2

D.3.4 Testing Functionality

Having verified that the cipher process is reversible – the original plaintext could be retrieved from the ciphertext – and that real-time decryption and playback was possible given current computing power, it now becomes necessary to verify that it is possible to perform real-time decryption and playback of streaming video, supporting all digital playback modes. In this section I will discuss the last set of tests intended to verify that the designed cipher is compatible with a range of different streaming server products. For these tests, I will use three different streaming servers, testing the capabilities of each to varying degrees, in order to prove that existing server products are not required to have an understanding of the MPEG-1 Cipher algorithm and can be used as provided to stream protected video.

D.3.4.1 Microsoft NetShow Theatre Streaming Server

The first server under test is Microsoft NetShow Theatre, a Windows product that can stream multiple concurrent high bitrate streams. The server configuration is made up of three workstations running multi-platform software to together provide a streaming video solution. One workstation provides the server management functionality as well as maintaining a database of content installed on the server. The actual content is installed on the remaining two workstations, these computers share the load of streaming any requested video, each taking turns in streaming a short segment (approx. 1 second). If one platform fails, the failure is detected and the other machine takes up the entire load, thus providing a degree of failsafe operation. Each workstation is equipped with a 155Mbps ATM Network Card and the combined server is capable of streaming approximately 280Mbps of different concurrent video streams. While Microsoft has currently discontinued development of the NetShow Theatre product, it offers a stable streaming server platform and can still be used to implement a high bitrate streaming video solution. Even though the product itself is now obsolete, it is still a viable test platform for proving cipher functionality with a range of server products as other current and future products will be similar in scope and basic design to this system.

The NetShow Theatre platform offers a DirectShow capable source filter, enabling full DirectShow compatibility when streaming video. This means that the DirectShow Cipher Filter can be utilised to decrypt the streaming video in real-time prior to passing it to the decoding and rendering filters for display. Unfortunately we cannot utilise the Filter Graph Editor to construct and test functionality as this only enables play and pause functionality. Instead we can use the developed test application – *"StreamCipher.exe"* – to playback the encrypted video stream. This application can be used to provide seek functionality and high-speed playback modes on top of the basic play and pause functions.

In order to correctly perform the tests outlined below, we will install a copy of each plaintext test file on the server along with all three encrypted copies of the test file. The plaintext bitstream will be used as a baseline test to prove that normal streaming and playback functions with un-encrypted files. The other installed files will be the corresponding bitstream encrypted with the keys **0xff**, **0x4a** and **0x42**. The Microsoft NetShow Theatre server will only stream a bitstream if it conforms to the MPEG-1 System Stream format, this means that it is not possible to perform the following series of tests on the first three test files – "tennis.mpg", "flowg.mpg" and "us.mpg". This does not invalidate the cipher as designed as the original plaintext files can also not be installed onto the streaming server.

We can now use the client playback application to run a series of tests against the installed files to check for functionality. At this stage we are no longer directly measuring the cipher performance as per the previous section but note that performance can be verified to be at an acceptable level if playback occurs without any visible or audible glitches (the stream is correctly decrypted in real-time before it is required by the subsequent decoders), all tests are performed using visual inspection to confirm correct functionality. The details of each individual test are outlined below and the results are presented in Table D-11, where success for streaming of the encrypted stream signifies success in testing all three encrypted bitstreams of the same video asset.

- **Installation** – Attempt to use the provided Microsoft NetShow Theatre Server user interface to install the test files, both in plaintext and encrypted form. Since the server will only accept installation of files that it can correctly parse and stream, this test will ensure that the modifications made to the plaintext bitstream for Copyright protection do not limit the ability of the file to be streamed from a variety of existing streaming server platforms.
- **Real-time Streaming** – Each test file is streamed and played back using the client playback application. In the case of the plaintext bitstream, we check that correct playback (both visible and audible) occurs. When playing back the encrypted streams, we also check for correct playback to occur. If an error occurs during playback, this could indicate one of two possibilities, either the cipher cannot decrypt the streaming video correctly or there are not enough CPU cycles available to perform the task. Assuming all functions correctly (as expected since playback will involve a similar filter graph to that used to test real-time decryption and playback), we can show that the cipher design permits real-time streaming and playback with the Streaming Server.

- **Pause and Play** – Each test file is streamed while randomly selecting the pause and play features from the player controls. Since control of the streaming modes is remotely controlled by the client playback application to the server, it is impossible to guarantee that the same timestamps will be exactly selected on subsequent experiments, random selection of timestamps to test streaming restarts will enable us to say with confidence that the cipher design correctly handles pauses in playback of the content from the Streaming Server.
- **Indexed Playback** – Each test file is streamed while randomly shifting the media position bar forwards and backwards through the stream, the randomness in timestamps is used for the same reasons outlined above. This test is used to verify that the cipher design permits streaming from random resynchronisation points within the content, given that the streaming server will select the actual point where streaming occurs based on the selected timestamp and the contents of the bitstream to be at the start of a Group of Pictures. By selecting both forward and backwards jumps, the test proves the ability of the cipher to decode the stream from any point in the encrypted bitstream.
- **High Speed Playback** – Each test file is streamed in high-speed, both fast forward and rewind, to ensure that the cipher can correctly decrypt these special streams. This basic test will ensure that the cipher design can correctly cope with high-speed streams, usually implemented as a sequence of individual I-Frames within the original video stream. Also tested at this stage is seek functionality within the high-speed playback modes, testing the ability of the cipher to commence real-time decryption and playback at any random playback point within the bitstream during high-speed playback. Of course, with the first three test streams being too short for testing seek functionality, they are obviously also too short to test the high-speed playback modes.
- **Random Playback Modes** – The final test involves the random selection of different playback modes of the encrypted stream and ensures that correct playback ensues. This includes all of the above tests as well as changing all possible changes in playback states between play, pause, fast forward and rewind. While the above tests should also confirm the expected results of this test, this test should ensure that random selection of any possible function available to the user can be correctly handled by the cipher module – the stream is correctly decrypted and displayed back to the user.

As expected, the results when streaming the plaintext file are all successful. These tests perform basic functionality tests of the streaming servers without considering the concept of the encrypted bitstream. Given that all three plaintext bitstreams were generated from independent sources and encoders, we can surmise that the same results can be extended to the all MPEG-1 compliant bitstreams. Of more interest are the results when streaming the encrypted files, in this case we are actually testing the prototype cipher while streaming the encrypted bitstreams from the server. These test results verify that the prototype cipher can correctly decrypt the stream delivered from the server in all respective playback modes. The results also show that the resultant decrypted stream is then correctly decoded for subsequent playback, thereby proving the viability of the cipher design to support all of the digital playback modes supported by the server.

Movie and Cipher Key	Installation Test	Real-Time Streaming	Pause and Play	Indexed Playback	High-speed Playback	Random Playback Modes
Chicken – Plaintext	☑	☑	☑	☑	☑	☑
Chicken – 0xff	☑	☑	☑	☑	☑	☑
Chicken – 0x4a	☑	☑	☑	☑	☑	☑
Chicken – 0x42	☑	☑	☑	☑	☑	☑
Monash Nursing – Plaintext	☑	☑	☑	☑	☑	☑
Monash Nursing – 0xff	☑	☑	☑	☑	☑	☑
Monash Nursing – 0x4a	☑	☑	☑	☑	☑	☑
Monash Nursing – 0x42	☑	☑	☑	☑	☑	☑
Diablo2_5 – Plaintext	☑	☑	☑	☑	☑	☑
Diablo2_5 – 0xff	☑	☑	☑	☑	☑	☑
Diablo2_5 – 0x4a	☑	☑	☑	☑	☑	☑
Diablo2_5 – 0x42	☑	☑	☑	☑	☑	☑

Table D-11 Functionality Tests with Microsoft NetShow Theatre Streaming Server

D.3.4.2 SGI Mediabase 3.1 Streaming Server

The second server under test is Mediabase Version 3.1 as produced by Silicon Graphics (SGI). At Monash University, this streaming server software is installed on an SGI Challenge L workstation comprising 8 individual processors. The Mediabase streaming server platform is no longer maintained by SGI, it is now maintained and kept by Kasenna (www.kasenna.com) and has progressed to Version 5.0, however protocol for streaming MPEG-1 remains unchanged. In the test configuration, the Challenge L workstation is connected to the network via a 155Mb/s ATM Network Card. Like the Microsoft NetShow Theatre streaming server product, Mediabase support all possible digital video playback modes – indexed playback, high-speed playback, pause – and can therefore be used to prove the functionality of the designed cipher.

Like NetShow Theatre, the Mediabase platform also provides a DirectShow capable source filter, however, the Mediabase DirectShow filter does not provide support for the high-speed playback modes. As for testing the NetShow Theatre platform, we can use the *"StreamCipher.exe"* application to playback an encrypted video stream from the Mediabase server, noting that it is not possible to confirm functionality during high-speed playback using this application. The second test application – *"SGIStreamCipher.exe"* – can instead be used to test high-speed playback. As described in Appendix C.6.4, this application does not provide real-time decoding and playback but instead saves the decrypted stream to disk for later playback and confirmation of successful retrieval of a viewable plaintext stream.

Unlike the NetShow product, Mediabase will accept installation bitstreams that conform to the MPEG-1 Video Stream format as well as the System Stream format – this means that all six test bitstreams can be successfully installed onto the Mediabase server. The tests performed replicate the aforementioned tests for the NetShow Theatre streaming server. The installation test is performed once since there is only the one server product under test. Similarly high-speed playback, indexed high-speed playback and the random selection of playback modes can only be tested using the *"SGIStreamCipher.exe"* application. On the other hand, the tests involving real-time streaming, pause during playback and indexed playback are performed using both the *"StreamCipher.exe"* application and the *"SGIStreamCipher.exe"* application. Confirmation of functionality is performed visually and audibly, either in real-time in the case of DirectShow playback or through later playback of the created plaintext bitstreams during use of the *SGIStreamCipher* application. Since neither of the two test applications is able to playback an MPEG-1 Video Stream directly, these tests must be performed through manual construction of a Filter Graph in the Filter Graph editor – note that the three Video Stream only bitstreams form sequences that are too short to adequately test indexed or high-speed playback modes in any case.

The results of tests when streaming from the Mediabase server – shown in Table D-12 – are all successful. In each case all test bitstreams were successfully installed onto the streaming server. Of the three Video only bitstreams, these files were successfully streamed, decrypted and played back in real-time using a manual Filter Graph construction – as previously mentioned, the assets were too short to test indexed or high-speed playback and the *"SGIStreamCipher.exe"* application did not support Video only streaming. Of more interest was the testing of the three longer MPEG-1 System Stream conformant bitstreams. Using the DirectShow enabled test application, it can be verified that real-time decryption and playback, paused playback and indexed playback of an encrypted bitstream was successful. Usage of the *"SGIStreamCipher.exe"* application further confirmed that high-speed playback, indexed high-speed playback and random playback mode changes on an encrypted video stream server by Mediabase resulted in the correct decryption of a viable MPEG-1 bitstream.

The conclusion to be drawn from this result is that the Mediabase streaming server can deliver a previously encrypted bitstream in all playback modes and that the subsequent stream can be correctly decoded for playback at the client end. This proves the viability of the cipher design to support all digital playback modes supported by Mediabase.

Movie and Cipher Key	Installation Test	Real-Time Streaming	Pause and Play	Indexed Playback	High-speed Playback	Random Playback Modes
tennis – Plaintext	☑	☑*	☑*	N/A	N/A	N/A
tennis – 0xff	☑	☑*	☑*	N/A	N/A	N/A
tennis – 0x4a	☑	☑*	☑*	N/A	N/A	N/A
tennis – 0x42	☑	☑*	☑*	N/A	N/A	N/A
flowg – Plaintext	☑	☑*	☑*	N/A	N/A	N/A
flowg – 0xff	☑	☑*	☑*	N/A	N/A	N/A
flowg – 0x4a	☑	☑*	☑*	N/A	N/A	N/A
flowg – 0x42	☑	☑*	☑*	N/A	N/A	N/A
us – Plaintext	☑	☑*	☑*	N/A	N/A	N/A
us – 0xff	☑	☑*	☑*	N/A	N/A	N/A
us – 0x4a	☑	☑*	☑*	N/A	N/A	N/A
us – 0x42	☑	☑*	☑*	N/A	N/A	N/A
Chicken – Plaintext	☑	☑	☑	☑	☑#	☑#
Chicken – 0xff	☑	☑	☑	☑	☑#	☑#
Chicken – 0x4a	☑	☑	☑	☑	☑#	☑#
Chicken – 0x42	☑	☑	☑	☑	☑#	☑#
Monash Nursing – Plaintext	☑	☑	☑	☑	☑#	☑#
Monash Nursing – 0xff	☑	☑	☑	☑	☑#	☑#
Monash Nursing – 0x4a	☑	☑	☑	☑	☑#	☑#
Monash Nursing – 0x42	☑	☑	☑	☑	☑#	☑#
Diablo2_5 – Plaintext	☑	☑	☑	☑	☑#	☑#
Diablo2_5 – 0xff	☑	☑	☑	☑	☑#	☑#
Diablo2_5 – 0x4a	☑	☑	☑	☑	☑#	☑#
Diablo2_5 – 0x42	☑	☑	☑	☑	☑#	☑#

N/A Not possible to test through Filter Graph editor, test applications could not playback a MPEG-1 Video Stream conformant bitstream.

* Playback could only be effected via manual construction of a Filter Graph using the Filter Graph editor.

# Test could only be performed using *"SGIStreamCipher.exe"*

Table D-12 Functionality Tests with SGI Mediabase 3.1 Streaming Server

D.3.4.3 Apple QuickTime Streaming Server

The final server platform under test is the Apple QuickTime and/or Apple Darwin Streaming Server. Both products are offering from Apple and compiled from the same code base, the differences being that the Apple QuickTime Streaming Server contains code optimised for execution on Apple hardware and the fact the Apple offers product support on the QuickTime Streaming Server. The server configuration consists of a Linux platform running the Darwin Streaming Server software. In order to install an MPEG-1 movie for streaming from either the QuickTime server or the Darwin server, it is necessary to convert the original file from an MPEG-1 System Stream comformant bitstream to an Apple QuickTime Hinted Movie bitstream. This transformation can be performed using a registered version of the Apple QuickTime Player software by choosing the *"Export..."* option from the **File** menu. The process of hinting a movie file inserts extra information into the file that allows the QuickTime or Darwin servers to offer indexed playback of the original MPEG-1 bitstream across a network using the RTSP and RTP protocols.

The first test in verifying the functionality of the cipher is to ensure that the encrypted bitstream can be successfully installed onto the server. The procedure differs somewhat for the QuickTime and Darwin servers as the file must first be hinted prior to installation. For testing purposes, all the plaintext and encrypted test files were hinted using QuickTime Pro Version 6.0 for Windows and then successfully installed onto the server. No problems were encountered during this stage thus showing that either server type could successfully stream the encrypted bitstreams in all supported playback modes – normal playback, pause and indexed playback.

Both the Apple QuickTime and Darwin Streaming Servers will stream an installed hinted MPEG-1 bitstream using the RTSP and RTP protocols, this means that a suitable RTSP capable client player must be used to retrieve the stream. Under normal circumstances, the Apple QuickTime player can be used to receive, decode and playback any stored video file. To subject the QuickTime or Darwin Servers to the same battery of tests listed in D.3.4.1 and D.3.4.2, it would be necessary to construct a playback application to retrieve, decrypt and decode the encrypted bitstream from the server. This task is not impossible, indeed Apple provide an SDK (Apple, 2002a) similar in scope to DirectShow that would enable development of a decryption module that could be used in the same way as the MPEG-1 Cipher Filter to provide real-time decryption and playback of the encrypted bitstream.

While developing a QuickTime/Darwin client player application that could stream and playback an encrypted file is possible, the work required to perform this task is equivalent to the work required to build the DirectShow filter for use in playback of encrypted streams from a DirectShow capable server. It was deemed to be too time-consuming to also develop a decryption filter for the QuickTime architecture due to the time and effort required to integrate with a complex system – a large amount of time was also invested in developing the decryption filter to integrate into the DirectShow environment. As such, testing of the Apple QuickTime and Darwin Streaming Servers was limited to verifying that the encrypted bitstream could successfully be hinted and installed onto the server platforms. The design of the server ensures that successful hinting and installation would ensure that the encrypted MPEG-1 bitstream will be properly streamed to the client.

D.4 SEAL Based Cipher

In this section I will discuss the tests performed to verify the viability and functionality of the prototype cipher designed in Chapter 5. The same set of tests are performed as for testing the prototype cipher with the one obvious difference being the selection of keys (the SEAL cipher takes 160-bit keys as opposed to 8-bit keys for the prototype cipher). The procedures for the experiments remain constant with those of the previous section, the results are tabulated in the following sub-sections.

D.4.1 Testing Repeatability

The procedure involved to confirm the repeatability of the designed MPEG-1 cipher using SEAL as a base cipher is exactly the same as that to prove repeatability of the prototype cipher. The key difference is in the selection of the secret key to use for testing purposes. This is an issue as the prototype cipher required only an 8-bit key whereas the final cipher design requires a 160-bit key. Using the same approach as that used, in Section D.3.1, for testing purposes we will use a key composed of 160 1-bits – (`0xffffffffffffffffffffffffffffffffffffffff`). Again the aim of this experiment is to confirm, through multiple executions of the cipher, that the encrypted bitstream is produced consistently. As in Section D.3.1, the experiment is repeated for each of the six test files, the results are tabulated in Table D-13.

As can be seen from the tabulated results, all tests in this case were successful, just as for the prototype cipher. The three resultant files from each input test file were all equal, indicating that the process of applying the 160-bit key to the input bitstream is predictable and able to produce consistent results, and concluding that it is possible to consistently reverse the results to obtain the original bitstream.

Statistic	tennis	flowg	us	Chicken	Monash Nursing	Diablo2_5
Video Stream Selected	1,239,953	2,813,788	2,049,416	30,542,212	83,500,150	150,153,682
Video Stream Encrypted	1,187,354 (95.8%)	2,684,552 (95.4%)	1,955,732 (95.4%)	25,867,063 (84.7%)	78,605,323 (94.1%)	142,369,502 (94.8%)
Audio Stream Selected	0	0	0	2,528,722	6,385,721	12,888,252
Audio Stream Encrypted	0	0	0	2,512,183 (99.3%)	6,350,629 (99.4%)	12,814,492 (99.4%)
Three Output Files Equal	☑	☑	☑	☑	☑	☑

Table D-13 Encryption Statistics and Repeatability of Encryption Process – SEAL Cipher

If we once again examine the proportion of bytes actually encrypted against the number of bytes selected for encryption, we notice a slight decrease in the number of Video Stream bytes left as plaintext but a marked decrease in the number of Audio Stream bytes left as plaintext. Indeed, in the case of encrypting the Audio Stream, the proportion of plaintext bytes where those bytes were selected

for encryption is now approaching the expected value ($^2/_{256}$ or approx. 0.78%). This improvement is primarily due to the change in the conditions for no change to occur – from a plaintext byte being equal to either 0xff or 0x00 to a plaintext byte being equal to either 0xff or a pseudo-random value selected by the SEAL cipher. This change means that the number of bytes left as plaintext is less reliant on the incidences of a single byte value (the 8-bit prototype key) and more subject to chance, therefore leading to a value closer to that expected.

When considering the Video Stream, we only see a slight improvement in the figures. Again, this result can be explained as in Section D.3.1:

- Firstly, for every slice in the MPEG-1 Video Stream, the number of bytes determined as selected for encryption is three higher than what it should be. This leads to a scenario where the reported figures are skewed due to an incorrect determination of the number of bytes selected for encryption.

- Secondly, this result is also in part determined by the results contained in Table D-4, since the frequency of 0x00 and 0x01 bytes is higher than expected for a purely random bitstream, and these bytes are never encrypted in a Video Stream, we can conclude that again a large number of bytes remain as plaintext due to the nature of an MPEG-1 Video Stream.

D.4.2 Testing Reversibility

The next step is to confirm the reversibility of the encryption procedure, again the procedure is exactly the same as that described in Section D.3.2, except for the choice of keys. Also similar is the aim of this series of tests, to prove that if the same key used during encryption is applied to the encrypted bitstream, then the original MPEG-1 bitstream can be re-obtained, and to verify that if an incorrect key is applied to the encrypted bitstream, then the original bitstream is not obtained, but rather a different, invalid MPEG-1 bitstream is obtained instead.

As discussed in the previous section, while there are 256 possible keys that can be applied using the prototype cipher, the SEAL based cipher has the capability of supporting up to 2^{160} different keys. Obviously, it is not feasible to perform a complete test using all these keys – therefore, like when testing the prototype cipher, a subset of four keys will be used for testing purposes, these four keys are:

- Key 1 (`0xffffffffffffffffffffffffffffffffffffffff`) – While not necessarily being as obvious a choice as for the prototype cipher, this key still remains suitable as a test key for SEAL, it is not a weak key in any respect and is easy to enter for testing purposes.

- Key 2 (`0x0000000000000000000000000000000000000000`) – Unlike in the prototype cipher where a key of 0x00 resulted in an unchanged ciphertext bitstream, the SEAL cipher ensures that a key consisting of 160 zero bits produces a suitable pseudo-random output for encryption purposes. Again, the key is not weak in any way and like Key 1, is easy to enter for testing purposes.

- Key 3 (0x546865204d504547205345414c20636970686572) – This key corresponds to the ASCII code sequence for the 20 character string "The MPEG SEAL cipher". A phrase of 20 characters was chosen to produce a 20 byte key sequence for convenience. Again, SEAL has no weak keys and any one key is as suitable as any other unique key. Using this approach provides a convenient technique for selecting one of over 10^{48} different keys.

		Decryption Key			
Filename	Encryption Key	Key 1	Key 2	Key 3	Key 4
tennis.mpg	Key 1	☑	☒	☒	☒
	Key 2	☒	☑	☒	☒
	Key 3	☒	☒	☑	☒
	Key 4	☒	☒	☒	☑
flowg.mpg	Key 1	☑	☒	☒	☒
	Key 2	☒	☑	☒	☒
	Key 3	☒	☒	☑	☒
	Key 4	☒	☒	☒	☑
us.mpg	Key 1	☑	☒	☒	☒
	Key 2	☒	☑	☒	☒
	Key 3	☒	☒	☑	☒
	Key 4	☒	☒	☒	☑
Chicken.mpg	Key 1	☑	☒	☒	☒
	Key 2	☒	☑	☒	☒
	Key 3	☒	☒	☑	☒
	Key 4	☒	☒	☒	☑
Monash Nursing.mpg	Key 1	☑	☒	☒	☒
	Key 2	☒	☑	☒	☒
	Key 3	☒	☒	☑	☒
	Key 4	☒	☒	☒	☑
Diablo2_5.mpg	Key 1	☑	☒	☒	☒
	Key 2	☒	☑	☒	☒
	Key 3	☒	☒	☑	☒
	Key 4	☒	☒	☒	☑

Table D-14 Comparison of Decrypted File with Original File – SEAL Cipher

- Key 4 (0x4a61736f6e204275742050684420546865736973) – This key was selected using the same technique as Key 3, only this key corresponds to the ASCII code sequence for the 20 character string "Jason But PhD Thesis".

The procedure for application of these four keys is identical to that described in Section D.3.2, the results from this test are shown in Table D-14. These results confirm the functionality of the cipher in reversing the encryption process, the original bitstream can only be retrieved if the correct key is used for decryption. Visual confirmation indicated that these files were able to be decoded and played back normally. The results also verified that the decoder could not playback or display any of the resultant output files that were decrypted using an incorrect key

D.4.3 Performance Testing

As for testing the prototype cipher, the same two test computers specified in Table D-6 were used as test platforms to verify the performance characteristics of the MPEG-1 cipher, also the same test procedures explained in Section D.3.3 are used to produce the results reported in Table D-15, Table D-16, Table D-17 and Table D-18, the major differences being:

- The choice of test key – in this case the key is **Key 1** as described in the previous section.
- We can re-use the measurements of file-copying from testing the prototype cipher since there was no encryption performed when taking these measurements.
- We can re-use the measurements of CPU Load during plaintext playback since there was no decryption performed when taking these measurements.

Statistic	Chicken	Monash Nursing	Diablo2_5
Average Copy Time	**1.5743s**	**3.9680s**	**8.9610s**
Cipher Test 1 – (Time * Load = Proc. Time)	10s * 81.3% = 8.130s	26s * 76.7% = 19.942s	85s * 42.9% = 36.465s
Cipher Test 2	9s * 79.5% = 7.155s	26s * 77.9% = 20.254s	90s * 40.3% = 36.270s
Cipher Test 3	10s * 73.5% = 7.350s	25s * 79.4% = 19.850s	84s * 43.3% = 36.372s
Average Cipher Time	**7.5450s**	**20.0153s**	**36.3690s**
Cipher Only Time	**5.9707s**	**16.0473s**	**27.4080s**
Approximate CPU Load at Real-Time	3.73%	5.99%	5.07%
Approximate Cipher Processing Rate	45.1 Mb/s	45.6 Mb/s	49.4 Mb/s

Table D-15 Basic Performance Results on Test Platform 1 – SEAL Cipher

Again, there are no results for the first three test files in the first series of tests because the encryption time was too short to provide any meaningful results. The results of the basic tests on both platforms indicate that while the cipher is not quite as fast as the prototype cipher, it is still very fast, requiring between 3.7 and 6 % of available CPU cycles on the Pentium II and less than 1.2 % of available CPU cycles on the Pentium 4. The results also show that the approximate cipher processing rate is fairly consistent on a given platform. We can also make the same assumptions on the applicability to real-time decryption and decoding, the required CPU load to perform the decryption is relatively low and should enable a processor that can decode and display a given stream to be able to also decrypt it at the same time.

Statistic	Chicken	Monash Nursing	Diablo2_5
Average Copy Time	**0.5067s**	**1.4457s**	**2.3197s**
Cipher Test 1 – (Time * Load = Proc. Time)	4s * 38.2% = 1.528s	11s * 41.9% = 4.609s	28s * 28.8% = 8.064s
Cipher Test 2	3s * 70.3% = 2.109s	11s * 40.5% = 4.455s	29s * 25.8% = 7.482s
Cipher Test 3	3s * 64.1% = 1.923s	11s * 41.9% = 4.609s	27s * 29.9% = 8.073s
Average Cipher Time	**1.8533s**	**4.5577s**	**7.8730s**
Cipher Only Time	**1.3467s**	**3.1120s**	**5.5533s**
Approximate CPU Load at Real-Time	0.840%	1.161%	1.026%
Approximate Cipher Processing Rate	200.0 Mb/s	235.3 Mb/s	243.8 Mb/s

Table D-16 Basic Performance Results on Test Platform 2 – SEAL Cipher

Statistic	tennis#	flowg#	us#	Chicken	Monash Nursing#	Diablo2_5#
Playback – Average Load	**47.20%**	**59.17%**	**41.30%**	**87.67%**	**56.13%**	**59.60%**
Encrypted Playback – CPU Loads	51.1% 50.5% 51.9%	67.5% 68.2% 66.4%	44.7% 43.5% 45.1%	90.6% 90.8% 90.5%	62.1% 61.9% 62.7%	65.8% 66.2% 65.9%
Decryption and Playback – Average Load	**51.17%**	**67.37%**	**44.43%**	**90.63%**	**62.23%**	**65.97%**
Cipher Only - CPU Load	**3.97%**	**8.20%**	**3.13%**	**2.96%**	**6.10%**	**6.37%**
Previous Prediction	N/A	N/A	N/A	**3.73%**	**5.99%**	**5.07%**

# – Test platform was unable to playback these files full screen.

Table D-17 Real-time Playback Performance Results on Test Platform 1 – SEAL Cipher

When examining the performance of real-time decryption and playback, we obtain similar results to the prototype cipher, again showing that the CPU cycles required by the cipher is in the minority compared to those required to decode and display the resultant stream. Again, we can conclude that wherever full-screen playback was possible, it was always possible to perform real-time decryption and full-screen playback. Also again, this will not always be the case, with some scenarios where full-screen playback requires almost all the available CPU cycles, leaving none for decryption. Even though the SEAL based cipher requires more resources to perform real-time decryption and playback, we can again speculate that it should be possible to perform this task on a modern computer using a much higher bitrate MPEG-2 stream. Finally, the slower and older Pentium II based platform again struggles to provide real-time decryption and decoding, but would have minimal problems in performing real-time decryption using the more complicated cipher if decoding and playback was handled by a hardware based MPEG decoder.

Statistic	tennis	flowg	us	Chicken	Monash Nursing.	Diablo2_5
Playback – Average Load	**11.93%**	**16.80%**	**8.1%**	**12.10%**	**15.17%**	**15.03%**
Encrypted Playback – CPU Loads	13.5% 13.4% 14.1%	17.7% 18.7% 18.8%	8.6% 9.1% 8.7%	12.9% 13.1% 12.6%	16.2% 16.0% 16.3%	16.1% 16.2% 16.8%
Decryption and Playback – Average Load	**13.67%**	**18.40%**	**8.8%**	**12.87%**	**16.17%**	**16.37%**
Cipher Only - CPU Load	**1.74%**	**1.60%**	**0.70%**	**0.77%**	**1.00%**	**1.34%**
Previous Prediction	**N/A**	**N/A**	**N/A**	**0.840%**	**1.161%**	**1.026%**

Table D-18 Real-time Playback Performance Results on Test Platform 2 – SEAL Cipher

D.4.4 Testing Functionality

Finally, we need to again verify that the securely encrypted bitstreams can be streamed from all our test server platforms without suffering a loss in the functionality provided. That is, all playback modes supported by the server in question must still be supported when an encrypted bitstream is installed, and that the delivered stream can be successfully decrypted and decoded for real-time display at the client computer. In this section I will repeat the tests outlined in D.3.4 with the securely encrypted bitstreams and show that full functionality is still maintained.

D.4.4.1 Microsoft NetShow Theatre Streaming Server

The results of testing functionality of the SEAL based cipher with the Microsoft NetShow Theatre Streaming Server are presented in Table D-19, noting that installation of MPEG-1 Video Stream compliant bitstreams (such as *"tennis.mpg"*, *"flowg.mpg"* and *"us.mpg"*) is not possible using this server platform. The details of the server under test and the testing parameters are exactly the same as described in Section D.3.4.1, the primary differences being:

- It is no longer necessary to confirm functionality using the plaintext bitstream as this has already been shown.
- When testing bitstreams encrypted with the secure SEAL based cipher, the four keys outlined in Section D.4.2 are used, again specified as **Key1**, **Key2**, **Key3** and **Key4**.

These results show that like the prototype cipher, the SEAL based cipher can also correctly decrypt the stream delivered from the server in all respective playback modes, and subsequently correctly decoded for playback. As such, the secure cipher is proven to support all of the digital playback modes supported by the Microsoft NetShow Theatre Streaming Server platform.

Movie and Cipher Key	Installation Test	Real-Time Streaming	Pause and Play	Indexed Playback	High-speed Playback	Random Playback Modes
Chicken – Key 1	☑	☑	☑	☑	☑	☑
Chicken – Key 2	☑	☑	☑	☑	☑	☑
Chicken – Key 3	☑	☑	☑	☑	☑	☑
Chicken – Key 4	☑	☑	☑	☑	☑	☑
Monash Nursing – Key 1	☑	☑	☑	☑	☑	☑
Monash Nursing – Key 2	☑	☑	☑	☑	☑	☑
Monash Nursing – Key 3	☑	☑	☑	☑	☑	☑
Monash Nursing – Key 3	☑	☑	☑	☑	☑	☑
Diablo2_5 – Key 1	☑	☑	☑	☑	☑	☑
Diablo2_5 – Key 2	☑	☑	☑	☑	☑	☑
Diablo2_5 – Key 3	☑	☑	☑	☑	☑	☑
Diablo2_5 – Key 4	☑	☑	☑	☑	☑	☑

Table D-19 Functionality Tests with Microsoft NetShow Theatre Streaming Server – SEAL

D.4.4.2 SGI Mediabase 3.1 Streaming Server

The second server under test is Mediabase Version 3.1, the test platform is described fully in Section D.3.4.2. Also fully described in this section is the test procedure to confirm functionality of the server to support the encrypted bitstream under all playback modes and the ability to successfully decrypt and decode these delivered bitstreams at the client using either the *"StreamCipher.exe"* or *"SGIStreamCipher.exe"* application (or the DirectShow Filter Graph editor in the case of the three MPEG-1 Video Stream conformant bitstreams). As described in Section D.4.4.1, it is no longer necessary to test the plaintext bitstream functionality and the Keys outlined in Section D.4.2 are used to create the encrypted bitstreams. The testing results are presented in Table D-20.

As for testing the prototype cipher, the results of tests when streaming from the Mediabase server are all successful. All test bitstreams were successfully installed onto the streaming server. The three Video only bitstreams were successfully streamed, decrypted and played back in real-time using a manual Filter Graph construction. The three System bitstreams were successfully decrypted and played back in all playback modes supported by the respective playback applications *"StreamCipher.exe"* and *"SGIStreamCipher.exe"*. In conclusion, the Mediabase streaming server can deliver a securely encrypted bitstream in all playback modes and that the subsequent stream can be correctly decoded for playback at the client end, therefore proving the viability of the cipher design to support all digital playback modes supported by Mediabase.

Movie and Cipher Key	Installation Test	Real-Time Streaming	Pause and Play	Indexed Playback	High-speed Playback	Random Playback Modes
tennis – Key 1	☑	☑*	☑*	N/A	N/A	N/A
tennis – Key 2	☑	☑*	☑*	N/A	N/A	N/A
tennis – Key 3	☑	☑*	☑*	N/A	N/A	N/A
tennis – Key 4	☑	☑*	☑*	N/A	N/A	N/A
flowg – Key 1	☑	☑*	☑*	N/A	N/A	N/A
flowg – Key 2	☑	☑*	☑*	N/A	N/A	N/A
flowg – Key 3	☑	☑*	☑*	N/A	N/A	N/A
flowg – Key 4	☑	☑*	☑*	N/A	N/A	N/A
us – Key 1	☑	☑*	☑*	N/A	N/A	N/A
us – Key 2	☑	☑*	☑*	N/A	N/A	N/A
us – Key 3	☑	☑*	☑*	N/A	N/A	N/A
us – Key 4	☑	☑*	☑*	N/A	N/A	N/A
Chicken – Key 1	☑	☑	☑	☑	☑#	☑#
Chicken – Key 2	☑	☑	☑	☑	☑#	☑#
Chicken – Key 3	☑	☑	☑	☑	☑#	☑#
Chicken – Key 4	☑	☑	☑	☑	☑#	☑#
Monash Nursing – Key 1	☑	☑	☑	☑	☑#	☑#
Monash Nursing – Key 2	☑	☑	☑	☑	☑#	☑#
Monash Nursing – Key 3	☑	☑	☑	☑	☑#	☑#
Monash Nursing – Key 4	☑	☑	☑	☑	☑#	☑#
Diablo2_5 – Key 1	☑	☑	☑	☑	☑#	☑#
Diablo2_5 – Key 2	☑	☑	☑	☑	☑#	☑#
Diablo2_5 – Key 3	☑	☑	☑	☑	☑#	☑#
Diablo2_5 – Key 4	☑	☑	☑	☑	☑#	☑#

N/A Not possible to test through Filter Graph editor, test applications could not playback a MPEG-1 Video Stream conformant bitstream.

* Playback could only be effected via manual construction of a Filter Graph using the Filter Graph editor.

Test could only be performed using *"SGIStreamCipher.exe"*

Table D-20 Functionality Tests with SGI Mediabase 3.1 Streaming Server – SEAL

D.4.4.3 Apple QuickTime Streaming Server

Finally, the tests outlined in Section D.3.4.3 were repeated for the bitstreams encrypted using the secure SEAL based cipher and subsequently installed on the test platform Darwin Streaming Server. The successful conversion of all encrypted bitstreams to an Apple QuickTime Hinted Movie and subsequent installation onto the streaming server leads to the same conclusion as for the prototype cipher – The Apple QuickTime Streaming Server and the Darwin Streaming Server platforms can successfully stream the encrypted (using the SEAL based cipher) MPEG-1 bitstream in all of the digital playback modes supported by these streaming servers. No client playback application was developed to test decryption and playback of these encrypted bitstreams.

D.5 Conclusions

Both the prototype and more secure cipher passed all tests performed. Of particular interest is the SEAL based MPEG-1 bitstream cipher. The cipher design required minimal CPU cycles, enabling it to execute simultaneously with video decoding and display. The cipher design permitted installation of an encrypted bitstream onto a range of video streaming servers that had no prior knowledge of the encryption algorithm. These installed bitstreams we subsequently successfully streamed from these servers in a variety of supported playback modes. Furthermore, the bitstreams received at the client decoder were successfully decrypted and decoded in a variety of playback modes, including pause, indexed and high-speed playback.

The goals of this research were successfully met. The designed cipher can be successfully utilised with a wide range of pre-existing streaming video products. It is also likely that compatability will extend to future streaming servers as well. The cipher does not require special streaming server implementations and can support a variety of streaming playback modes. Finally, the concepts used in the cipher design can also be applied to an MPEG-2 bitstream with the effect of successful streaming of an encrypted MPEG-2 bitstream.

Appendix E

References

Abdulaziz, N. (2001)

"Digital Watermarking and Data Hiding in Multimedia", PhD Thesis, Monash University

Agi, I. and Gong, L. (1996)

"An Empirical Study of Secure MPEG Video Transmissions", In: *ISOC Symposium on Network and Distributed System Security*, pp. 137-144

Alattar, A. M. and Al-Regib, G. I. (1999)

"Evaluation of Selective Encryption Techniques for Secure Transmission of MPEG Video Bit-Streams", In: *IEEE Symposium on Circuits and Systems*, pp. 340-343

Alattar, A. M., Al-Regib, G. I. and Al-Semari, S. A. (1999)

"Improved Selective Encryption Techniques for Secure Transmission of MPEG Video Bit-Streams" Vol., 1999, 256-260.

Anderson, D. B. (1996)

"A Proposed Method for Creating VCR Functions using MPEG Streams", In: *IEEE 12th International Conference on Data Engineering*, 1996, pp. 380-382

Anderson, M. (1990)

"VCR Quality Video at 1.5 Mbits/s", In: *National Communication Forum*, pp.

Apple (2002a)

"Apple Quicktime Player". http://www.apple.com/quicktime, 2002a

Apple (2002b)

"Darwin Streaming Server Software Download and Documentation". http://developer.apple.com/darwin/projects/streaming, 2002b

Ashmawi, W., Guerin, R., Wolf, S. and Pinson, M. (2001)

"On the impact of policing and rate guarantees in Diff-Serv networks: a video streaming application perspective" *Computer Communication Review,* **Vol. 31, No. 4,** 2001, 83-95.

Aslam, T. (1998)

"Protocols for E-Commerce" *Dr. Dobbs Journal,* **Vol. December,** 1998, 52-58.

Bao, F. (2000)

"Multimedia Content Protection by Cryptography and Watermarking in Tamper-resistant Hardware", In: *Multimedia2000*, pp. 139-143

Bao, F., Sun, Q., Hu, J., Deng, R. H. and Wu, J. (1998)

"Copyright protection through watermarking: towards tracing illegal users", In: *The 6th IEEE International Workshop on Intelligent Signal Processing and Communications Systems (ISPACS'98)*, November, pp.

Binns, J. and Branch, P. (1998)

"McIVER Video-on-Demand", In: *Victorian Association for Library Automation 9th Biennial Conference*, pp. 249-261

Birk, Y. and Mondri, R. (1999)

"Tailored transmissions for efficient near video-on-demand service", In: *IEEE International Conference on Multimedia Computing and Systems*, pp.

Bloom, J., Cox, I., Kalker, T., Linnartz, J.-P., Miller, M. and Traw, B. (1999)

"Copy Protection for DVD Video" *Proceedings of the IEEE,* **Vol. 87, No. 7,** 1999, 1267-1276.

Bozoki, E. (1999)

"IP Security Protocols" *Dr. Dobb's Journal,* **Vol. December,** 1999, 42-55.

Branch, P. (1996)

"Video On Demand Trials at Monash University", In: *The Virtual University Symposium*, 1996, pp.

Branch, P. and Durran, J. (1996)

"PC Based Video on Demand Trials", In: *EdTech96*, pp. 21-24

Branch, P., Newstead, A. and Kaushik, R. (1996)

"Design of a Wide Area, Video-On-Demand User Interface", In: *ATNAC 1996*, 1996, pp.

Branch, P. and Tonkin, B. (1997)

"Multicampus Video On Demand at Monash University" *Australian Journal of Educational Technology,* **Vol. 13,** 1997, 85-97.

Brandenburg, K. and Stoll, G. (1994)

"The ISO/MPEG-Audio Codec: A Generic Standard for Coding of High Quality Digital Audio" *Journal of the Audio Engineering Society,* **Vol. 42, No. 10,** 1994, 780-792.

Bridie, J. (1997)

CTIE-TR-1997-06: Systems required to provide on-line Media Delivery Services, CTIE - Monash University, 1997

Bridie, J. and Branch, P. (1998)

CTIE-TR-1998-06: SWIFT and McIVER Integration, CTIE - Monash University, 1998

But, J. (1999a)

CTIE-RR-1999-02: Real Time Encryption/Decryption of an MPEG System Stream, CTIE - Monash University, 1999a

But, J. (1999b)

CTIE-RR-1999-04: Real-Time Decryption of Streaming Video, CTIE - Monash University, 1999b

But, J. and Egan, G. (2002a)

"Designing a Scalable Video On Demand System", In: *International Conference on Communications, Circuits and Systems (ICCCAS'02)*, pp. 559-565

But, J. and Egan, G. (2002b)

"Designing an Affordable Scalable Video On Demand System", In: *2nd ATcrc Telecommunications and Networking Conference and Workshop*, pp. 16-21

Chan, S. and Tobagi, F. (1999)

"Caching schemes for distributed video services", In: *IEEE International Conference on Communications (ICC'99)*, pp.

Chang, I. F. (1998)

"Killer Applications for Internet/Internet2 and their impact to the telecommunication industry", In: *20th Annual Pacific Telecommunications Conference (PTC'98)*, pp. 697-705

Chang, Y.-H., Coggins, D., Pitt, D., Skellern, D., Thapar, M. and Venkatraman, C. (1994)

"An Open-Systems Approach to Video on Demand" *IEEE Communications Magazine,* **Vol. May 1994,** 1994, 68-80.

Chen, H. J., Krishnamurthy, A., Little, T. D. C. and Venkatesh, D. (1995)

"A Scalable Video-on-Demand Service for the Provision of VCR-like Functions", In: *2nd International Conference on Multimedia Computing and Systems*, May 1995, pp. 65-72

Chiariglione, L. (1995)

"MPEG: A Technological Basis for Multimedia Applications" *IEEE Multimedia,* **Vol. Spring 1995,** 1995, 85-89.

Chiariglione, L. (1997)

"MPEG and Multimedia Communications" *IEEE Transactions Circuits and Systems for Video Technology,* **Vol. 7, No. 1,** 1997, 5-18.

Chiariglione, L. (1998)

"Impact of MPEG Standards on Multimedia Industry" *Proceedings of the IEEE,* **Vol.,** 1998.

Cocchi, R., Shenker, S., Estrin, D. and Zhang, L. (1993)

"Pricing in Computer Networks: Motivation, Formulation and Example" *IEEE/ACM Transactions - Network,* **Vol.,** 1993, 614-627.

Cornall, T. (1998)

CTIE-TR-1998-03: Evaluation of Optus Cable Network for 2Mbit/s Real-Time Services, CTIE - Monash University, 1998

Cornall, T. (1999)

CTIE-RR-1999-01: Report on outcomes of DML Stage 2, CTIE - Monash University, 1999

Cornall, T. and Lipton, J. (1997)

CTIE-TR-1997-05: Multimedia Distribution and Networks, CTIE - Monash University, 1997

Cornall, T., Pentland, B. and But, J. (1997)

CTIE-TR-1997-10: Digital Media Library Phase 1, CTIE - Monash University, 1997

Cornall, T., Pentland, B. and Egan, P. G. (1999)

"Digital Media Library Project. Video on demand for schools." In: *International Symposium on Intelligent Multimedia and Distance Education*, August 1999, pp. 59-64

Cruickshank, H., Mertzanis, I., Evans, B. G., Leitold, H. and Posch, R. (1998)

"Securing Multimedia Services over Satellite ATM Networks" *International Journal of Satellite Communications,* **Vol. 16-4, No. July-August,** 1998, 183-195.

deCarmo, L. (2000)

"Security Protocols and Performance" *Dr. Dobbs Journal,* **Vol. November,** 2000, 40-48.

Denning, D. E. (1983)

Cryptography and Data Security, Addison Wesley ISBN 0-201-10150-5.

Diffie, W. and Hellmann, M. E. (1976)

"New Directions in Cryptography" *IEEE Transactions on Information Theory,* Vol. 6, 1976, 644-654.

Egan, P. G. (1998)

CTIE-TR-1998-05: Distributed Multimedia Service Model, CTIE - Monash University, 1998

Fernandes, A. D. (1999)

"Elliptic-Curve Cryptography" *Dr. Dobbs Journal,* **Vol. December,** 1999, 56-63.

Fist, S. (1994)

"Dial M For Movie: Video-on-Demand" *Australian Communications,* **Vol. August 1994,** 1994, 65-72.

Fluhrer, S. R. and McGrew, D. A. (2000)

"Statistical Analysis of the Alleged RC4 Keystream Generator", In: *7th Internation Workshop on Fast Software Encryption*, pp. 19-30

Fowler, D. (1999)

"The next Internet" *Networker,* **Vol. 3, No. 3,** 1999, 20-29.

Frimout, E. D., Biemond, J. and Lagendick, R. L. (1995)

"Extraction of a dedicated fast playback MPEG bit stream" *Proceeding of the SPIE,* Vol. 2501, 1995, 76-87.

Gemmell, J., Vin, H. M., Kandlur, D. D., Rangan, P. V. and Rowe, L. A. (1995)

"Multimedia Storage Servers: A Tutorial" *Computer,* **Vol. May 1995,** 1995, 40-49.

Golic, J. (1997)

"Statistical Analysis of the alleged RC4 Keystream Generator", In: *EUROCRYPT'97*, pp.

Grimm, D. and Cornall, T. (1998)

CTIE-RR-1998-07: Safe Video Delivery for the DML Trial, CTIE - Monash University, 1998

Griwodz, C., Merkel, O., Dittmann, J. and Steinmetz, R. (1998)

"Protecting VoD the Easier Way", In: *ACM Multimedia98*, pp. 21-28

Group, T. L. S. W. (1996)

"The SSL Protocol, Version 3.0 - Internet Draft". http://www.consensus.com/ietf-tls/tls-ssl-version2-00.txt, 1996

Handschuh, H. and Gilbert, H. (1997)

"÷^{2} Cryptanalysis of the SEAL Encryption Algorithm", In: *4th International Workshop on Fast Software Encryption*, pp. 1-12

Haskell, B. G., Puri, A. and Netravali, A. N. (1997)

Digital Video: An introduction to MPEG-2, Chapman & Hall ISBN 0-412-08411-2.

Hsing, T. R., Chen, C.-T. and Bellisio, J. A. (1993)

"Video Communications and Services in the Copper Loop" *IEEE Communications Magazine,* **Vol. January,** 1993, 62-68.

IETF (1998a)

"IP Authentication Header". http://www.ietf.org/internet-drafts/draft-ietf-ipsec-rfc2402bis-05.txt, 1998a

IETF (1998b)

"IP Encapsulating Security Payload (ESP)". http://www.ietf.org/internet-drafts/draft-ietf-ipsec-esp-v3-06.txt, 1998b

IETF (1998c)

"Security Architecture for IP". http://www.ietf.org/internet-drafts/draft-ietf-ipsec-rfc2401bis-01.txt, 1998c

ISO (1996a)

ISO/IEC 11172-1. Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s, Part 1: Systems, ITU, 1996a

ISO (1996b)

ISO/IEC 11172-2. Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s, Part 2: Video, ITU, 1996b

ISO (1996c)

ISO/IEC 11172-3. Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s, Part 3: Audio, ITU, 1996c

ISO (1996d)

ISO/IEC 11172-4. Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s, Part 4: Conformance, ITU, 1996d

Jain, R. (1999)

"The convergence of PCs and TV" *IEEE Multimedia,* **Vol. October/December,** 1999.

Jayanta, K. D., Salehi, J. D., Kurose, J. F. and Towsley, D. (1994)

"Providing VCR Capabilities in Large-Scale Video Server", In: *ACM International Conference on Multimedia*, pp. 25-32

Jung, G. S., Kang, K. W. and Malluhi, Q. (2000)

"Multithreaded Distributed MPEG-1Video Delivery in the Internet Environment", In: *SAC'00*, pp.

Jurišic, A. and Menezes, A. J. (1997)

"Elliptic Curves and Cryptography" *Dr. Dobbs Journal,* **Vol. April,** 1997, 26-36.

Kaliski, B. S. and Robshaw, M. J. B. (1996)

"Multiple Encryption: Weighing Security and Performance" *Dr. Dobbs Journal,* **Vol. January,** 1996, 123-127.

Knudsen, L., Meier, W., Preneel, B., Rijmen, V. and Verdoolaege, S. (1999)

"Analysis Methods for (alleged) RC4", In: *ASIACRYPT'99*, pp. 327-341

Kunkelmann, T. and Horn, U. "Partial Video Encryption Based on Scalable Coding" **Vol.**

Kunkelmann, T. and Horn, U. (1998)

"Video Encryption Based on Data Partitioning and Scalable Coding - A Comparison", In: *Interactive Distributed Multimedia Systems & Telecommunications Services, 5th International Workshop IDMS'98*, 1998, pp. 95-106

Kunkelmann, T. and Reinema, R. (1997)

"A Scalable Security Architecture for Multimedia Communications Standards", In: *IEEE International Conference on Multimedia Computing and Systems 97*, June 1997, pp.

Kunkelmann, T., Reinema, R., Steinmetz, R. and Blecher, T. (1997)

"Evaluation of Different Video Encryption Methods for a Secure Multimedia Conferencing Gateway", In: *4th COST 237 Workshop*, December 1997, pp.

Kunkelmann, T., Vogler, H., Moschgath, M.-L. and Wolf, L. (1998)

"Scalable Security Mechanisms in Transport Systems for Enhanced Multimedia Services", In: *Multimedia Applications - ECMAST'98*, 1998, pp.

Le, N. K. (1998)

CTIE-TR-1998-02: Distributed Server Systems, CTIE - Monash University, 1998

Leditschke, M. and Johnson, A. (1995)

"Implementation of MPEG-2 Trick Modes", In: *Australian Telecommunication Networks & Applications Conference*, December 1995, pp. 39-44

LeGall, D. (1991)

"MPEG: A Video Compression Standard for Multimedia Applications" *Communications of the ACM,* **Vol. 34-4,** 1991, 47-58.

Lin, C.-W., Zhou, J., Youn, J. and Sun, M.-T. (2001)

"MPEG Video Streaming with VCR Functionality" *IEEE Transactions on Circuits and Systems for Video Technology,* Vol. 11, No. 3, 2001, 415-425.

Little, T. D. C. and Venkatesh, D. (1994)

"Prospects for Interactive Video-on-Demand" *IEEE Multimedia,* Vol. 1, No. 3, 1994, 14-24.

Maples, T. B. and Spanos, G. A. (1995)

"Performance Study of a Selective Encryption Scheme for the Security of Networked Real-time Video", In: *4th International Conference on Computer and Communications,* pp.

Masavage, K. J. (1992)

Understanding Digital Video, Optivision, 1992

Masavage, K. J. (1995)

The Types and Requirements of Various MPEG Video Compression Formats, Optivision, 1995

Memon, N. and Wong, P. W. (1998)

"Protecting Digital Media Content" *Communications of the ACM,* Vol. 41, No. 7, 1998, 35-43.

Menezes, A. J., Oorschot, P. C. v. and Vanstone, S. A. (1997)

Handbook of Applied Cryptography, CRC Press ISBN 0-8493-8523-7.

Meyer, C. H. and Matyas, S. M. (1982)

Cryptography: A New Dimension in Computer Data Security, John Wiley & Sons ISBN 0-471-04892-5.

Meyer, J. and Gadegast, F. (1995)

Security Mechanisms for Multimedia with Example MPEG-1 Video, Tech. Uni. of Berlin, 1995

Microsoft (2001a)

DirectX 9 SDK Documentation, Microsoft, 2001a

Microsoft (2001b)

"DirectX 9 Software Development Kit". http://www.microsoft.com/downloads/details.aspx?FamilyId=124552FF-8363-47FD-8F3B-36C226E04C85&displaylang=en, 2001b

Microsoft (2002)

"Microsoft Windows Media". http://www.microsoft.com/windows/windowsmedia, 2002

Middleton-Williams, C. (1993)

"Digital Video Technology" *Journal of Electrical and Electronics Engineering, Australia,* Vol. 13, No. 3, 1993, 205-212.

Mister, S. and Tavares, S. E. (1998a)

"Cryptanalysis of RC4-like Ciphers", In: *Workshop on Selected Areas in Cryptography,* pp. 131-143

Mister, S. and Tavares, S. E. (1998b)

"Some Results on the cryptanalysis of RC4", In: *Proceedings of the 19th Bienniel Symposium on Communications*, pp. 393-397

Mitchell, J. L., Pennebaker, W. B., Fogg, C. E. and LeGall, D. J. (1996)

MPEG Video Compression Standard, Chapman & Hall ISBN 0-412-08771-5.

Mohammed, A. (2002)

"DiffServ experiments: Evaluation of some approaches to quality of service control over the Alcatel-NCSU Internet2 testbed" *Proceedings of the SPIE - The International Society of Optical Engineering,* Vol. 4866, 2002, 23-34.

Nelson, R. (1998)

CTIE-TR-1998-16: McIVER Network Performance Requirements, CTIE - Monash University, 1998

Nguyen, J. (1995)

Hardware and Software MPEG: A White Paper, Sigma Designs, Inc., 1995

NIST (1993a)

Data Encryption Standard, FIPS Publication 46, NIST, 1993a

NIST (1993b)

Secure Hash Standard, FIPS Publication 180, NIST, 1993b

Noll, P. (1997)

"MPEG Digital Audio Coding" *IEEE Signal Processing Magazine,* Vol. 14-5, 1997, 59-81.

Noll, P. and Pan, D. (1997)

"ISO/MPEG Audio Coding" *International Journal of High Speed Electronics and Systems,* Vol. 8, No. 1, 1997, 69-118.

Noronha, C. (2001)

Stream Cipher Cryptanalysis, Final Year Undergraduate Research Thesis - Monash University, 2001

Pan, D. (1993)

"Digital Audio Compression" *Digital Technical Journal,* Vol. 5, No. 2, 1993.

Pan, D. (1995)

"A Tutorial on MPEG/Audio Compression" *IEEE Multimedia,* Vol. Summer 1995, 1995, 60-74.

Patrick, S. and Moccio, D. (1998)

Writing a WebFORCE ® MediaBase Player, SGI - Document 007-3569-003, 1998

Pentland, B. (1999)

CTIE-TR-1999-03: Minimum Network Requirements for DML Video Streaming, CTIE - Monash University, 1999

Pereira, F. (1996)

"MPEG4: A New Challenge for the Representation of Audio-Visual Information", In: *International Picture Coding Symposium,* March 1996, pp. 7-16

Preneel, B., Rijmen, V. and Bosselaers, A. (1998)

"Priciples and Performance of Cryptographic Algorithms" *Dr. Dobbs Journal,* Vol. December, 1998, 126-131.

Puri, A. (1994)

"Video Coding Using the MPEG-2 Compression Standard" *SPIE,* Vol. 2049, 1994, 1701-1713.

Qiao, L. and Nahrstedt, K. (1996)

"Comparison of MPEG Encryption Algorithms" *Computers and Graphics,* Vol. 22, 1996, 437-448.

Qiao, L. and Nahrstedt, K. (1997)

"A New Algorithm for MPEG Video Encryption", In: *1st International Conference on Imaging Science, Systems and Technology,* pp.

Qiao, L., Nahrstedt, K. and Tam, M.-C. (1997)

"Is MPEG Encryption by using Random List instead of Zig-Zag order secure?" In: *IEEE International Symposium on Consumer Electronics,* pp.

Ramarao, R. and Ramamoorthy, V. (1991)

"Architectural Design of On-Demand Video Delivery Systems: The Spatio-Temporal Storage Allocation Problem", In: *IEEE International Conference on Communications,* pp.

Rangan, P. V., Vin, H. M. and Ramanathan, S. (1992)

"Designing an on-demand multimedia service" *IEEE Communications Magazine,* **Vol. July,** 1992, 56-64.

Reader, C. (1996)

"MPEG4: Coding for content, interactivity, and universal accessibility" *Optical Engineering,* **Vol. 35-1, No. 1,** 1996, 104-108.

Rivest, R. L., Shamir, A. and Adleman, L. (1978)

"A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" *Communications of the ACM,* **Vol. 21,** 1978, 120-126.

Rogaway, P. and Coppersmith, D. (1993)

"A Software-Optimised Encryption Algorithm", In: *Cambridge Security Workshop - Fast Software Encryption*, pp. 56-63

Rogaway, P. and Coppersmith, D. (1998)

"A Software-Optimised Encryption Algorithm" *Journal of Cryptography,* **Vol. 11, No. 4,** 1998, 273-287.

RSA (1996)

Answers to Frequently Asked Questions About Today's Cryptography, RSA Laboratories, 1996

Saunders-McMaster, L. (1997)

"Internet2: an overview of the next generation of the Internet" *Computers in Libraries,* **Vol. 17, No. 3,** 1997, 57-59.

Schneier, B. (1996a)

Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley & Sons ISBN 0-471-11709-9.

Schneier, B. (1996b)

"Differential and Linear Cryptanalysis" *Dr. Dobbs Journal,* **Vol. January,** 1996b, 42-48.

Schneier, B. (1998)

"The Twofish Encryption Algorithm" *Dr. Dobbs Journal,* **Vol. Decembet,** 1998, 30-38.

Shanableh, T. and Ghanbari, M. (2001)

"The Importance of Bi-Directionally Predicted Pictures in Video Streaming" *IEEE Transactions on Circuits and Systems for Video Technology,* **Vol. 11, No. 3,** 2001, 402-414.

Shi, C. and Bhargava, B. (1998a)

"An Efficient MPEG Video Encryption Algorithm", In: *17th IEEE Symposium on Reliable Distributed Systems*, October 1998, pp. 381-386

Shi, C. and Bhargava, B. (1998b)

"A Fast MPEG Video Encryption Algorithm", In: *ACM Multimedia '98*, 1998, pp. 81-88

Shi, C. and Bhargava, B. (1998c)

"Light-weight MPEG Video Encryption Algorithm", In: *Multimedia98*, pp. 55-61

Shlien, S. (1994)

"Guide to MPEG-1 Audio Standard" *IEEE Transactions on Broadcasting,* **Vol. 40, No. 4,** 1994, 206-218.

Sikora, J. J. (2001)

"QoS in Internet2" *Proceedings of the SPIE - The International Society of Optical Engineering,* **Vol. 4211,** 2001, 122-130.

Sikora, T. (1997)

"MPEG Digital Video-Coding Standards" *IEEE Signal Processing Magazine,* **Vol. 14-5,** 1997, 82-100.

Simmons, G. J. (1992)

Contemporary Cryptology: The Science of Information Integrity, IEEE Press ISBN B-87942-277-7.

Spanos, G. A. and Maples, T. B. (1996)

"Security for Real-Time MPEG Compressed Video in Distributed Multimedia Applications", In: *IEEE 15th Annual International Conference on Computers and Communications*, pp. 72-78

Stinson, D. R. (1995)

Cryptography: Theory and Practice, CRC Press ISBN 0-84938-521-0.

Tang, L. (1996)

"Methods for Encrypting and Decrypting MPEG Video Data Efficiently", In: *ACM International Multimedia Conference 96*, November 1996, pp. 219-222

Teitelbaum, B., Hares, S., Dunn, L., Neilson, R., Narayan, V. and Reichmeyer, F. (1999)

"Internet2 QBone: building a testbed for differentiated services" *IEEE Network,* Vol. 13, No. 5, 1999, 8-16.

Tonkin, D. B. (1998)

CTIE-TR-1998-17: Approaches to Video Distribution, CTIE - Monash University, 1998

Tosun, A. S. and Feng, W.-c. (2000)

"Efficient Multi-layer Coding and Encryption of MPEG Video Streams", In: pp. 119-122

Tosun, A. S. and Feng, W.-c. (2001)

"A Light-weight Mechanism for Securing Multi-Layer Video Streams", In: *IEEE International Conference on Information Technology: Coding and Computing*, pp. 157-161

Unknown (1999)

"MPEG Test Bitstreams (tennis.mpg, flowg.mpg, us.mpg)". http://peipa.essex.ac.uk/ipa/src/formats/mpeg/stanford, 1999

Viña, A., Lérida, J. L., Molano, A. and delVal, D. (1994)

"Real-Time Multimedia Systems", In: *13th IEEE Symposium on Mass Storage Systems,* June 1994, pp. 77-83

Wallace, G. K. (1991)

"JPEG: A Digital Image Compression Standard" *Communications of the ACM,* Vol. 34-4, 1991, 31-44.

Wang, S., Mai, Z., Magnussen, W., Xuan, D. and Zhao, W. (2002)

"Implementation of QoS-Provisioning system for voice over IP", In: *Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 266-275

Wu, D., Hou, Y. T., Zhu, W., Zhang, Y.-Q. and Peha, J. M. (2001)

"Streaming Video over the Internet: Approaches and Directions" *IEEE Transactions on Circuits and Systems for Video Technology,* Vol. 11, No. 3, 2001.

Wu, M.-Y., Ma, S.-J. and Shu, W. (2002)

"Scheduled Video Delivery for Scalable On-Demand Service", In: *NOSSDAV '02*, pp.

Zeng, W. and Lei, S. (1999)

"Efficient Frequency Domain Video Scrambling for Content Access Control", In: *ACM Multimedia99*, pp. 285-294

Zeng, W., Wen, J. and Severa, M. (2002)

"Fast Self-Synchronous Content Scrambling by Spatially Shuffling Codewords of Compressed Bitstreams", In: pp. 169-172