Dynamic Block Encryption with Self-Authenticating Key Exchange

by

Nasser Al-Ismaily

Doctoral Thesis

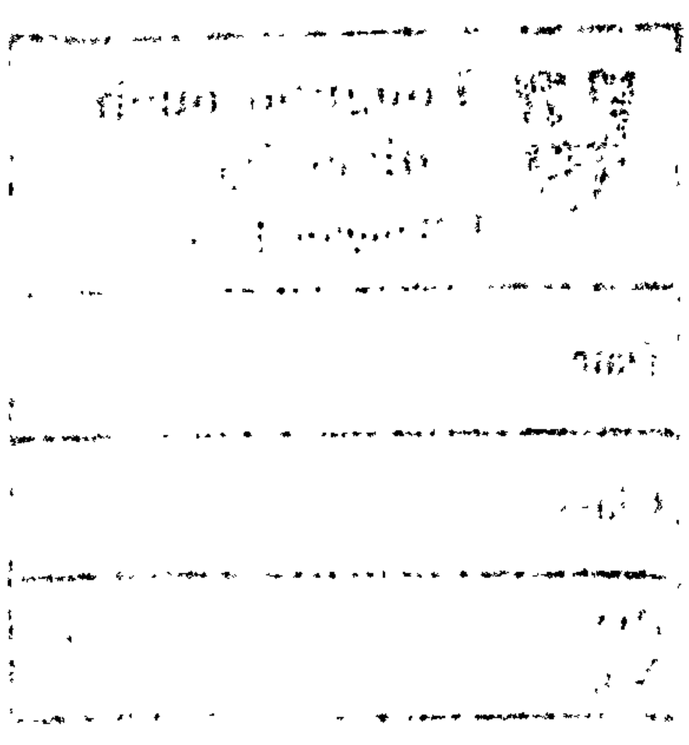
Submitted in partial fulfillment of the requirements

for the award of PhD

Department of Computer Science

Loughborough University

November 23, 2005



'The First World War was the chemists' war, because the mustard gas and chlorine were employed for the first time. The Second World War was the physicists' war, because the atom bomb was detonated. The Third World War will be the mathematicians war, because mathematicians will have control over the next great weapon of war - "information"'

Simon Singh

Abstract

One of the greatest challenges facing cryptographers is the mechanism used for key exchange. When secret data is transmitted, the chances are that there may be an attacker who will try to intercept and decrypt the message. Having done so, he/she might just gain advantage over the information obtained, or attempt to tamper with the message, and thus, misguiding the recipient. Both cases are equally fatal and may cause great harm as a consequence.

In cryptography, there are two commonly used methods of exchanging secret keys between parties. In the first method, symmetric cryptography, the key is sent in advance, over some secure channel, which only the intended recipient can read. The second method of key sharing is by using a public key exchange method, where each party has a private and public key, a public key is shared and a private key is kept locally. In both cases, keys are exchanged between two parties.

In this thesis, we propose a method whereby the risk of exchanging keys is minimised. The key is embedded in the encrypted text using a process that we call 'chirp coding', and recovered by the recipient using a process that is based on correlation. The 'chirp coding parameters' are exchanged between users by employing a USB flash memory retained by each user. If the keys are compromised they are still not usable because an attacker can only have access to part of the key. Alternatively, the software can be configured to operate in a one time parameter mode, in this mode, the parameters are agreed upon in advance. There is no parameter exchange during file transmission, except, of course, the key embedded in ciphertext.

The thesis also introduces a method of encryption which utilises dynamic

blocks, where the block size is different for each block. Prime numbers are used to drive two random number generators: a Linear Congruential Generator (LCG) which takes in the seed and initialises the system and a Blum-Blum Shum (BBS) generator which is used to generate random streams to encrypt messages, images or video clips for example. In each case, the key created is text dependent and therefore will change as each message is sent.

The scheme presented in this research is composed of five basic modules. The first module is the *key generation module*, where the key to be generated is message dependent. The second module, *encryption module*, performs data encryption. The third module, *key exchange module*, embeds the key into the encrypted text. Once this is done, the message is transmitted and the recipient uses the *key extraction module* to retrieve the key and finally the *decryption module* is executed to decrypt the message and authenticate it. In addition, the message may be compressed before encryption and decompressed by the recipient after decryption using standard compression tools.

Acknowledgments

I thank God for giving me the ability, health and knowledge to go through this work.

I would also like to thank the government of Oman for providing the funds for me to undertake this research program.

I have been lucky to work with my supervisor, Ana Salagean, who offered me valuable assistance on many aspects of research work. From her assistance, inspiration, and positive criticism I have learned a lot about doing research and presenting the work accordingly. Many thanks to Jonathan Blackledge, for spending a lot of his precious time offering me guidance and supervision on my work. I also thank Sekharjit Datta for his involvement with the program. Thanks to Mohamed Jaffar from De Montfort University for his guidance and supervision during the first year of this program.

I am grateful to my whole family for their confidence in my work, encouragement and unconditional support.

Glossary of Terms

ANSI	American National Standards Institute
BBS	Blum Blum Shub.
CBC	Cipher Block Chaining Mode
CFB	Cipher Feedback Block Mode
DBX	Dynamic Block Encryption
ECB	Electronic Code Book Mode
FEAL	Fast Data Encipherment Algorithm
GIMPS	Great Internet Mersenne Prime Search.
GOST	Russian cryptographic algorithm similar to DES in many ways
GPS	Global Positioning System
IDEA	International Data Encryption Algorithm
IFS	Iteration Function System
ISO	International Organisation for Standardization
IV	Initialisation vector
LCG	Linear Congruential Generator
MD5	Message Digest
NSA	National Security Agency
OFB	Output Feedback Block Mode
OTP	One Time Pad
PKI	Public Key Infrastructure
PRNG	Pseudo Random Number Generator
SEAL	Software-Optimised Encryption Algorithm
SHA	Secure Hash Algorithm

Glossary (continued)

Asymmetric Encryption

A form of cryptosystem in which encryption and decryption are performed using two different keys, one of which is referred to as the public key. Also known as public key encryption.

Authentication

A process used to verify the integrity of transmitted data, especially a message.

Block Cipher

A symmetric encryption algorithm in which a large block of plaintext bits (typically 64) is transformed as a whole into a ciphertext block of the same length.

Cipher

An algorithm for encryption and decryption. A cipher replaces a piece of information (an element in plaintext) with another object, with the intent to conceal the meaning. Typically, the replacement rule is governed by a secret key.

Ciphertext

The output of an encryption algorithm; the encrypted form of a message data.

Code

An unvarying rule for replacing a piece of information (e.g., letter, word, phrase) with another object, not necessarily of the same sort. Generally, there is no intent to conceal meaning. Examples include the ASCII character code (each character is represented by 7 bits) and frequency-shift keying (each binary value is represented by a particular frequency).

Confusion

A cryptographic technique that seeks to make the relationship between the statistics of the ciphertext and the value of the encryption key as complex as possible. This is achieved by the use of a complex scrambling algorithm that depends on the key and the input.

Cryptanalysis

The branch of cryptology dealing with the breaking of a cipher to recover information, or forging encrypted information that will be accepted as authentic.

Cryptography

The branch of cryptology dealing with the design of algorithms for encryption and decryption, intended to ensure the secrecy and authenticity of messages.

Cryptology

The study of secure communications, which encompasses both cryptography and cryptanalysis.

Decryption

The translation of encrypted text or data (called ciphertext) into original text or data (called plaintext). Also called deciphering.

Differential Cryptanalysis

A technique in which chosen plaintexts with particular XOR difference patterns are encrypted. The difference patterns of the resulting ciphertext provide information that can be used to determine the encryption key.

Diffusion

A cryptographic technique that seeks to obscure the statistical structure of the plaintext by spreading out the influence of each individual plaintext digit over many ciphertext digits.

Digital Signature

An authentication mechanism that enables the creator of a message to attach a code which acts as a signature. The signature guarantees the source and integrity of the message.

Encryption

The conversion of plaintext or data into unintelligible form by means of a reversible translation, based on a translation table or algorithm. Also called enciphering.

Hash Function

A function that maps a variable length data block or message into a fixed length value called a hash code. The function is designed in such a way that, when protected, it provides an authenticator to the data or message. Also referred to as a message digest.

Initialisation Vector

A random block of data that is used to begin the encryption of multiple blocks of plaintext, when a block-chaining encryption technique is used. The IV serves to foil known-plaintext attacks.

Message Digest

Hash function.

One-Way Function

A function that is easily computed, but the calculation of its inverse is infeasible.

Plaintext

The input to an encryption function or the output to a decryption function.

Private Key

One of the two keys used in an asymmetric encryption system. For secure

communication, the private key should only be known to its creator.

Pseudorandom Number Generator

A function that deterministically produces a sequence of numbers that are apparently statistically random.

Public Key

One of the two keys used in an asymmetric encryption system. The public key is made public, to be used in conjunction with a corresponding private key.

Secret Key

The key is used in a symmetric encryption system. Both participants must share the same key, this key must remain secret to protect the communication.

Skipjack

Secure encryption algorithm designed by NASA

Stream Cipher

A symmetric encryption algorithm in which ciphertext output is produced bit-by-bit or byte-by-byte from a stream of plaintext input.

Symmetric Encryption

A form of cryptosystem in which encryption and decryption are performed using the same key. Also known as conventional encryption.

Contents

Chapter 1

Introduction

The electronic age has managed to bring a number of changes in the way individuals conduct their daily routines. One of the most significant of these changes is the impact it has had upon basic human activities such as decision making, information processing and communication. Business communities and government organisations rely heavily on exchange, sharing, and processing of information to assist them in making everyday and strategic decisions. Security infrastructures have been put in place to help protect and preserve integrity of the information flowing across different channels. It is therefore necessary to provide continuous improvements to the security infrastructure in order to keep up with the fast pace of technology growth in areas of digital communication and software development.

As the world becomes more dependent on digital information exchange, this, in turn, threatens the security of the information itself. Information is the key factor in decision making for most organisations today and so has become the one of the most important assets that the company owns [92]. Most of

the data transported between different locations and recipients is in danger of being viewed and/or altered by any capable and interested eavesdropper. For example, the use of digital devices has touched every aspect of our lives. By using a personal digital assistant (PDA), for example, an individual can log on and check his/her bank account, make funds transfer, pay bills, or undertake other transactions such as trading on stock markets. Scientists rely heavily on computers to get results from different sources around the world; in 2003 astronomers were busy trying to track down signals from Beagle 2 on Mars. [33]

Since the introduction of the global positioning system (GPS), an increasing number of private motorists are relying on the use of GPS, and navigation in general, as part of their guide to operating in unknown regions. Previously, GPS was enjoyed by the civil and military services, but now it is at the disposal of most civilians. In the event of a major terrorist incident, or other planned 'breakins', there is the potential for a major disaster affecting us all through a breach of information interchange.

In August 11, 2003, Jeffrey Parson, an 18 year old high school student was suspected of unleashing a deadly internet 'worm'[1]. know as the 'MS Blaster'. The alleged worm operated on the weakness of Microsoft Windows operating system, and is said to have infected over 500,000 personal computers across the globe [54], [23].

This example is not an isolated incident, as there has been similar, although

[1]A worm is a hidden file that is typically imported into a computer when accessing information over the internet. It is a program that makes copies of itself (e.g. from one disk drive to another, or by copying itself using e-mail or another transport mechanism) and can provide a number of facilities for accessing the computer remotely

uncoordinated attacks, against PCs though the internet. However, this example illustrates the extent of damage that can be caused if one were to penetrate the security built against control centers for space research, global positioning satellites, or communication channels between financial institutions, as well as those with clients. Thus, it is becoming more and more important to secure the infrastructure of an increasingly, information dependent society.

Cryptography has been playing an important role in the IT world for securing and protecting data. Along with cryptography, digital watermarking is starting to be used in many applications to authenticate objects. There is an argument that watermarking has taken over what cryptography is missing [34]. This is because an encrypted file gives away the fact that there is information that is important and is thus a 'red rag to a bull' for the potential interceptor. Watermarking can provide a way of transmitting information in data that is seemingly insignificant because it does appear in an encrypted form. In this thesis, it is shown that both cryptography and digital watermarking can be used to protect data in a way that is mutually inclusive. Cryptography and watermarking used separately cannot guarantee security (some examples of the reasons on *Why Cryptography Fails* are demonstrated by Ross Anderson [2]) but, used together, they can enhance the security of a communications infrastructure.

1.1 Background

Cryptography is the science of writing messages that no one, except the intended recipient, is able to read. Cryptanalysis [28] deals with the way of

trying to break the messages and read them. 'Crypto' is from the Greek word 'Krypte' meaning hidden or vault and 'Graphy' is also from the Greek 'Grafik' which means to write [60]. William F. Friedman defines a cipher message as one produced by applying a method of cryptography to the individual letters of the plain text taken as either single entities or in groups of constant length. Practically, every cipher message is the result of the joint application of a 'general system' (or algorithm) or method of treatment, which is invariable and a specific key which is variable, at the will of the correspondents, and controls the exact steps followed under the 'general system'. It is assumed that the general system is known by the correspondents and the cryptanalyst.

Different cryptographic techniques have been developed and employed to protect information. Most of them make use of algorithms employing public key exchange protocol which may been broken either by exploiting the weakness in the key exchange mechanisms or through the algorithm itself. There are a number of ways for checking the cryptographic security of an object (e.g. data stream). Analysing the algorithm, or looking at the mathematical model, can assist in revealing the strength or weakness of an object. One can also try known attacks to determine its strength. However, by applying various known attacks, without success, does not mean that the object is secure. This is because most attacks are relatively comparable to 'laboratory' experiments', and hence they differ form real world attacks. However, this can, in theory, be considered as a first step towards stronger resistance to attack. According to Daemen [25] '... cryptographic security of a cipher can best be defined as security in the worst possible circumstances. Clearly, a cipher that is claimed to be cryptographically secure by this definition is claimed to be secure in all applications'

Steganography is another form of cryptography. This goes back thousands of years. For example, during the war, in order to conceal the messages, Histaiaues shaved the head of his messenger, wrote the message on his scalp, then waited for the hair to regrow, once the hair has grown, he would send the messenger, whom himself was not aware of the message he carried across the enemy lines. (War then was a little slower than now!) Steganography is the practice of embedding secret messages in other messages in a way that prevents an observer from learning that anything unusual is taking place. Encryption, by contrast, relies on ciphers or codes to scramble a message.

The practice of steganography has a distinguished history. The Greek historian Herodotus also used steganography and describes how one of his cunning countrymen sent a secret message warning of an invasion by scrawling it on the wood underneath a wax tablet. To casual observers, the tablet appeared blank. Both Axis and Allied spies during World War II used such measures as invisible inks [22]; for example, using milk, fruit juice or urine which darken when heated, or tiny punctures above key characters in a document that form a message when combined.

Modern steganographers have far-more-powerful tools. Software like White Noise Storm [46] and S-Tools allow a user to embed messages in digitized information; typically, audio, video or still image files, that are sent to a recipient. The software usually works by storing information in the least significant bits of a digitized file; those bits can be changed in ways that are not dramatic enough for a human eye or ear to detect. It is relatively simple, for example, to insert a message in the least significant bits of an image JPEG file that, when viewed, have no substantial differences to the original JPEG images. *Steghide* embeds messages in .bmp, .wav and .au files

via least significant bits and *MP3Stego* does the same for MP3 files. One program, called *snow*, hides a message by adding extra whitespace at the end of each line of a text file or e-mail message. Perhaps the strangest example of steganography is a program called *Spam Mimic*, based on a set of rules, called a mimic engine, and designed by Wayner [96]. It encodes a message into what looks just like a typical, quickly deleted spam message.

Other methods of securing objects are by watermarking. Although encrypting an object offers security, a culprit can buy a legal copy of an object, say music or a movie, and then start distributing the copies illegally. The method of tracking down the original buyer can be quite strenuous. By applying watermarking to an object, whenever an illegal copy is found, it can then be verified against the original.

The idea of watermarking can be dated back to the late Middle Ages. The earliest use has been to record the manufacture's trademark on the product so that authenticity can be easily established. Governments use it for currencies, postage stamps, revenue stamps, etc. [6]. Now, due to the information and computer age, digital watermarking is being rapidly expanded to cover a wide range of applications.

Digital watermarking is a process of embedding unobtrusive marks or labels into digital content. These embedded marks are typically invisible and can later be detected or extracted. The concept of digital watermarking is closely association with steganography. Watermarks added to digital content serve a variety of purposes:

- Ownership Assertion - to establish ownership of the content (i.e. image).

- Fingerprinting - to avoid unauthorized duplication and distribution of publicly available multimedia content.

- Authentication and integrity verification - the authenticator is inseparably bound to the content whereby the author has a unique key associated with the content and can verify integrity of that content by extracting the watermark.

- Content labeling - bits embedded into the data that give further information about the content such as a graphic image with time and place information.

- Usage control - added to limit the number of copies created where the watermarks are modified by the hardware and at some point do not allow further copies to be made (e.g. a DVD).

- Content protection - content stamped with a visible watermark that is very difficult to remove so that it can be publically and freely distributed.

Unfortunately, there is no universal watermarking technique to satisfy all of the above purposes. The content in the environment that is used determines the watermarking technique.

1.2 Prime Numbers and Cryptography

In the general field of Cryptology and, in particular, cryptography, prime numbers have emerged to play a central role, especially since the development of the programmable computer in Bletchley Park, England, in 1944.

Prime numbers and their properties were first studied deeply by ancient Greek mathematicians. The mathematicians of the Pythagorian school (500 BC to 300 BC) were interested in prime numbers for their mystical and numerological properties. They understood the idea of primality and were interested in perfect and amicable numbers. Even though they have been studied for many years, the full potential of prime numbers, has only relatively recently been realised. Prime numbers have become important when used within one way functions in encryption algorithms, in particular the design of one-way functions that exploit modular arithmetic. Proper studies and analysis of prime numbers can result in implementation of stronger encryption software. In general, one-way functions are based on exploiting the joint properties of prime numbers and modular arithmetic and the large majority of encryption algorithms have come to be based on exploiting the interplay between these two 'elements'.

One of most extensive studies in the field of number theory is knowledge of exactly how many prime numbers are there. Are prime numbers finite or infinite? [44] [67] Ever since the study of prime numbers began, a large number of mathematicians have developed different theories in finding the primes. Euclid's Elements appeared around 300 BC and by this time several important results about primes had been proved. In Book IX [32] of the Elements, Euclid proves that there are infinitely many prime numbers. This is one of the first proofs known which uses the method of contradiction to establish a result. Euclid also gives a proof of the Fundamental Theorem of Arithmetic: Every integer can be written as a product of primes in an essentially unique way. Euclid also showed that if the number $2n - 1$ is prime then the number $(2n - 1)(2n - 1)$ is a perfect number. In 1747, Euler was able to show that all even perfect numbers are of this form. It is not

known to this day whether there are any odd perfect numbers.

In about 200 BC, the Eratosthenes devised an algorithm for calculating primes called the Sieve of Eratosthenes. The next important developments were made by Fermat at the beginning of the 17th Century. He proved a speculation of Albert Girard that every prime number of the form $4n + 1$ can be written in a unique way as the sum of two squares and was able to show how any number could be written as a sum of four squares. He proved what has come to be known as Fermat's Little Theorem (to distinguish it from his so-called Last Theorem). This states that if p is prime then for any integer a we have $ap = a \bmod p$. Fermat's little theorem is actually the fundamental basis for asymmetric encryption, e.g. encryption systems that use public and private keys. Originally derived by GCHQ, Cheltenham, England in the early 1970s, it was first marketed in the USA in the late 1970s as the RSA algorithm. Today, the RSA algorithm, which is itself, essentially a by-product of Fermat's little theorem, now forms the kernel of a wide rage of encryption systems application, e.g. all PKI (Public Key Infrastructure) systems. The range of applications is also widespread and apart from being used in encryption in general, is used in biometrics such as in finger-print [55] and iris [39] recognition in which access to databases is acquired through PKI.

The importance of prime numbers in encryption has meant that one of the principal applications for research into prime numbers and number theory in general is cryptology. This includes the design of new algorithms that are applied, in conjunction with powerful computing technology, in the computation of new prime numbers.

1.3 Research Focus

Almost all block cipher algorithms use fixed length blocks. If an adversary successfully manages to break one block, the chances are that the rest of the blocks can be cracked by following a pattern. Hence information is compromised.

One way to improve the security of the cipher is to use variable size block encryption with some form of parameter modulation. Dynamic Block Encryption (DBX) uses variable size blocks. This makes it hard to crack because the block size is randomly selected, hence there is no way of knowing which block the data belongs to. Further each block is driven by what is in effect a variation on the theme of the same algorithm or alternative a uniquely different algorithm (multi-algorithmicity).

The other contribution to this thesis is the key exchange mechanism. In symmetric ciphers when encrypted data is transmitted, the key has to be sent separately. One way to send the key is by using a secure channel shared between the sender and the recipient. This key is then used repeatedly until users decide to change it. This poses an increased risk because the same key is used for long periods. Once compromised, the data transmitted is no longer secure.

The asymmetric cipher uses public key exchange where the user publishes his/her key on the Internet and the sender encrypts the file using the recipient's public key. The recipient decrypts using his own private key. One risk involves exposing the key. Hence a cryptanalyst already acquires a starting point.

DBX key exchange mechanism is designed to overcome both issues. The key

is chirp coded and then embedded in the ciphertext before transmission. This overcomes the problem of transmitting the key. The key changes every time the file is transmitted so that, if a key is compromised, it is rendered useless because the next key is entirely different. The key is transmitted within the ciphertext so that there is no problem in transmitting and exchanging keys.

1.4 Original Contribution

The original contributions of this thesis can be summerised as follows:

- **Dynamic blocks cipher with prime number modulation:** The implementation of dynamic blocks in which each block output is the result of utilizing a randomly selected prime number from a pre-determined database to drive a BBS cipher generator. (Section 5.4 and 6.3)

- **Key exchange mechanism:** A secure method of exchanging keys between two or more parties that is based on the use of chirp coding to embed the key used into the cipher text that it (the key) generates using the block cipher technique above. This method can be applied either by using fixed parameters or using a '*crypstic*'. (Section 5.3 and 6.2)

Publications

The following papers have been published based on the above contributions:

- Al-Ismaily, N., Salagean, A., Blackledge, J. and Datta, S., 'Digital Watermarking Encryption and Authentication' , Proc. of EPSRC PREP

2004 , EPSRC, Fourth Conference on Postgraduate Research in Electronics Photonics Communications and Software (PREP 2004), April 2004, 189-190, ISBN 1 899371 33 8 .

- Al-Ismaily, N., Salagean, A., Blackledge, J.M. and Datta, S., 'Encryption using Varying Block Length with Embedded Key Exchange Mechanism', Proceeedings of EPSRC PREP 2005, University of Lancaster, April 2005, pp 75-76 .

Both papers have been included in the accompanying CD.

1.5 About this Thesis

We derive an algorithm that utilizes dynamic length block cipher in which the block length changes randomly using a database of over one million primes. The initial seed is generated from a key which has been derived from the plaintext. The seed is then used to 'drive' a linear congruential generator which selects two primes. These primes are then used to 'drive' the Blum-Blum Shub (BBS) [10] generator to compute variable block lengths which vary from 5 to 50 characters in length. The block diagram given in Figures 1.1 and 1.2 outline the process.

Once a block length is selected, each character in a block is then XOR'ed with a random number picked from the second run of BBS generator. The key for

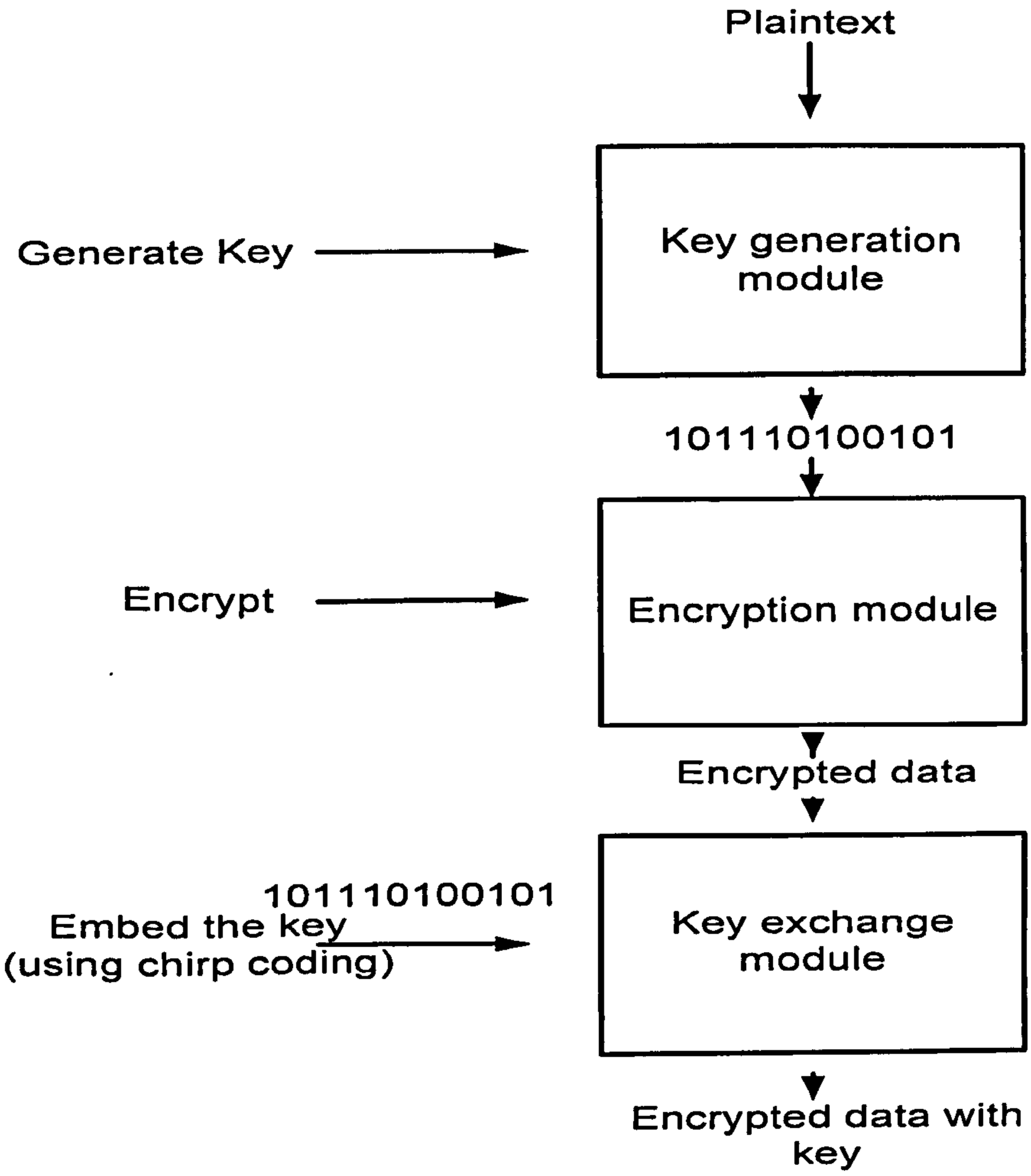
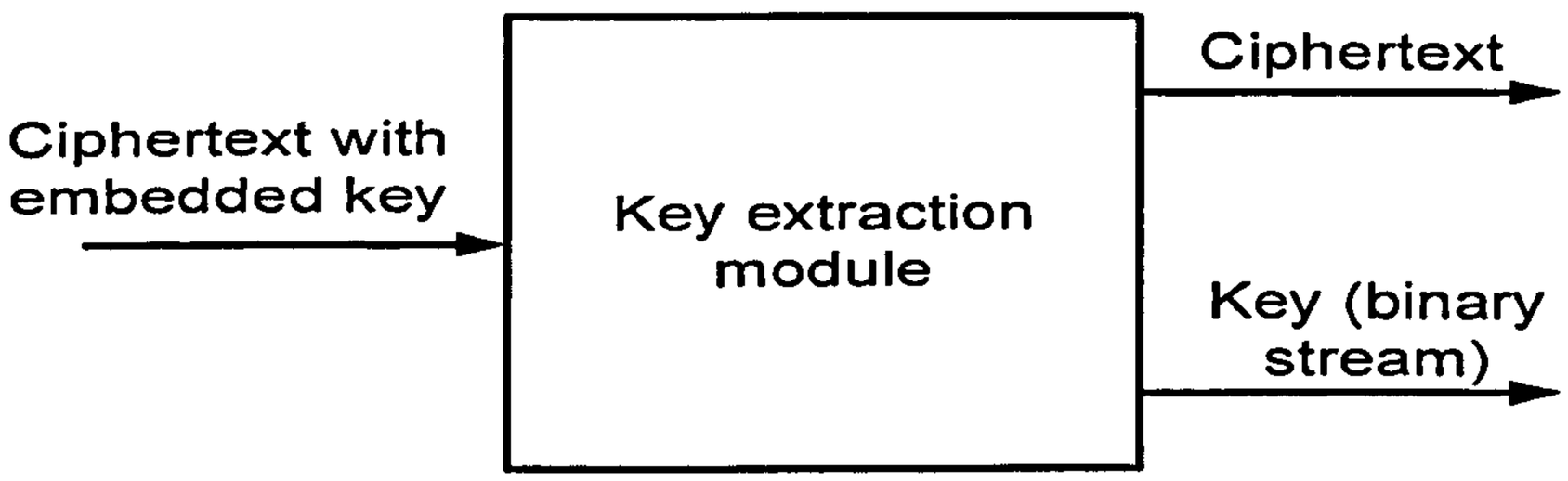


Figure 1.1: Encryption process with key embedding techniques



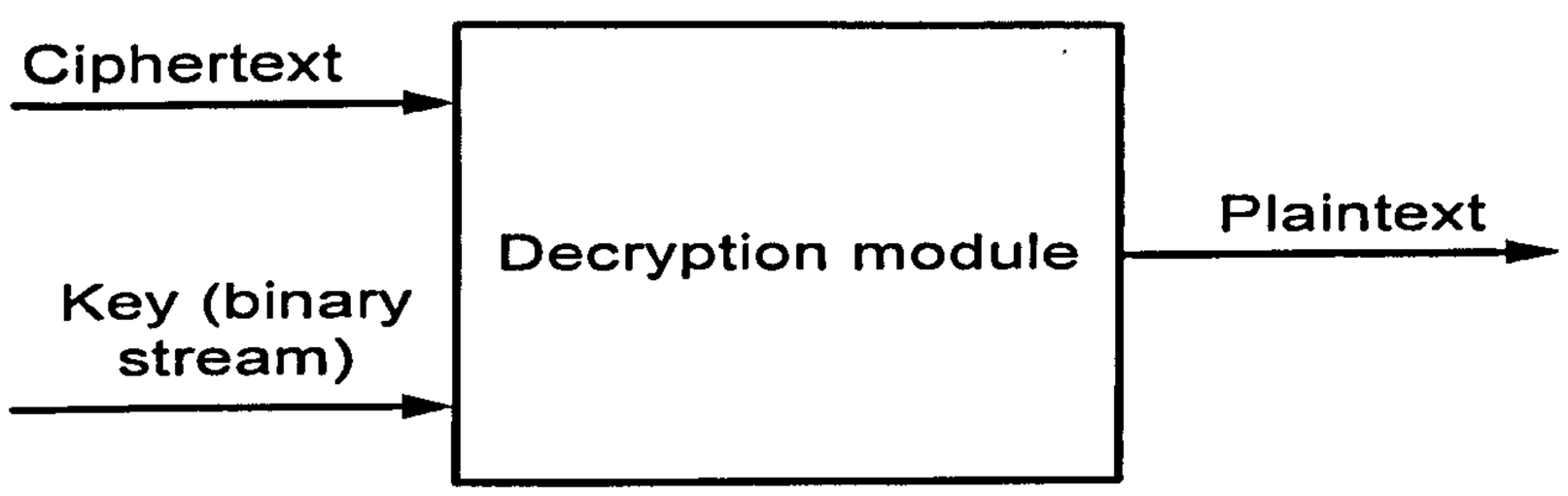


Figure 1.2: Decryption process

encryption is derived from the plaintext using one of four transforms including wavelet decompostion [64]. This provides a unique (plain text dependent) bit stream which is applied as a binary key. Chapter 2 and Chapter 3 provide essential background material (including a literature search - Chapter 2) upon which this research has been based.

In Chapter 4, we consider a method of watermarking the ciphertext with the binary key obtained. The method is based on application of the chirp function to produce a chirp stream, i.e. applying a linear frequency modulated waveform (of finite length) to represent a 0 or 1 (phase reversed). This method is then combined with the encryption engine (the prime number modulation and dynamic block cipher) that is the subject of Chapter 5. Test results based on m-code developed for this thesis are discussed in Chapter 6 which details those features of the computational procedures that are fundamental to this work and presents the principal functions and objects used to design and construct the encryption system developed. Figure 1.2 shows a basic encryption process.

All software development work undertaken for this research is provided in Appendix A. All prototyping work was undertaken using MathWorks Inc MATLAB Version 6, in particular, in implementing the prime number modulation and dynamic block cipher encryption engine with auto-authenticating key exchange. The m-code is given in Appendix A. In Appendix B we have included background on crypstic and its application on commercial world.

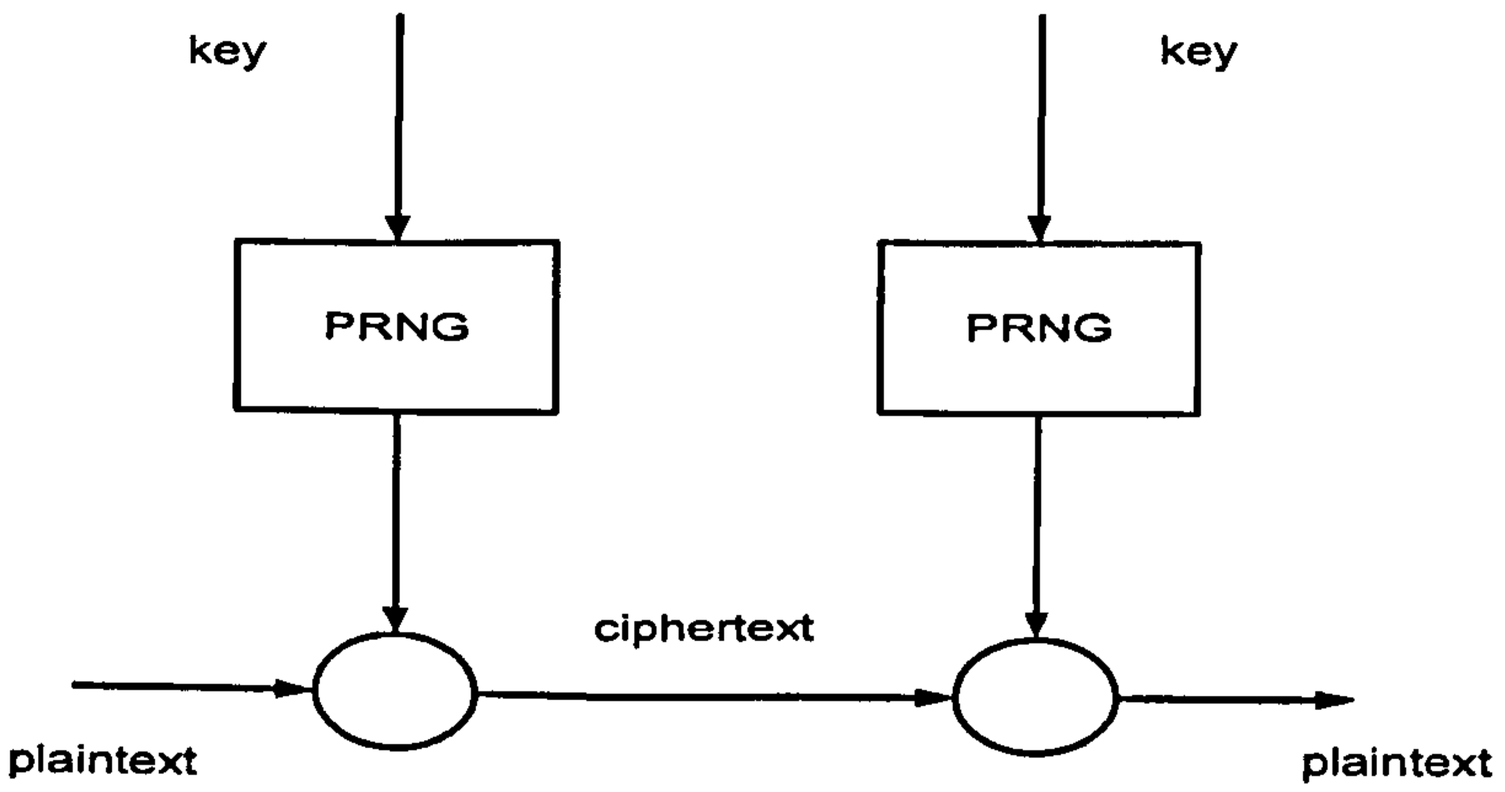


Figure 1.3: Schematic of the basic processes associated with a symmetric encryption system.

Chapter 2

Basic Cryptographic Methods

2.1 History of Cryptography

Cryptography is derived from a Greek word, *crypots*, meaning hidden. It is a study of the mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication. Cryptography is not the only means of providing information security, but rather, one of a class of techniques. One of the best examples of early cryptography is the Caesar cipher, named after Julius Caesar because he is thought to have used it even if there is no strong evidence that he actually invented it [4].

The Caesar cipher is a substitution cipher. Encryption is achieved by transforming each letter of the plaintext message into a different letter to produce the ciphertext. For example, if the shift factor is 3 (see Figure 2.1), then: A becomes K, E becomes M, L becomes J, and Q becomes S. Of course, the shift is dependent on the language used, whether it is English, Russian, or

Greek [75]. This cipher can be described using modular arithmetic. Let P be the numerical equivalent of a letter in the plaintext and C the numerical equivalent of the corresponding ciphertext letter. Then

```
P: A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T
P: 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19

P: U  V  W  X  Y  Z
P: 20 21 22 23 24 25

C:  10 17 24 5  12 19 0  7  14 21 2  9  16 23 4  11 18 25 6  13
C:  K  R  Y  F  M  T  A  H  O  V  C  J  Q  X  E  L  S  Z  G  N

C: 20 1  8  15 22 3
C: U  B  I  P  W  D
```

Figure 2.1: The correspondence of letters for the cipher with $C \equiv 7P + 10 \bmod (26)$

P: plaintext C: ciphertext

This correspondence is obtained in the following way. Letter L is assigned numerical 11. Since $C \equiv 7P + 10 \bmod 26$. $7 \times 11 + 10 = 87 \equiv 9 \bmod 26$. As we can see from the above table 9 is the numerical equivalent of J.

The example below illustrates enciphering of a message using Ceaser's Cipher. The message is:

```
PLEASE SEND MONEY.
```

Broken into a group (blocks) of 5 letters, the message now reads.

```
PLEAS ESEND MONEY.
```

It does not have to be 5 letters, in fact it can be any number. The reason for keeping it uniform is not to reveal the actual word size. For example, if a message contains a lot of 3 letter words, it is easy to guess the word is *the*.

Skipping details, the ciphertext obtained is

```
LJMKG MGMXF QEXMW.
```

Deciphering is, of course, the inverse of the above process.

Since Roman times and, in particular, since the development of programmable computes, a wide class of ciphers have evolved together with the terminology associated with Crytography in general. This Section discusses the background to different ciphers and associated terminology with an emphasis on that which is used throughout this thesis. Basic encryption systems fall into two primary categories, transposition and substitution cipher (or both).

2.1.1 Transposition Ciphers

In a transposition cipher the plaintext remains the same, but the order of characters is shuffled around within a block . A simple transposition cipher preserves the number of characters of a given type within a block, making it an easy task for cryptanalysts.

2.1.2 Substitution Ciphers

Substitution ciphers are block ciphers which replace symbols by other symbols. Simple substitution ciphers over small block sizes provide inadequate security even when the key space is extremely large. For example, letter E occurs more frequently than any other letter in the English text.

2.2 Block Ciphers

Block ciphers are a logical and natural extension to implementing an encryption algorithm and the use of block ciphers is fundamental to the work undertaken and the encryption system developed for this thesis. A block cipher is a type of symmetric-key encryption algorithm that transforms a fixed-length block of plaintext data into a block of ciphertext data of the same length. This transformation takes place under the action of a user-provided secret key.

Since different plaintext blocks are mapped to different ciphertext blocks (to allow unique decryption), a block cipher effectively provides a permutation (one to one reversible correspondence) of sets of all possible messages. The permutation affected during any particular encryption is of course secret, since it is a function of the secret key. When we use a block cipher to encrypt a message of arbitrary length, we use techniques known as modes of operation for the block cipher. To be useful, a mode must be at least as secure and as efficient as the underlying cipher. Modes may have properties in addition to those inherent in the basic cipher. The standard modes are:

2.2.1 Electronic Codebook Mode (ECB)

In ECB [76], each identical block of plaintext gives an identical block of ciphertext. The plaintext can be easily manipulated by removing, repeating, or interchanging blocks. ECB allows easy parallelization to yield higher performance.

Since ciphertext blocks are independent, malicious substitution of ECB blocks (e.g. insertion of frequency occurring blocks) does not affect the decryption of adjacent blocks. Furthermore, block ciphers do not hide patterns - identical ciphertext blocks imply identical plaintext blocks. For this reason, the ECB mode is not recommended for messages longer than one block, or, if keys are reused, for more than a single block message. Security may be improved somewhat by inclusion of random padding bits in each block.

The problem with ECB mode is that if a cryptanalyst has the plaintext and ciphertext for several messages, he/she can start to compile the codebook without knowing the key - see Figure 2.1. In most real world situations, fragments of messages tend to repeat and different messages may have bit sequences in common. Computer generated messages, like electronic mail, tend to have regular structures. Further, messages can be highly redundant or may have long strings of zeros or spaces.

2.2.2 Cipher Block Chaining Mode (CBC)

In CBC [77] mode, each plaintext block is XORed with the previous ciphertext block and then encrypted. An initialization vector is used as a 'seed'

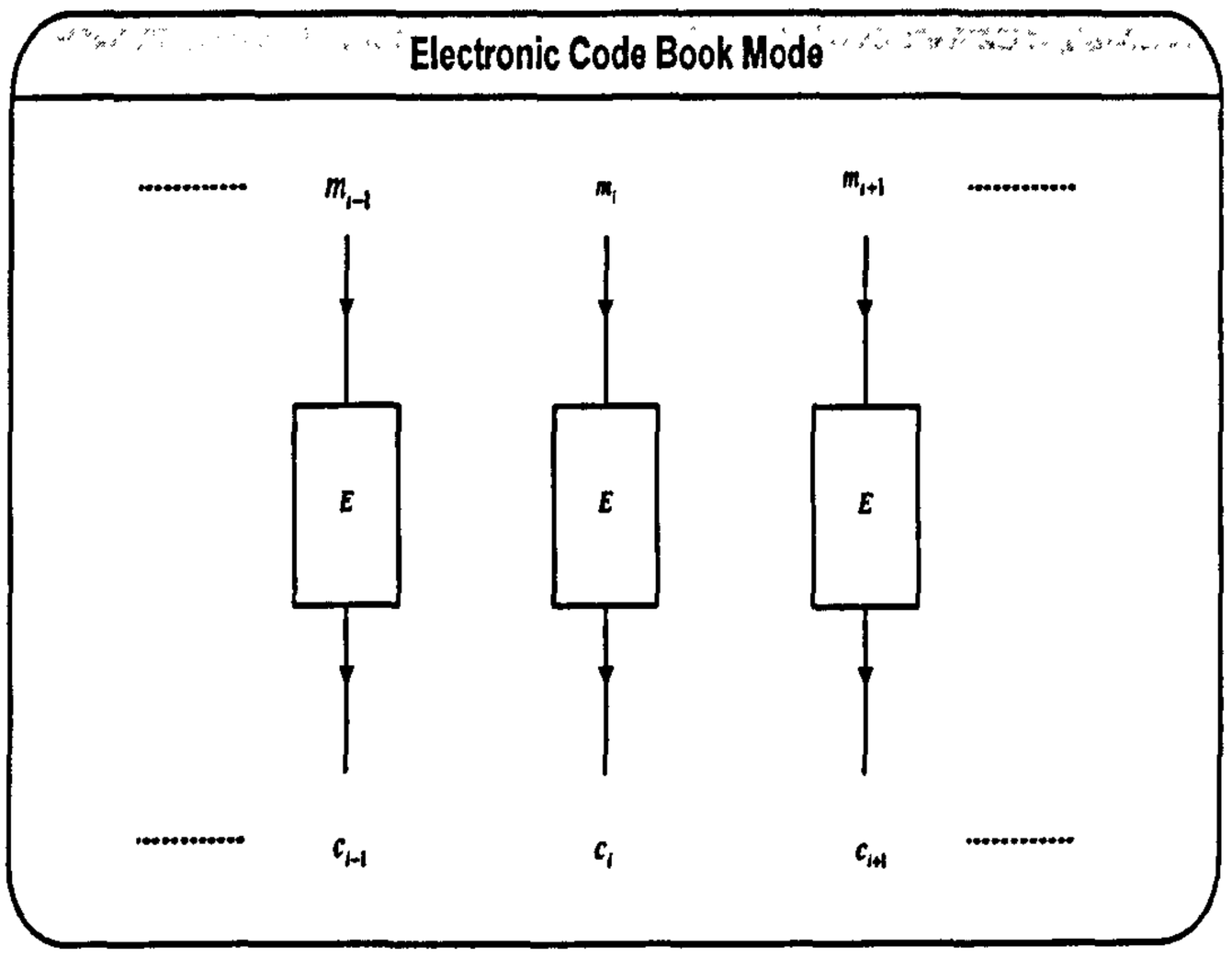


Figure 2.1: Electronic codebook mode.

for the process. The chaining mechanism causes a ciphertext to depend on all preceding plaintext blocks (the entire dependency on preceding blocks is, however, contained in the value of previous ciphertext block). Consequently, rearranging the order of ciphertext blocks affects decryption. Proper decryption of a correct ciphertext block requires a correct preceding ciphertext block as illustrated in Figure 2.2.

CBC mode forces identical plaintext blocks to encrypt to different ciphertext blocks only when some previous plaintext block is different. Two identical messages will still encrypt to the same cipher text. Worst still, two messages which begin the same, will encrypt in the same way to the first difference. Some messages have a common header: a letterhead or a 'from' line. While block replay is still impossible, this identical beginning can give a cryptanalyst some useful information. A simple solution to this problem is to encrypt random data over the first block. The block of random data is called the Initialisation Vector (IV), initialisation variable, or initial chaining value. The IV has no meaning, it is there just to make each message unique. A timestamp, or addition of some random bits make a good IV.

With the addition of IVs, identical plaintext messages encrypt to different ciphertext messages. Thus, it is impossible for an eavesdropper to attempt a block replay, and more difficult for him/her to build a codebook. While the IV should be unique for each message and encrypted with the same key, it is not an absolute requirement.

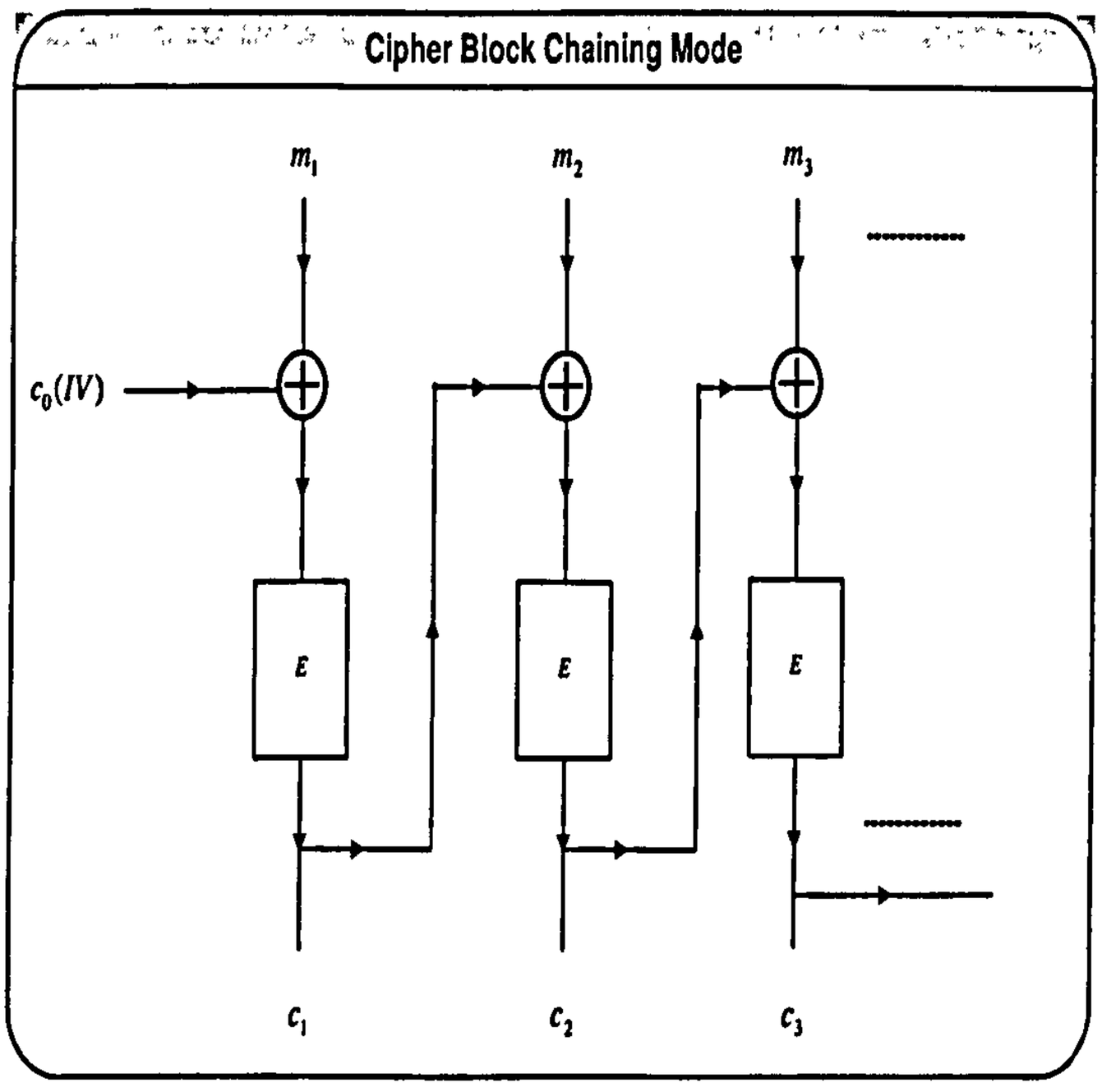


Figure 2.2: Cipher block chaining mode.

2.2.3 Cipher Feedback Block Mode (CFB)

In CFB [78] mode, the previous ciphertext block is encrypted and the output produced is combined with the plaintext block using XOR to produce the current ciphertext block. It is possible to define CFB mode so it uses feedback that is less than one full data block. An initialization vector is used as a 'seed' for the process - see Figure 2.3.

In CFB, the plaintext patterns are concealed in the ciphertext by the use of the XOR operation. Plaintext cannot be manipulated directly except by the removal of blocks from the beginning or the end of the ciphertext. With CFB mode and full feedback, when two ciphertext blocks are identical, the outputs from the block cipher operation at the next step are also identical. This allows information about plaintext blocks to leak. The security considerations for the initialization vector are the same as in CBC mode.

2.2.4 Output Feedback Block Mode (OFB)

OFB [79] mode is similar to CFB mode except that the data that is XORed with each plaintext block is generated independently of both the plaintext and ciphertext. An initialization vector is used as a 'seed' for a sequence of data blocks s_i, say, and each data block s_i is derived from the encryption of the previous data block $s_i - 1$. The encryption of a plaintext block is derived by XORing the plaintext block with the relevant data block (see Figure 2.4)

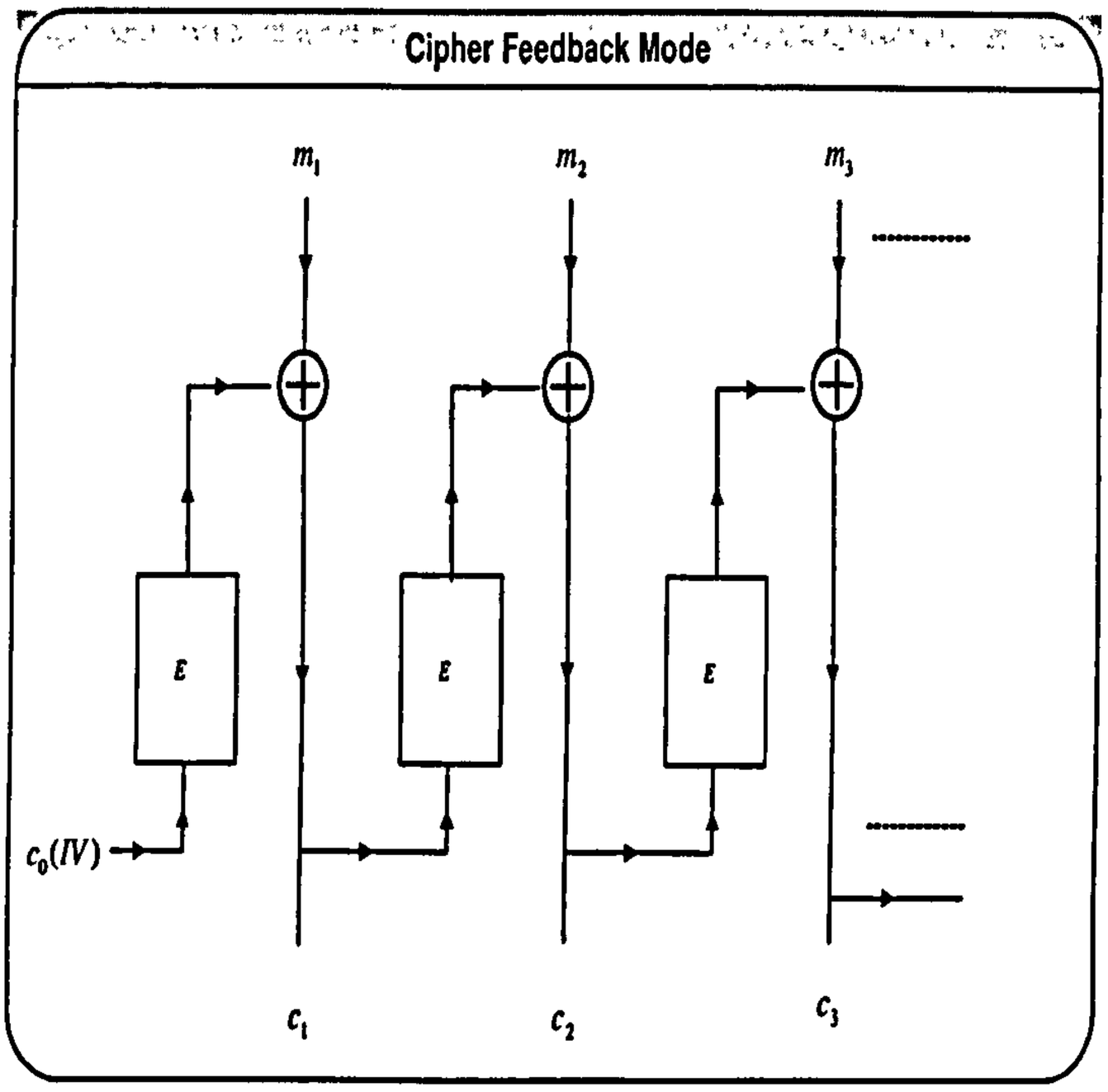


Figure 2.3: Cipher feedback chaining mode.

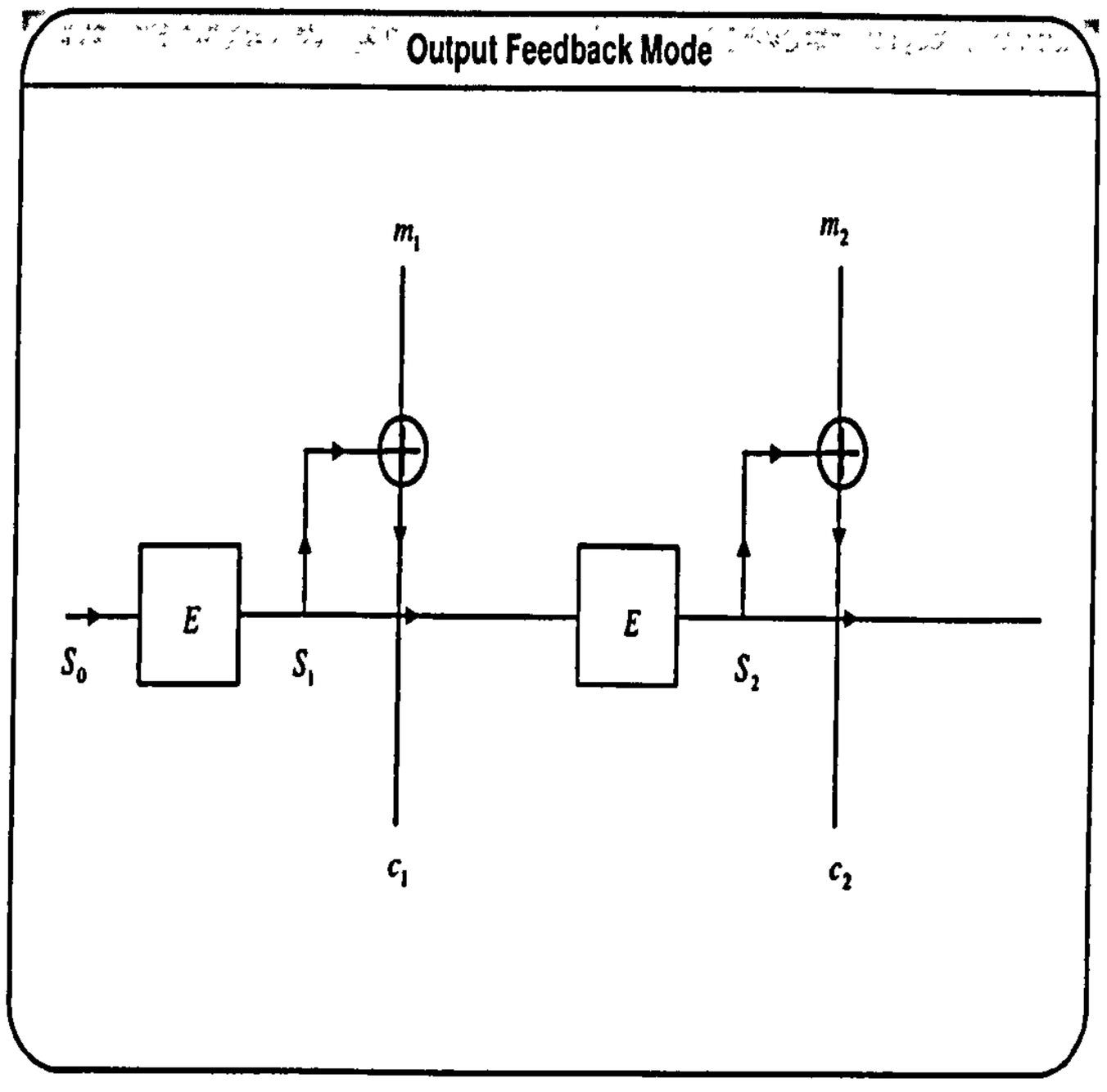


Figure 2.4: Output feedback chaining mode.

OFB mode has an advantage over CFB mode in that any bit errors that might occur during transmission are not propagated to affect the decryption of subsequent blocks. The security considerations for the initialization vector are the same as in CFB mode. A problem with OFB mode is that the plaintext is easily manipulated. Namely, an attacker who knows a plaintext block m_i may replace it with a false plaintext block x by XORing m_i with the corresponding ciphertext block c_i

2.3 Stream Cipher

Stream ciphers [61] [73] form an important class of symmetric key encryption schemes. What makes them useful is the fact that the encryption transformation can change for each symbol of plaintext being encrypted. In situations where transmission errors are highly probable, stream ciphers are advantageous because they have no error propagation problems. They can also be used when the data must be processed one symbol at a time (e.g. typical usage in low memory devices such as mobile phone communication). The security of the stream cipher depends entirely on the keystream generator. If the generator generates true random bits, the security can be considered as perfect. If the generator gives out a stream of zeros for example, then the ciphertext will be the same as the plaintext!

Stream ciphers can be designed to be exceptionally fast, much faster than any block cipher. While block ciphers operate on large blocks of data, stream ciphers typically operate on smaller units of plaintext, usually on a bit by bit basis. The encryption of any particular plaintext with a block cipher will result in the same ciphertext when the same key is used. With a stream ci-

pher, the transformation of these smaller plaintext units will vary, depending on when they are encountered during the encryption process.

A stream cipher generates a keystream. Encryption is accomplished by combining the keystream with the plaintext, usually with the bitwise XOR operation. The generation of the keystream can be independent of the plaintext and ciphertext, yielding what is termed as synchronous stream cipher. It can also depend on the data and its encryption, in which case, the stream cipher is said to be self-synchronizing. Most stream cipher designs are for synchronous stream ciphers.

Current interest in stream ciphers is most commonly attributed to the appealing theoretical properties of the one-time pad, but there have been, as of yet, no attempts to standardize any particular stream cipher proposal as has been the case with block ciphers. Interestingly, certain modes of operation of a block cipher effectively transform it into a keystream generator and, in this way, any block cipher can be used as a stream cipher. However, stream ciphers with a dedicated design are likely to be much faster. A number of shift registers are implemented and used in stream cipher techniques.

2.4 Symmetric Ciphers

In a symmetric cipher, both parties must agree on the encryption key (and encryption algorithm) in advance. The key used in symmetric cipher is the same for both the sender and recipient. Symmetric systems keys are also termed as shared secret systems or private key systems. Symmetric ciphers are significantly faster than asymmetric ciphers, but the requirements for key exchange make them difficult to use. DES (Digital Encryption Standard) an

DES3 (essentially the Digital Encryption Standard with triple encryption) and AES/Rijndael (Advanced Encryption Standard by Joan Daemen and Vincent Rijmen) are examples of symmetric ciphers which are used in many banking systems for example and in some military applications.

2.5 Asymmetric Ciphers

In an asymmetric cipher, the key is negotiated between the parties during communication. In this system, each person has two keys. The first key, the public key, is shared publicly. The second key is private, and is kept secret. When working with asymmetric cryptography, the message is encrypted using the recipients' public key. The recipient then decrypts the message using his/her private key. That is what makes the system asymmetric.

Because asymmetric ciphers tend to be significantly more computationally intensive, they are usually used in combination with symmetric ciphers to implement public key cryptography. The asymmetric cipher is used to encrypt a session key and the encrypted session key is then used to encrypt the actual message. This gives the key-exchange benefits of asymmetric ciphers with the speed of symmetric ciphers. RSA and Diffie-Hellman are asymmetric ciphers. [61] [41]. Asymmetric ciphers are also know as public key cryptography. This concept was first invented by Diffie and Hellman in 1976 [29] [83]. The public key is made freely available to the public. This may serve as a convenience, because one does not have to worry about how to exchange keys. But on the other hand, it is a good starting point for cryptanalysts. Given that $C = E_k(P)$ where P is the plaintext, C is the ciphertext and E is the key (k) dependent encryption algorithm, the analyst can guess P and

check the answer. This may cause problems if the number of possible text messages is small enough to allow for an exhaustive search. However, most of the public key algorithms are designed to resist chosen-plaintext attack. It is therefore not easy to deduce the secret key from the public key and the plaintext cannot be easily recovered from the ciphertext.

Even though the public key system with a secure algorithm can be considered secure, a lot of issues have been raised about the mechanism of key exchange, and who is involved in the process. In the article, *Ten Risks of PKI: What you are not being told about Public Key Infrastructure*, Carl Ellison and Bruce Shneier [31] highlight some important facts to be considered when using public key infrastructure. In essence, a detailed analysis of public key and asymmetric systems in general, reveals that the level of security is not as significant as that which can achieved using a well designed symmetric system which is the basis for encryption engines developed in this thesis.

The table below compares the difference on key length when using symmetric or asymmetric cipher. [83]:

Symmetric	**Public**
Key length	Key Length
56 bits	384 bits
64 bits	512 bits
80 bits	768 bits
112 bits	1792 bits
128 bits	2304 bits

Table 2.1: Symmetric and public-key lengths with similar resistance and brute-force attacks.

2.6 Hash Functions

A hash function is a one way function which takes an input and returns a fixed-size output string [97] [50]. Hash functions have a variety of general computational usages; they provide one of the best ways for checking the authenticity of stored files. For example, if a file has been modified, when its hash function is recalculated, there will be a change in the output value (hash values). Hash functions are quite useful for network administrators as they can use them for files that are quite important in running the system, and do not change at all, or maybe do not change often. Tripwire Inc. [90], provides software which periodically calculates hash function that a network administrator could monitor. If there are any changes, the administrator will be notified which helps to identify a potential attack to the network. Cryptographic algorithms such as RC5 and SHA1 [89] use hash functions. When employed in cryptography, the hash functions are usually chosen to have some additional properties such as:

the input can be of any length;

the output has a fixed length;

$H(x)$ is relatively easy and fast to compute for any given x;

$H(x)$ is one-way;

$H(x)$ is collision-free.

where $H(x)$ is the hash function.

2.7 One Time Pads

The One Time Pad (OTP), invented in 1917 [35], is a theoretically unbreakable method of encryption where the plaintext is combined with a random number stream of the same length. Co-invented by Gilbert Vernam, who also invented stream cipher, OTP is also known as Vernam cipher. This cipher is often described as perfectly secure and unbreakable. The method has been mathematically proven unbreakable. Even though the method is secure, it is not popular, mainly because of its drawbacks in the key exchange, i.e. the key cannot be used more than once. The research undertaken within this thesis, is in a broad sense, an attempt to produce a user friendly OTP by generating one time key from the plaintext which is: (i) used to encrypt the plaintext; (ii) transmitted with the ciphertext as a covert watermark. Even though this is an attempt, the cipher produced in this thesis cannot be strictly termed as OTP mainly because the random numbers generated by the BBS are not truly random.

2.8 Cryptanalysis

Any good crypto system must be able to withstand cryptanalysis. While there are several good books on cryptography, there are not many books on cryptanalysis. One reason is because this is a fast-moving field, and things are changing all the time. Thus, any book written on the subject can be obsolete before it gets printed (e.g. a self-study course in cryptanalysis [85] and

the Hackers Black Book). Cryptanalysts work on 'attacks' to try and break the system. In many cases, the cyptanalysts are aware of the algorithm used, and will try to break the algorithm in order to compromise the keys or gain access to the actual plaintext. It is worth noting that even though a number of algorithms are freely published, this does not in any way mean that they are the most secure. Major government institutions do not reveal what type of algorithm they use for their communication. The rationale for this is that, if we find it difficult to break a code with knowledge of the algorithm then how difficult it is then to break a code if the algorithm is unknown? On the other hand, within the academic community, security in terms of algorithm secrecy is not considered to be of high merit and publication of the algorithm(s) is always recommended. It remains to be understood whether this is a misconception within the academic world (due in part to the innocence associated with academic culture) or a covertly induced government policy. In 2003, it was reported that the US had broke ciphers used by the Iranian intelligent services [72], which goes to show that the encryption experts and the cryptanalists are in a leap frog race. What was not mentioned, is the fact that the Iranian ciphers were based on systems purchased indirectly from the US and, thus, based on US designed algorithms!

There are several methods by which a system can be attacked. In all the methods it is assumed that the cryptanalyst has full knowledge of the algorithms used! These are discussed below.

2.8.1 Ciphertext-only Attack

In this type of attack, the cryptanalyst has a ciphertext of several messages at his disposal. All of these messages have been encrypted using the same

algorithm. The challenge for the cryptanalyst is to try and recover the plaintext of these messages. At the same time, he/she will be in a better position if he/she can recover the actual keys used for encryption.

2.8.2 Known-plaintext Attack

The cryptanalyst task is simpler in this case because he/she has access to both the plaintext and the corresponding ciphertext. He/she needs to deduce the key used for encrypting those messages, or come up with an algorithm to decrypt any new messages encrypted with the same key.

2.8.3 Chosen-plaintext Attack

In this case the cryptanalyst possesses both the plaintext and the ciphertext. In addition to this he/she also has the ability to encrypt plaintext and recover the ciphertext produced. This gives him/her a more powerful tool which should enable him/her to deduce the keys.

2.8.4 Adaptive-chosen-plaintext Attack

This is an improved version of the chosen-plaintext attack. In this version, the cryptanalyst has the ability to modify the results based on the previous encryption. This version allows the cryptanalyst to choose a smaller block for encryption.

2.8.5 Chosen-ciphertext Attack

Here the cryptanalyst has access to several decrypted texts. In addition, the cryptanalyst is able to use the text and pass it through a 'black box' for an attempted decrypt. The cryptanalyst has to guess the keys in order to use this method which is performed on iterated basis (for different keys), until a decrypt is obtained.

2.8.6 Chosen-key attack

This method is based on some knowledge on the relationship between different keys and not a very practical attack strategy except in special circumstances.

2.8.7 Rubber-hose Cryptanalysis

This is based on the use of human factors such as blackmail, physical threat, or torture. It is often a very powerful attack and sometimes very effective.

2.8.8 Differential Cryptanalysis

Discovered by Eli Biham and Adi Shamir in the late 1980s, this is a more general form of cryptanalysis. It is the study of how differences in an input can affect the resultant difference in the output. Biham and Shamir published a number of attacks against various block ciphers and hash functions, including a theoretical weakness in the Data Encryption Standard (DES). This method of attack is usually on a chosen plaintext attack, meaning that

the attacker must be able to obtain encrypted ciphertexts for some set of plaintexts of his own choosing.

2.8.9 Linear Cryptanalysis

This is a known plaintext attack which uses linear relations between inputs and outputs of an encryption algorithm that holds with a certain probability. This approximation can be used to assign probabilities to the possible keys and locate the most probable one.

2.9 Discussion

All the attack strategies discussed above are based on *a priori* knowledge of the exact algorithm that is being used (published or otherwise). Since conventional encryption systems are based on a single algorithm that operates in the same way under different conditions and in a way that is independent of the input plaintext, it is an example of mono-static data processing.

One obvious way of increasing the security of an encryption system is to consider new algorithms whose functional form is kept secure. But this approach falls short of the principle of algorithm accessibility insisted upon primarily by the academic community. However, there is another approach that can be considered which is based on the modulation of:

(i) the algorithms themselves;

(ii) the parameters that 'drive' them.

This involves application of a 'multi-dynamic' paradigm. In this thesis we consider both cases. In case (i) above, we review the use of deterministic chaos [69] for designing a multiplicity of algorithms which can be published in the knowledge that a multiplicity of new algorithms can be generated relatively easily which is, at least in principle, inexhaustive. This facility exists because of the infinite variety of non-linear (chaotic) iteration functions that can be invented for this purpose; a facility that is not available to the same extent with conventional approaches to algorithm design, i.e. through the use of pseudo random number generators. This is discussed in section 3.17. With regard to case (ii) above, with conventional encryption algorithms, the parameters that 'drive' them are invariably prime numbers. Thus, in order to exercise the principal of multi-dynamicism using conventional encryption algorithms, we can introduce a design strategy that is based on prime number modulation which is the subject of Chapter 5. This is based on the theory of encryption discussed in Chapter 3 which includes a review of the properties of prime numbers and how these properties can be used to compute prime numbers efficiently in order to implement prime number modulation in practice. In addition to applying a multi-dynamic paradigm to the design of an encryption engine, we also investigate how this approach can be applied to the generation and exchange of keys which forms the subject of Chapter 5.

Chapter 3

Encryption Techniques and Systems

This chapter begins with an introduction to prime numbers, demonstrating the importance in their study when it comes to cryptography. It includes a review that highlights why, in cryptographic applications, prime numbers are so essential. We then discuss different types of (prime number based) pseudo random number generators (PRNGs) together with their strengths and weaknesses and explain how these generators can be implemented in order to design Dynamic Block Encryption (DBX) systems. Finally we provide a brief introduction to different types of cryptographic techniques and discuss the operational characteristics of the more secure ciphers such RSA and DES.

3.1 Prime Numbers

There is huge literature concerning prime numbers and their relationship to number theory in general. In this section, we shall discuss those properties of prime numbers that are of specific importance to encryption in terms of their computation and, in particular, the characteristics that are necessary to implement a prime number modulation scheme.

A prime number is an integer greater than 1 that is divisible by no other integer except by 1 and itself [98]. The study of prime numbers has fascinated mathematicians for hundreds of years because of their mystical and numerological properties, finding the properties of prime numbers very appealing. However, for years there has never been any real use for them. All that changed as the need to design encryption systems using programmable computers grew. This is because prime numbers are building blocks of all integers. Every integer is either itself a prime or the product of primes. In this sense, there is a similarity between prime numbers and atoms. Prime numbers are as important to number theorists and atoms are to materials scientists.

If P is the set of all prime numbers, then any positive integer a can be written uniquely in the following form:

$$a = \prod_{p \in P} p^{a_p}$$

where each $a_p \geq 0$

For example,

$$3600 = 2^4 \times 3^2 \times 5^2$$

The value of any positive integer can be specified by simply listing all the

nonzero exponents in the foregoing formulation. Integer 12 is represented by $a_2 = 2, a_3 = 1$ and integer 18 is represented by $a_2 = 1, a_3 = 2$ for example.

Multiplication of two numbers is equivalent to adding the corresponding exponents:

$$k = mn \rightarrow \qquad k_p = m_p + n_p \qquad \texttt{for all } p \epsilon P$$

$$k = 72 \times 60 = 4320$$

$$72 = 2^3 \times 3^2$$

$$60 = 2^2 \times 3^1 \times 5^1$$

$$k_2 = 3 + 2 = 5$$

$$k_3 = 2 + 1 = 3$$

$$k_5 = 1 = 1$$

$$4320 = 2^5 \times 3^3 \times 5^1$$

Primes are also quite useful when it comes to determining the *greatest common divisor (gcd)* of two integers:

Again we have:

$$72 = 2^3 \times 3^2$$

$$60 = 2^2 \times 3^1 \times 5^1$$

$$gcd(60, 72) = 2^2 \times 3^1 \times 5^0$$

and in general,

$$k = gcd(a, b) \rightarrow \qquad k_p = \texttt{ min } (a_p, b_p) \texttt{ for all } p$$

3.1.1 Fermat's Little Theorem

Fermat's Little Theorem states the following: If p is prime and a is a positive integer not divisible by p, then

$$a^{p-1} = 1 \pmod{p}$$

For example, given $a = 7, p = 17$

$7^{16} = 1 \pmod{3}$

The same formula can also be written as $a^p \equiv a \pmod{p}$.

3.1.2 The Search for Prime Numbers

To date, there is no known rule that tells us what the n^{th} largest prime is. However, by analysing the pattern of primes we reveal some interesting

features. For example, if we compute the differences between successive primes, we obtain the following list: 1, 2, 2, 4, 2, 4, 6, 2, 6, 6, 4, 6, 6, 2, 6, 4, 2, 6, 4, 6, 8, 4, 2, 4, 2, 4, 14, 4, 6, 2, 10. (That is, 1 = 3 - 2, 2 = 5 - 3, 2 = 7 - 5, 4 = 11 - 7, and so on). The list is somewhat disorderly, but the numbers in it start to get gradually larger. Of course, they do not increase steadily, but the numbers as much as 10 and 14 do not appear until quite late on, while the first few are all 4 or under [37].

If we write out the first ten thousand primes, then the gaps between successive numbers get larger. This is to be expected because as an integer n becomes larger, there is a greater likelihood for smaller integers $m < n$ to exist such that $n/m = k$ where k is an integer. In other words, as an integer increases in size, the probability of finding successive primes gets smaller.

Numerous scholars have developed different theories on prime numbers. For example, in 1742 Christian Goldbach [20] wrote to Leonhard Euler and stated that every even integer greater than 4 is a sum of 2 odd primes and every integer greater than 5 is a sum of 3 primes. This is the famous Goldbach conjecture. Because it is a conjecture rather than a proof, there has been a number of attempts to disprove this statement by using computers with large word lengths and extremely large numbers. Mathematicians and other researchers have tested the conjecture against larger and larger even numbers and 'there are strong grounds for believing that Goldbach's conjecture is true, and it feels like just a matter of time before someone figures out how to prove it' [68] says Joe Buhler of the Mathematical Sciences Research Institute in Berkeley, California. 'The real justification is algorithmic. In figuring out how to carry out the computations that far, one has to extend and polish algorithmic programming techniques, and the nature of the scientific advance

in this case is much more in algorithmics than in number theory'.

Like other aspects of mathematics, mathematicians try to identify patterns which can then be quantified in terms of a probable theorem. The striking feature about prime numbers is that they appear to have patterns and correlation (in terms of different indirect computational properties), but that these properties are not universal, i.e. they are not invariant of the scale in magnitude of the prime numbers that are considered. Thus, prime numbers are, in a sense, very elusive entities. Just as one property appears to be correct, they 'play another trick'. To date, there is still controversy as to whether prime numbers have some deterministic pattern yet to be discovered or are actually random. Given their apparent random nature, prime numbers continue to fascinate modern mathematicians [3]. For example, Ivan Vinogradov, in 1937, tried to prove the work of Goldbach. He was able to combine his *bilinear form technique* and his *mean value theorem* to reduce the Goldbach Ternary Problem to checking a finite number of cases.

A number of methods are available to detect primes, most of these are effective only for smaller primes, but when dealing with large numbers, it becomes difficult to determine whether a particular number is prime or not. Even though there are an infinite number of primes, [66] of the first 25 billion whole numbers, only 1,091,987,405 or about 4 percent are primes, and the proportion of primes decreases as the numbers get bigger. Since the numbers get so large, the need for efficient ways of identifying primes is a subject of continuing research. One method for identifying a number as a prime is by dividing it by primes; we first test if it is an even number and then find if it divisible by 3, 5, 7 and so on. This method is fine for small numbers, but once the numbers get large it becomes slow and difficult to compute.

Since 1945, and the development of programmable computers, cryptographers have taken a keen interest in prime numbers, and this has increased the need for analysing them. In cryptography, the need is necessary because it is trivial to multiply 2 primes together but very difficult to compute the two primes given the product. For example, given the primes 7317631 and 234239, multiplying them yields 1714074567809. The problem is then to obtain 7317631 and 234239 from 1714074567801. Clearly, factoring huge numbers is not an easy task.

Goldbach's conjecture remains just a conjecture. Thus, work continues to be undertaken to prove Goldbach's conjecture *prime pairs.*

In 1998, Herman te Riele [45] used a Cray C916 supercomputer to check that all even numbers up to 10^{14} satisfy the Goldbach conjecture. His work was based on: (i) the assumption that under the Generalized Riemann hypothesis, every odd number ≥ 7 can be written as a sum of three prime numbers; (ii) under the assumption of the Riemann hypothesis, every even positive integer can be written as a sum of at most four prime numbers. Goldbach's conjecture has then been verified for all even numbers in the intervals [105i; 105i + 108], for i = 3, 4,..., 20 and [1010i; 1010i + 109], for i = 20, 21,..., 30.

The search for Goldbach's twin primes continues. With the increase in computer power, we obtain results faster. In 2000 Richstein managed to verify up to 4×10^{14} primes. He also investigated a number of different ways in which a number can be expressed as the sum of two primes. He proved that, as the even integers get larger, the number of such prime-pairs increases. Richstein found the number of such sums for all even integers up to 500 million.

Table 3.1 shows how the number of pairs used to express a number increases as the even number gets larger.

Integer	No. of Goldbach Partitions
10	2
100	6
1,000	28
10,000	127
100,000	810
1,000,000	5,402
10,000,000	38,807
100,000,000	291,400

Table 3.1: A table showing Goldbach's twin primes

The search for Goldbach's twin primes began some time ago, but picked up speed with the introduction of computers. Initially, it was taking time for mathematicians to push the limit for Goldbach's pairs, the table below shows the nature of the progress over time, i.e. over the last 200 years.

Other mathematicians have taken interest in calculating and detecting prime numbers. The French mathematician, Mersenne, came up with the theory that $2^n - 1$ is prime if n is prime. However, it was later proved that not all numbers under this equation are primes. For example: even though 11 is prime, $2^{11} - 1$ is 2047 which is not prime. Since the Mersenne conjecture, a long search was initiated to find what are known today as Mersenne's Primes. The Greatest Internet Mersenne Prime Search (GIMPS) [57] project involved members connecting their PCs to a central computer which tested and verified Mersenne primes. In May 2004, the $41st$ Mersenne prime was found. The number itself contains 7,235,733 decimal digits, with $n = 224{,}036{,}583$. The search for the largest Mersenne prime still continues and the 'race' will

Bound	Reference
1×10^4	Desboves 1885
1×10^5	Pipping 1938
1×10^8	Stein and Stein 1965ab
2×10^{10}	Granville et *al. 1989*
4×10^{11}	Sinisalo 1993
1×10^{14}	Deshouillers *et al. 1998*
4×10^{14}	Richstein 2001
2×10^{16}	Oliveira e Silva March 2003
6×10^{16}	Oliveira e Silva Oct 2003

Table 3.2: The progress in validating Goldbach's twin primes.

no doubt continue for some time to come.

3.1.3 Prime Types

Ferrier's Prime

According to Hardy and Wright (1979), the 44-digit Ferrier's prime $F \equiv \frac{1}{17}(2^{148}+1) = 20988936657440586486151264256610222593863921$, determined to be a prime using only a mechanical calculator [16], **is the largest prime found before the days of electronic computers.** Mathematica can verify primality of this number in a (small) fraction of a second, showing how far the art of numerical computation has advanced in the intervening years.

Sophie Germain Prime

A prime p is said to be a Sophie Germain prime if both p and $(2p+1)$ are prime. The first few Sophie Germain primes are 2, 3, 5, 11, 23, 29, 41, 53, 83, 89, 113, 131, ...

The largest known Sophie Germain prime is $7068555 * 2^{121302-1}$, which has 36523 digits [17]. It is not known if there are an infinite number of Sophie Germain primes [42].

Wieferich Prime

A Wieferich prime [18] is a prime p which is a solution to the congruence equation $2^{p-1} \equiv 1 (\mathtt{mod}\ p^2)$

Note the similarity of this expression to the special case of Fermat's little theorem $2^{p-1} \equiv 1 \pmod{p}$ which holds for all odd primes. The first few Wieferich primes are 1093, 3511, ... with none other less than 4×10^12. Interestingly, one less than these numbers have suggestive periodic binary representations: $1092 = 10001000100_2, 3510 = 110110110110_2$.

3.1.4 Prime Patterns

The fascination with prime numbers still continues and there are a number of (non-universal))patterns which, when studied, can help us find a way to succeed in learning more about the prime numbers. With the help of computers, prime numbers dispay some interesting patterns as seen in [40].

Relative Primes

For $n+1$ integers less than or equal to $2n$, there are always two of them which are *relatively* prime. For example, if n is 5, then take any 6 integer from the set: 1,2,3,4,5,6,7,8,9,10. Out of these, there are two which are relatively prime [42].

3.2 Generating Prime Numbers

A number of methods are available for generating primes. The most widely used is the sieve of Eratosthenes. This method works fine for primes less than 100,000,000. For larger values, we need to apply different methods.

3.2.1 Primality Test

When working with large numbers, we often need to test them for primality before we can use them. Stallings [89] has formalised a routine TEST which takes a number n and tests for primality. Even though the condition can be met, it does not mean that the number is prime, but is a highly probable prime number. Given n is the number to be tested.

1. Find integers k, q, with $k > 0, q$ odd so that $(n - 1 = 2^k q)$;
2. Select a random integer $a, 1 < a < n - 1$;
3. **if** $a^q \bmod n = 1$ **then** return('inconclusive');
4. **for** $j = 0$ **to** $k - 1$ **do**
5. **if** $a^{2^j q} \bmod n = n - 1$ **then** return('inconclusive');
6. Return('composite');

The interest in primality testing has grown rapidly since cryptographers introduced the public key exchange mechanism in the 1990s. The security involved primarily relies in factoring very large numbers. Integer factorization poses many problems, a key problem being the testing of numbers for primality. A reliable and fast test for primality would bring us a step closer to decoding data containing secret information. Therefore, cryptanalyst research communities have began to address the problem of primality testing with increased vigor. A primality test is a simple function that determines if a given integer is prime or composite. Some methods used for primality testing are addressed below.

Sieve of Eratosthenes

This method exemplifies both the simplicity of testing for primality and the restraints on the efficiency of such tests. The algorithm itself is a fairly straightforward process and easy to implement, based almost completely on the definition of primes [1]. However, even though it is easy to implement, it is by no means efficient. In cryptography, most primality testing is concerned with large numbers, usually in excess of 100 digits. If we were to use the Sieve of Eratosthenes to determine the primality of a number with just 20 digits, we would need first to find at least all the primes up to 10^{10}. There are around 450 million primes less than 10^{10}. At the rate of finding one prime per second (including 'crossing off' all the multiples), we would be working for a little over 14 years to find 450 million primes, which would then have to be divided into our original 20-digit number. Clearly, the amount of time

needed to determine just one number is quite impractical for cryptographic purpose. Of course, we can use faster and more efficient computers to run this algorithm. We can also find better methods of storing the primes numbers. They could be stored in a database for quick retrieval for example. However, in general, this method is still inefficient.

Trial Division Method

If n is a composite positive integer, then n has a prime divisor p which is less than or equal to $\sqrt{n}$. The algorithm used for testing first checks for all prime numbers p that are less than or equal to $\sqrt{n}$ and whether they divide n. The prime numbers $p \leq \sqrt{n}$ can either be generated by the sieve of Eratosthenes or obtained from a database of prime numbers.

Fermat Test

From Fermat's theorem [15]: If n is prime, then $a^{n-1} \equiv 1 \bmod n$ for all $a \in Z$ with with $\gcd(a, n) = 1$. It can be used to determine that a positive integer is composite. For example, let $n = 341 = 11 * 31$, then we have $2^{340} \equiv 1 \bmod 341$, even though n is composite. If we use the Fermat test with $n = 341$ and $a = 3$, then n is proven composite. The Fermat test proves that n is composite, but does not find a divisor of n. It only shows that n lacks a property that all prime numbers have. Therefore the Fermat test cannot be used as a factoring algorithm.

Euclid's Theorem

Theorem: There are infinitely many prime numbers.

Proof [26]: Suppose there exist only a finite number of primes, $p_1, p_2, p_3, ...p_n$. Now, consider the integer $N = p_1 p_2 ... p_n + 1$. None of the existing primes divides N, since the division N/p_i will always give the remainder 1. Thus either N is a (new) prime number, or N contains a (new) prime factor, which is different from all of those given. This theory therefore proves that there is an infinite number of primes.

The following example starts with the prime 2 and yields at least one new prime in each step:

$N_2 = 2 + 1 = 3$

$N_3 = 2 \times 3 + 1 = 7$

$N_4 = 2 \times 3 \times 7 + 1 = 43$

$N_5 = 2 \times 3 \times 7 \times 43 + 1 = 1807 = 13 \times 139$

$N_6 = 2 \times 3 \times 7 \times 43 \times 139 + 1 = 251085 = 5 \times 50207$

$N_7 = 2 \times 3 \times 7 \times 43 \times 139 \times 50207 + 1$

$= 12603664039 = 23 \times 1607 \times 340999$

$N_8 = 2 \times 3 \times 7 \times 43 \times 139 \times 50207 \times 340999 + 1$

$= 429836833293963$

$= 23 \times 79 \times 2365347734339$

$N_9 = 2 \times 3 \times 7 \times 43 \times 139 \times 50207 \times 340999 \times 2365347734339 + 1$

$= 1016587861619057545906876 1119$

$= 17 \times 127770091783 \times 4680222564147 1129$

There are a number of open questions relating to prime numbers that are the concern of all cryptologists. Solutions may not necessarily solve and provide

a drastic change in cryptographci world, but will help clear some issues which are currently not properly unserstood. This questions can be found here [65]:

3.3 Random Number Generation

The concept of randomness is part of our daily lives in a range of things we do, whether it is buying a lottery ticket, checking weather pattern, or simply running after a ball [51]. The security of a number of cryptographic algorithms depends on the generation of unpredictable random numbers. It is quite essential for use with any sequence that is needed to be generated [91], even though it is difficult to design a true random number generator purely using software. There are a lot of hardware tools available to create random numbers. These are true random numbers and the sequence is impossible to guess. For example, a logic can be constructed in such a way that numbers are generated by the movement of the mouse. This gives random numbers, because no one else will be able to produce exactly the same movement. We can also obtain random numbers by tapping the noise made by a CPU in a motherboard.

Good random number generators enhance the strength of cryptography and many different methods of generating random numbers have been developed. One of the interesting yet simple methods is called the *diceware passphrase* [71]. In this method a list of words is generated and each word numbered. The numbers are generated from ordinary dice, which acts as a random number generator. The numbers that come up in the rolls are assembled as a five digit number, e.g. 43146. That number is then used to look up a word in a word list. A major advantage of the Diceware approach is that the level

of unpredictability in the passphrase can be easily calculated. Each Diceware word adds 12.9 bits of entropy to the passphrase. (That is, log2(65) bits). Five words (slightly over 64 bits) are considered a minimum length.

The best random numbers are created by harnessing natural physical processes, such as radioactivity, which is known to exhibit truly random behaviour. A piece of radioactive material used the emissions detected with a *Geiger counter* [19]. The emissions sometimes can be detected in rapid succession, and at other times there is a long delay between emissions; these delays are unpredictable and random [87]. A display is connected to the Geiger counter which rapidly cycles through the alphabet at a fixed rate, but stops momentarily as soon as an emission is detected. The letter on the display is then used for a random number. The display restarts and once again cycles through the alphabet until it is stopped at random by the next emission, and again, the letter on the display is added to the key, and so on. This process generates truly random numbers but it is impractical to use for cryptographic purpose.

The term 'random' must be used loosely because software based random number generators as used in cryptography are basically pseudo-random, i.e. simulations of random processes at best. A pseudo-random generator is a deterministic algorithm that expands short random seeds into much longer bit sequences that appear to be random. In other words, although the output of a pseudo-random generator is not really random, there is no easy method of telling the difference [36]. The better the pseudo-random number generator, the better the design of an encryption engine. [88] In turn, most generators used for encryption exploit the properties of prime numbers and hence are prime number dependent, hence the importance of prime numbers in applied

cryptography.

Random number generators are not random because they do not have to be. Most simple applications, such as computer games for example, need very few random numbers. Nevertheless, use of a poor random number generator can lead to strange correlations and unpredictable results which are compounded in terms of spurious correlations. These must be avoided at all costs.

The problem is that a random number generator does not produce a random sequence. In general, random number generators do not necessarily produce anything that looks even remotely like the random sequences produced in nature. However, with some careful tuning, they can be made to approximate such sequences. Of course, it is impossible to produce something truly random on a computer. As John von Neumann states, 'Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin'. Computers are deterministic, stuff goes in at one end, completely predictable operations occur inside, and different stuff comes out the other end, a principle that includes a notion that is fundamental to Digital Signal Processing (DSP) and computing in general, namely, 'rubbish in given rubbish out'. Put the same data into two identical computers, and the same data comes out of both of them (most of the time!).

A computer can only be in a finite number of states (a large finite number, but a finite number nonetheless), and the data that comes out will always be a deterministic function of the data that went in and the computer's current state. This means that any random number generator on a computer (at least, on a finite-state machine) is, by definition, periodic. Anything that is periodic is, by definition, predictable and can not therefore be random. A true random number generator requires some random input; a computer can

not provide this.

3.3.1 Pseudo Random Sequences

The best a computer can produce is a pseudo random sequence generator. Many attempts have been made to define a pseudo random sequence formally and in this section, a general overview is given of these attempts. A pseudo random sequence is one that looks random. The sequence's period should be long enough so that a finite sequence of reasonable length - that is, one that is actually used - is not periodic. If for example, a billion random bits are required, then a random sequence generator should not be chosen that repeats after only sixteen thousand bits. These relatively short non-periodic sequences should be as indistinguishable as possible from random sequences. For example, they should have about the same number of ones and zeros, about half the runs (sequences of the same bit) should be of length one, one quarter of length two, one eighth of length three, and so on. In addition, they should not be compressible. The distribution of run lengths for zeros and ones should be the same. These properties can be empirically measured and then compared with statistical expectations.

A sequence generator is pseudo random if it has the following properties: It looks random, which means that it passes all the statistical tests of randomness that we can find. Considerable effort has gone into producing good pseudo random sequences on a computer. Discussions of generators abound in the literature, along with various tests of randomness. All of these generators are periodic (there is no exception); but with potential periods of 2^{256} bits and higher, they can be used for the largest applications. The problem with all pseudo random sequences is the correlations that result from their

inevitable periodicity. Every pseudo random sequence generator will produce them if they are used extensively. A non periodic pseudo random sequence must have the property that it is unpredictable. It must be computationally non-feasible to predict what the next random bit will be, given complete knowledge of the algorithm or hardware generating the sequence and all of the previous bits in the stream.

3.3.2 Real Random Sequences

Is there such a thing as randomness? What is a random sequence? How do you know if a sequence is random? Is for example '101110100' more random than '101010101'? Quantum mechanics tells us that there is honest-to-goodness randomness in the real world but can we preserve that randomness in the deterministic world of computer chips and finite-state machines? Philosophy aside, a sequence generator is really random if it has the following additional property: It cannot be reliably reproduced. If the sequence generator is run twice with the exact same input (at least as exact as computationally possible), then the sequences are completely unrelated; their cross correlation function is effectively zero. This property is not usually possible to produce on a finite state machine and for some applications of random number sequences, is not desirable, as in cryptography for example. Thus, we refer to those processes that produce number streams which look random (and passes appropriate statistical tests) and are unpredictable as Pseudo Random Number Generators (PRNG).

3.3.3 Pseudo Random Number Generators

The performance of many DSP algorithms depends on the degree of noise present in the signal and because many types of DSP algorithms are sensitive to noise, it is important to test their behaviour in the presence of noise. This is usually done by synthesizing noise signals which is accomplished using pseudo random number generators. Random numbers are not numbers generated by a random process but are numbers generated by a completely deterministic arithmetic process. The resulting set of numbers may have various statistical properties which together are called randomness. A typical mechanism for generating random numbers is via the iterative process defined by

$$x_{n+1} = (ax_n + b) \bmod P, \quad n \geq 0$$

which produces an integer number stream in the range $[0, P]$ and is known as the Linear Congruential Generator (LCG) [52]. Here, the modular function mod operates in such a way as to output the remainder from the division of $ax_n + b$ by P, e.g.

$$23 \bmod 7 = 2 \quad \text{and} \quad 6 \bmod 8 = 6.$$

By convention $a \bmod 0 = a$ and $a \bmod b$ has the same sign as b. The reason for using modular arithmetic is because modular based functions tend to behave more erratically than conventional functions. For example consider the function $y = 2^x$ and the function $y = 2^x \bmod 13$ for example. The table below illustrates the difference between the output of these two function.

x	1	2	3	4	5	6	7	8
2^x	2	4	8	16	32	64	128	256
$2^x \bmod 13$	2	4	8	3	6	12	11	9

This approach to creating random sequences was first introduced by D H Lehmer in 1949. The values of the parameters are constrained as follows: $0 < a < P$, $0 \leq b < P$ and $0 \leq x_0 < P$. The essential point to understand when employing this method, is that not all values of the four parameters (a, b, x_0 and P) produce sequences that pass all the tests for randomness. Further, all such generators eventually repeat themselves cyclically, the length of this cycle (the period) being at most P. When $b = 0$, the algorithm, is faster and referred to as the multiplicity congruential method and many authors refer to mixed congruential methods when $b \neq 0$.

An initial value or seed x_0 is repeatedly multiplied by a and added to b, each product being reduced by modulo P. The element x_0 is commonly referred to as the seed. For example, suppose we let $a = 13$, $b = 0$, $P = 100$ and $x_0 = 1$; we will then generate the following sequence of two digit numbers

1, 13, 69, 97, 61, 93, 09, 17, 21, 73, 49, 37, 81, 53, 89, 57, 41, ...

For certain choices of a and P, the resulting sequence $x_0, x_1, x_2, ...$ is fairly evenly distributed over $(0, P)$ and contains the expected number of upward and downward double runs (e.g. 13, 69, 97) and triple runs (e.g. 9,17,21,73) and agrees with other predictions of probability theory. The values of a and P can vary and good choices are required to obtain runs that are statistically acceptable and have long cycle lengths, i.e. produce a long stream of numbers before the stream is repeated. For example, suppose we choose $a = 7$, $b =$

12, $P = 30$ and $x_0 = 0$, then the following sequence is generated

$$0,\ 12,\ 16,\ 4,\ 10,\ 22,\ 16,\ 4,\ 10,\ 22,\ 16,\ 4,\ \ldots$$

Here, after the first three digits, the sequence repeats the digits 4, 10, 22, 16; the 'cycle length' of the number generator is very short. To improve the cycle length, the value of P should be a prime number whose 'size' is close to that of the word length of the computer. The reason for using a prime number is that it is divisible by only 1 or itself. Hence, the modulo operation will always produce an output which is distinct from one element to the next. Many prime numbers are of the form $2^n - 1$ where n is an integer (Mersenne prime numbers and not for any value of n). A typical example of a Mersenne prime number is given by $2^{31} - 1 = 2147483648$. Values of the multiplier a vary considerably from one application to the next and include values such as 7^5 or 7^7 for example.

For long periods, P must be large. The other factor to be considered in choosing P is the speed of the algorithm. Computing the next number in the sequence requires division by P and hence a convenient choice is the word size of the computer. Perhaps the most subtle reasoning involves the choice of the multiplier a such that a cycle of period of maximum length is obtained. However, a long period is not the sole criterion that must be satisfied. For example, a = b = 1, gives a sequence which has a maximum period P but is anything but random. It is always possible to obtain the maximum period but a satisfactory sequence is not always attained. When P is the product of distinct primes only $a = 1$ will produce a full period, but when P is divisible by a high power of some prime, there is considerable latitude in the choice of a.

There are a few other important rules for optimising the performance of a

random number generator using the linear congruential method in terms of developing sensible choices for a, b and P, these include:

- b is relatively prime to P.

- $a - 1$ is a multiple of every prime dividing P.

- $a - 1$ is a multiple of 4 if P is a multiple of 4.

These conditions allow a linear sequence to have a period of length P.

Random number generators are often designed to produce a floating point number stream in the range $[0, 1]$. This can be achieved by normalising the integer stream after the random integer stream has been computed. A typical example of a random number generator is given below using pseudo code:

initialise seed
compute random integers
compute maximum output value
divide by maximum value to normalise

Here, the first loop computes the random integer stream using the LCG, the second loop computes the maximum value of the array and the third loop nomalizes it so that on output, the random number stream consists of floating point numbers (to single or double precision) between 0 and 1 inclusively. The seed is typically a relatively long integer which is determined by the user. The exact value of the seed should not change the statistics of the output, but it will change the numerical values of the output array. These values can only be reproduced using the same seed, i.e. such pseudo random number

generators do not satisfy the property that their outputs cannot be reliably reproduced.

The output of such a generator is good enough for many simulation type applications. There are a few simple guidelines to follow when using such random number generators:

(i) Make sure that the program calls the generator's initialization routine before it calls the generator.

(ii) Use initial values that are 'somewhat random', i.e. have a good mixture of bits. For example 2731774 and 10293082 are 'safer' than 1 or 4096 (or some other power of two).

(iii) Note that two similar seeds (e.g. 23612 and 23613) may produce sequences that are correlated. Thus, for example, avoid initialising generators on different processors or different runs by just using the processor number or the run numbers as the seed.

A typical C function for computing uniform random noise in the range 0 to 1 is included in the accompanying CD.

In addition to the standard linear congruential generator discussed so far, a number of 'variations on a theme' can be considered such as the iteration

$$x_i = (a_1 x_{i-1}^2 + a_2 x_{i-1} + a_3) \bmod P$$

or

$$x_i = (a_1 x_{i-1}^3 + a_2 x_{i-1}^2 + a_3 x_{i-1} + a_4) \bmod P$$

and so on where a_n are predefined (integer) numbers and P is a prime.

3.3.4 Shuffling

A relatively simple method that further randomizes the output of a PRNG is to shuffle the values with a temporary storage. We first initialize an array x_i, $i = 1, 2, ..., N$ with random numbers from the random number generator given above for example. The last integer random number computed x_N is then set to M say. To create the next random sequence y_i, we apply the following process:

```
for i=1 to N, do:
    j=1+int(N*M)
      y(i)=x(j)
      M=x(i)
```

3.4 Additive Generators

An alternative solution to random number generation which creates very long cycles of values is based on additive generators. A typical algorithm commences by initialising an array x_i with random numbers (not all of which are even) so that we can consider the initial state of the generator to be $x_1, x_2, x_3, ...$ We then apply

$$x_i = (x_{i-a} + x_{i-b} + ... + x_{i-m}) \bmod 2^n$$

where $a, b, ..., m$ and n are assigned integers. An example of this PRNG is the 'Fish generator' given by

$$x_i = (x_{i-55} + x_{i-24}) \bmod 2^{32}.$$

This approach to pseudo random number generation is fast as no multiplication operations (e.g. ax_i) are required. The period of the sequence of random numbers is also very large and of the order of $2^f(2^{55}-1)$ where $0 \leq f \leq n$.

A further example is the linear feedback shift register given by

$$x_n = (c_1 x_{n-1} + c_2 x_{n-2} + c_m x_{n-m}) \bmod 2^k$$

which, for specific values of $c_1, c_2, ...c_m$ has a cycle length of 2^k.

3.4.1 PRNG and Cryptography

In cryptography, pseudo random number generation plays a central role as does modular arithmetic in general. One of the principal goals in cryptography is to design random number generators that provide outputs (random number streams) where no element can be predicted from the preceding elements given complete knowledge of the algorithm. Another important feature is to produce generators that have long cycle lengths. A further useful feature, is to ensure that the Entropy of the random number sequence is a maximum, i.e. that the histogram of the number stream is uniform. Finally, the use of modular arithmetic in the development of encryption algorithms tends to provide functions which are not invertible. They are one-way functions that can only be used to reproduce a specific (random) sequence of numbers from the same initial condition.

The basic idea in cryptography is to convert a plaintext file to a ciphertext file using a key that is used as a seed for the PRNG. A plaintext file is converted to a stream of integer numbers using ASCII (American Standard Code for Information Interchange) conversion. For example, suppose we wish

to encrypt a name Blackledge for which the ASCII[1] decimal integer stream or vector is

$$\mathbf{f} = (66, 108, 97, 99, 107, 108, 101, 100, 103, 101).$$

Suppose we now use the linear congruential PRNG defined by

$$n_{i+1} = an_i \bmod P$$

where $a = 13$, $P = 131$ and let the seed be 250659, i.e. $n_0 = 250659$. The output of this iteration is

$$\mathbf{n} = (73, 32, 23, 37, 88, 96, 69, 111, 2, 26).$$

If we now add the two vectors together, we can generate the cipher stream

$$\mathbf{c} = \mathbf{f} + \mathbf{n} = (139, 140, 120, 136, 195, 204, 170, 211, 105, 127).$$

Clearly, provided the recipient of this number stream has access to the same algorithm (including the values of the parameters a and P) and crucially to the same seed, the vector $\mathbf{n}$ can be regenerated and $\mathbf{f}$ obtained from $\mathbf{c}$ by subtracting $\mathbf{n}$ from $\mathbf{c}$. This process can of course be accomplished using binary streams where the binary stream representation of the plaintext $\mathbf{f}_b$ and that of the random stream $\mathbf{n}_b$ say are used to generate the cipher binary stream $\mathbf{c}_b$ via the process

$$\mathbf{c}_b = \mathbf{n}_b \oplus \mathbf{f}_b$$

where $\oplus$ denotes the XOR operation. Restoration of the plaintext is then accomplished via the operation

$$\mathbf{f}_b = \mathbf{n}_b \oplus \mathbf{c}_b = \mathbf{n}_b \oplus \mathbf{n}_b \oplus \mathbf{f}_b.$$

[1]Any code can be used.

Clearly, the process above is just an example of digital signal processing in which the information contained in a signal **f** (i.e. the plaintext) is 'scrambled' by introducing additive noise. Here, the seed plays the part of a key that is utilised for the process of encryption and decryption; a form of encryption that is commonly known as symmetric encryption in which the key is a private key known only to the sender and recipient of the encrypted message. Given that the algorithm used to generate the random number stream is publically available (together with the parameters it uses which are typically 'hard-wired' in order to provide a random field pattern with a long cycle length), the problem is how to securely exchange the key to the recipient of the encrypted message so that decryption can take place. If the key is particular to a specific communication and is used once and once only for this communication (other communications being encrypted using other keys), then the processes is known as a one-time pad, because the key is only used once. Simple though it is, this process is not open to attack. In other words, no form of cryptanalysis will provide a way of deciphering the encrypted message. The problem is how to exchange the keys in a way that is secure and thus, solutions to the key exchange problem are paramount in symmetric encryption. A well known historical example of this problem involved the distribution of the keys used to initialize the Enigma cipher used by the German forces during the Second World War. The Enigma machine (which was named after Sir Edward Elgar's composition, the 'Enigma Variations') was essentially an electromechanical PRNG in which the the seed was specified using a plug board and a set of three (and later four) rotors whose initial positions could be changed. These settings were effectively equivalent to a password or a private key as used today. For a period of time and using a very simplistic and rather exaggerated explanation, the German land

forces sometimes communicated the password used on a particular day (and at a set time) by radio transmission using standard Morse code. This transmission was sometimes repeated in order to give the recipient(s) multiple opportunity to receive the key(s) accurately. Worse still, in some rare but important cases, the passwords were composed of simple names (of some of the politicians at the time for example) or phrases. Thus, in many cases, a simple password consisting of a well known name or phrase was transmitted a number of times sequentially leading to near perfect temporal correlation of the initial transmission. This was a phenomenally irresponsible way of using the Enigma system. In today's environment, it is like choosing a password for your personal computer which is a simple and possibly well known name (of the your boss or chief executive for example) or phrase that is easily remembered, shouting it out a number of times to your colleagues in a open plan office and then wondering why everyone seems to know something about your private life! In this sense, the ability for the British war time intelligence services to decipher the German land forces communications is self-evident. The use of Enigma by the German naval forces (in particular, the U-boat fleet) was far more secure in that the password used from one day to the next was based on a code book provided to the users prior to departure from base. Thus, no transmission of the daily passwords was required and, if not for a lucky break, in which one of these code books was recovered in tact by a British destroyer (HMS Bulldog) from a damaged U-boat, breaking the Enigma naval transmissions under their time variant code-book protocol would have been effectively impossible. Although the Enigma story has many facets to those discussed here, a careful study of this historically intriguing technology reveals that the breaking of Enigma had as much to do with German incompetency and some bad luck as it did with British in-

telligence coupled with some good luck. Thus is the reality of how random events (or lucky breaks to some) of the past can effect the outcome of the future!

The discussion above has been used by way of an example to highlight the problem of exchanging keys when applying a symmetric encryption scheme. It also provides an example of how, in addition to developing the technology for encryption, it is imperative to develop appropriate protocols and procedures for using it effectively with the aim of reducing inevitable human error, one of the underlying principles being, the elimination of any form of temporal correlation. Another fundamental principle which has been demonstrated time and again throughout the history of cryptology is that although improvements in methods and technology are to be welcomed, information security is ultimately to do with cultivating the 'right state of mind' and that part of this state should include a healthy respect for the enemy.

3.4.2 Gaussian Random Number Generation

The generation of Gaussian random numbers which are taken to conform to the distribution

$$P(y) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{y^2}{2\sigma^2}\right)$$

where σ is the standard deviation, is important in the analysis of real signals because many signals are characterized by additive noise that is Gaussian or normally distributed. In cryptography, this result has applications in modeling transmission noise for example.

The method is based on the Box-Muller transform which, in effect transforms uniformly distributed deviates into Gaussian distributed deviates. The basic

idea is to first create two uniform deviates x_1 and x_2 say on $(0,1)$. Now, assume that we wish to create two values y_1 and y_2 which conform to the Gaussian probability distribution function

$$P(y) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right)$$

which has a zero mean and a standard deviation of 1. We can then consider a relationship between x_1, x_2, y_1 and y_2 of the form

$$y_1 = \sqrt{-2\ln x_1}\cos(2\pi x_2) \quad \text{and} \quad y_2 = \sqrt{-2\ln x_1}\cos(2\pi x_2)$$

or equivalently

$$x_1 = \exp\left[-\frac{1}{2}(y_1^2 + y_2^2)\right] \quad \text{and} \quad x_2 = \frac{1}{2\pi}\tan^{-1}\frac{y_2}{y_1}.$$

Further, suppose we let

$$\sin(2\pi x_2) = \frac{v_1}{R} \quad \text{and} \quad \cos(2\pi x_2) = \frac{v_2}{R}.$$

Then $R^2 = v_1^2 + v_2^2$ and if we set $x_1 = R^2$, then we obtain the result that

$$y_1 = v_1\sqrt{\frac{-2\ln r}{r}} \quad \text{and} \quad y_2 = v_2\sqrt{\frac{-2\ln r}{r}}$$

where $r = R^2$. Here, v_1 and v_2 are uniform deviates on $(0,1)$ such that $r \leq 1$.

Note, that if we compute the joint probability distribution of y_1 and y_2, then

$$p(y_1, y_2)dy_1 dy_2 = p(x_1, x_2)\left|\frac{\partial(x_1, x_2)}{\partial(y_1, y_2}\right| dy_1 dy_2$$

where the Jacobian determinant is given by

$$\left|\frac{\partial(x_1, x_2)}{\partial(y_1, y_2}\right| = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{y_1^2}{2}\right)\right]\left[\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{y_2^2}{2}\right)\right]$$

which shows that y_1 and y_2 are independent and that the method creates two Gaussian deviates from two uniformly random deviates as required. Thus, an algorithm for implementing this method is as follows:

repeat

v_1=RAND()

v_2=RAND()

$r = v_1^2 + v_2^2$

until $r \leq 1$

$$y_1 = v_1 \sqrt{\frac{-2 \ln r}{r}}$$
$$y_2 = v_2 \sqrt{\frac{-2 \ln r}{r}}$$

where the function RAND() is taken to output a uniform random deviate using the linear congruential method discussed earlier.

The following C code provides a function GNOISE that outputs a Gaussian random field using the method discussed above. The process generates two arrays of uniform deviates (with different seeds) using the function UNOISE and feeds these deviates as pairs into the Box-Muller transform.

```
#include<math.h>

void UNOISE( float s[], int n, long int seed );

void GNOISE( float s[], int n, long int seed )
/* FUNCTION: Generates an n size array s of Gaussian distributed   */
/*           noise with zero mean and a standard deviation of 1.   */
```

```
{
  int i, k, nn;
  float r, fac, v1, v2, *x1, *x2;

/*Allocate internal work space.*/
x1 = (float *) calloc( n+1, sizeof( float ) );
x2 = (float *) calloc( n+1, sizeof( float ) );

nn=n/2;
UNOISE(x1,nn,seed);/*Generate uniform deviates.*/
seed=seed+3565365; /*Add randomly chosen integer to seed.*/
UNOISE(x2,nn,seed);/*Generate new set of uniform deviates.*/

k=0;
for(i=1; i<=nn; i++)
{
v1 = 2.0 * x1[i] - 1.0; /* -1 < v1 < 1 */
v2 = 2.0 * x2[i] - 1.0; /* -1 < v2 < 1 */

r  = pow( v1, 2 ) + pow( v2, 2 );
r = r/2;/* r<=1 */

/* Apply the Box-Muller transform */
fac=sqrt( (double) -2.0 * log(r)/r);

/*Write to output array.*/
```

```
        s[k]=v1*fac;
        s[k+1]=v2*fac;
        k=k+2;
        }
}
```

3.5 Blum-Blum Shub

Blum-Blum Shub (BBS) [10] generator is one of the cryptographically strongest pseudo random number generator available. The generator works as follows: First we choose two large prime numbers, p and q, that both have a remainder of 3 when divided by 4. That is: $p \equiv q \equiv 3 (\text{mod } 4)$. This is equivalent to

$$(p \bmod 4) = (q \bmod 4) = 3$$

.

We then apply th following iteration

$$x_{i+1} = x_i^2 \bmod (pq)$$

The BBS is referred to as a cyptographically secure pseudorandom bit generator. A pseudorandom bit generator is said to pass the next-bit test if there is not a polynomial-time algorithm that, on input of the first k bits of an output sequence, can predict the $(k+1)^{st}$ bit with a probability significantly greater than 1/2. In other words, given the first k bits of the sequence, there is not a practical algorithm that can even allow us to state that the next bit will be 1 (or 0) with a probability greater than 1/2. The output of the BBS is unpredictable. The security of the BBS is based on the difficulty of

factoring n which can be made public, however, unless the cryptanalyst can factor n, it is effectively impossible for him/her to predict the output of the generator. This generator is unpredictable to the left and unpredictable to the right. This means that for a given sequence generated by the generator, a cryptanalyst cannot predict the next or previous bit in the sequence.

Randomness and Fairness - The concept of fairness is significant when it comes to generating random numbers. By flipping a coin for example, we would expect the number of times it lands on head to be equal to that of tails. This is the essence of a fair coin.

Blum Blum Shub generator produces cryptographically secure random numbers. By running the generator at a certain execution cycle, it can be shown that the distribution is normal, even though some of the numbers may be repeated, but the pattern of repetition is the same.

Figure 3.1 shows the distribution of random numbers run over an interval of 1000. The actual random numbers selected is 10000. The histogram displays what may be termed as uniform distribution. From the graph we can see that the numbers bewteen 4000 and 5000 have the highest distribution and that between 9000 and 10000 have the lowest. This is to be expected because the numbers are generated randomly and are therefore subject to deviation. If the generator is run over a long period, the average result of the random numbers will represent normal distribution. The model does generate random numbers fairly.

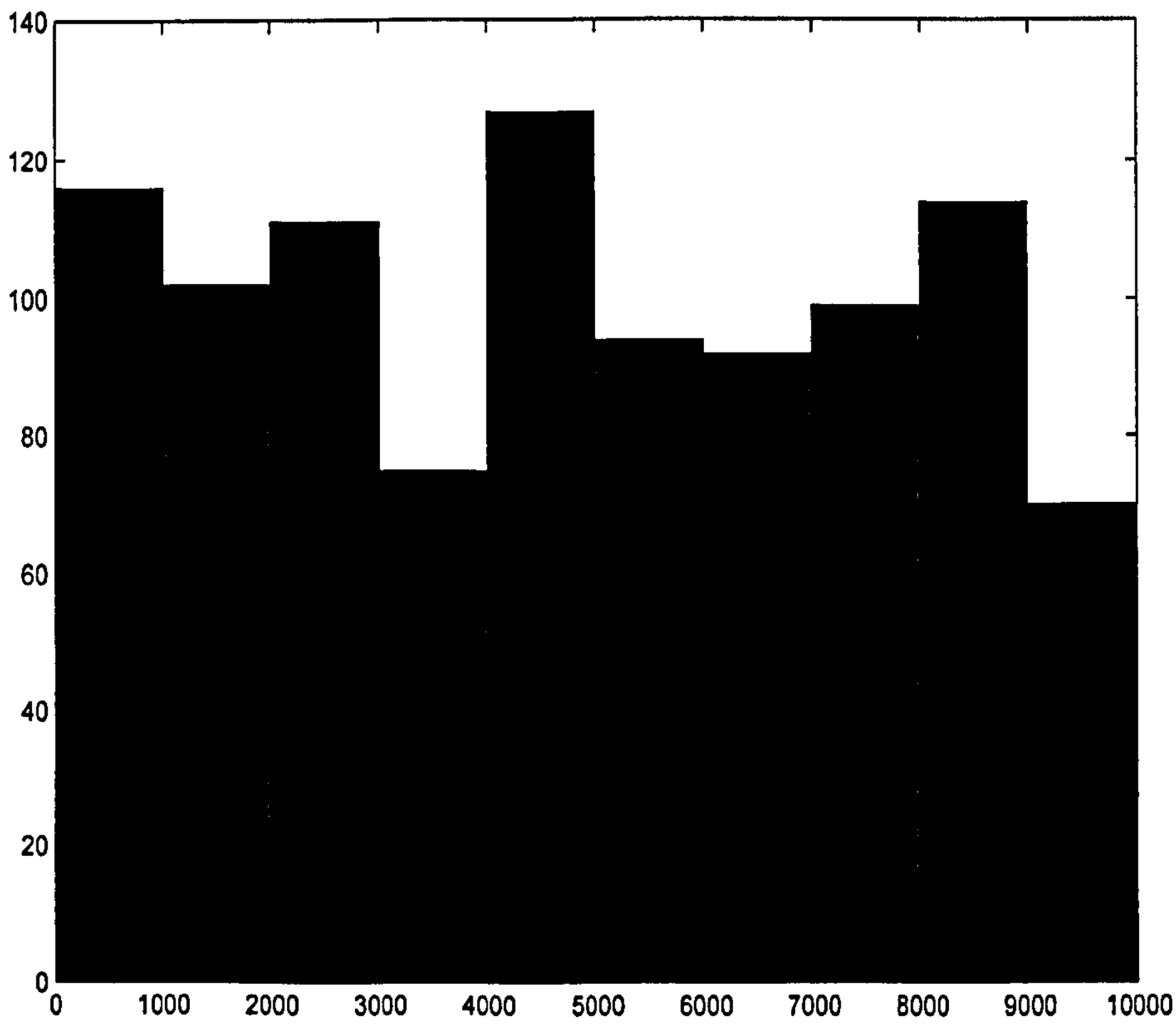


Figure 3.1: Blum Blum Shub generator displaying normal distribution of random numbers

3.6 HotBits using Radioactive Decay

HotBits [93] random numbers are generated using quantum mechanical laws of nature. The numbers are generated by timing successive pairs of radioactive decays detected my Geiger-Muller tube interfaced to a computer. The Hotbits radiation source is Krypton-85. It works by utilising the material's physical properties which results radioactive emissions. Details of the process is beyond the scope of this research but the method is based on the beta decay process $\|85_{K}r- > \|85_{R}b + \beta^{-} + \gamma$. In order to be able to utilise the output, a user needs to contact the server, where upon the output is transmitted directly to his/her PC using the web. These numbers might be truly random, as they are produced by nature; however, they are not secure enough to be used for cryptography, since a third party is involved.

3.7 RSA (Rivest Shamir and Adleman)

RSA gets its name after the three inventors, Rivest, Shamir and Adleman who developed the generator in the mid 1970s.[2] It has since withstood years of extensive cryptanalysis. However, because to date, cryptanalysis has neither proved nor disproved RSA's security, it does suggest a high confidence level in the algorithm.

RSA [38] gets its security from the difficulty of factoring large numbers [70].

[2]There are some claims that the method was first developed at GCHQ in England and then re-invented (or otherwise) by Rivest, Shamir and Adleman; the idea was not published openly by GCHQ, only as an internal report that, to date, has not been opened to public scrutiny.

The public and private keys are functions of a pair of large (100 to 200 digits or even larger) prime numbers. Recovering the plaintext from the public key and the cipher text is conjectured to be equivalent to factoring the product of the two primes. According to Koblitz (1987, p88), 'The success of the so-called RSA cryptosystem, which is one of the oldest and most popular public key cryptosystems, is based on tremendous difficulty of factoring'. The essence of the RSA method is outlined below.

Algorithm for key generation

1. Generate two large random (and distinct) primes p and q, each of roughly the same size.
2. Compute $n = pq$ and $\phi = (p-1)(q-1)$.
3. Select random integer $e, 1 < e < \phi$, such that $gcd(e, \phi) = 1$.
4. Use the extended Eucledian algorithm to compute the unique integer $d, 1 < d < \phi$, such that $ed \equiv 1(mod\phi)$.
5. For use A, the public key is (n, e) and the private key is d where e is the encryption component, d is the decryption component and n is the modulus.

Encryption

User B encrypts a message for A, which A decrypts. This is based on the following:

1. Obtain $A's$ authentic public key (n, e).
2. Represent the message as an integer m in the interval $[0, n-1]$
3. Compute $c = m^e \bmod n$

4. Send the ciphertext c to A.

Decryption

To recover plaintext m from c. A should:

Use the private key d to recover $m = c^d \bmod n$.

Security of RSA

With the current computing power, factoring n to calculate m from c and e is not within reach. By using more powerful computers for factorization, the same computers are also used to compute large values of primes. It turns out that the increase in computing power may not necessarily pave way to breaking the RSA. It is conceivable that an entirely different way to cryptanalyse RSA might be discovered. However, if this new way allows the cryptanalyst to deduce d, it could also be a new way to factor large numbers. It is possible to attack RSA by guessing the value of $(p-1)(p-1)$. This attack is no easier than factoring n. Factoring n is the most obvious means of attack [81]. Any adversary will have the public key, e, and the modulus n. To find the decryption key, d, the attacker has to factor n. It is possible for a cryptanalyst to try every possible d but this brute force approach is less efficient than trying to factor n.

There have been a number of attacks on RSA, in conclusion to his analysis on RSA attacks, Boneh [11] concludes that: 'The attacks discovered so far mainly illustrate the pitfalls to be avoided when implementing RSA.' So

basically, even though RSA can be attacked, it can can still be considered secure, when used properly. In order to ensure the strength of the cipher, RSA runs factoring challenges [80] on its websites. The challenge is hard to pass, because of the high rewards (around £100,000) and also hard to take on.

3.8 DES

The Data Encryption Standard (DES), known as the Data Encryption Algorithm (DEA) by ANSI and the DEA-1 by the ISO, has been the world wide standard for over 20 years. It is a symmetric (private key) system that has held up remarkably well against years of cryptanalysis.

In order for it to be acceptable as the standard encryption algorithm, DES algorithm had to meet the following:

- provide a high level of security;
- must be completely specified and easy to understand;
- the security of the algorithm must reside in the key (the security should not depend on the secrecy of the algorithm);
- the algorithm must be available to all users;
- the algorithm must be acceptable for use in diverse applications;
- it must be economically implementable in electronic devices;
- it must be efficient to use;
- it must be able to be validated;

- it must be exportable.

3.8.1 Outline of DES

DES is a block cipher using 64-bit blocks. The key length is 56-bit. It is usually expressed as a 64-bit number, but every 8^{th}-bit is used for parity checking and is ignored. All security of DES rests within the keys.

DES operates on a 64-bit block of plaintext. After the initial permutation, the block is split into two halves, each 32-bits long. Then there are 16 rounds of identical operations, called function f, in which the data are combined with the key. After the 16^{th}-round, the two halves are joined, and a final permutation (the inverse of the initial permutation) completes the algorithm.

In each round the key bits are shifted, and then 48-bits are selected from the 56-bits of the key. The right half of the data is expanded to 48-bits via an expansion permutation, combined with 48-bits of the shifted and permuted key via an XOR, sent through 8 'S-boxes' producing 32 new bits, and permuted again. These four operations make up function f. The output of function f is then combined with the left half via another XOR. The result of these operations becomes the new right half; the old right half becomes the new left half. These operations are repeated 16 times, making 16 rounds of DES. In effect, DES is based on randomization of the data via the process of shuffling.

If B_i is the result of the iteration, L_i and R_i are left and right halves of B_i, K_i is the 48-bit key for round i, and f is the function that does all the substituting and permuting and XORing with the key, then a round is as follows:

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

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

3.8.2 DES Algorithm

DES is a Feistel cipher which processes plaintext blocks of $n = 64$ bits, producing 64-bit ciphertext blocks. The effective size of the secret key K is $k = 56$ bits; more precisely, the input key K is specified as a 64-bit key, 8 bits of which (bits 8, 16, ... 64) may be used as parity bits. The 2^{56} keys implemented (at most) 2^{56} of 2^{64} possible bijections on 64-bit blocks.

Encryption proceeds in 16 stages of rounds. From the input key K, sixteen 48-bit subkeys K_i are generated, one for each round. Within each round, eight fixed carefully selected 6-to-4 bit substitution mappings (S-boxes) S_i, collectively denote S, are used. The 64-bit plaintext is divided into 32-bit halves L_0 and R_0. Each round is functionally equivalent, taking 32-bit input L_{i-1} and R_{i-1} from a previous round and producing 32-bits output. Hence E is a fixed expression permutation, mapping $Ri - 1$ from 32 to 48-bits (all bits are used once, some are used twice). P is another permutation on 32-bits. An initial permutation (IP) precedes the first round; following the last round, the left and right halves are exchanged and, finally, the resulting string is bit permuted by the inverse of IP. Decryption involves the same key and algorithm, but with subkeys applied to the internal rounds in the reverse order. A simplified view is that the right half of each round (after expanding a 32-bit input of 8 characters of 6-bits each) carries out key-dependent substitutions on each of the 8 characters, then uses a fixed bit

transposition to redistribute the bits of the resulting characters to produce a 32-bit output.

The DES algorithm, on a step by step basis is given below.

Input: plaintext $m_1...m_{64}$; 64-bit key $K = k_1...k_{64}$ (includes 8 parity bits)

Output: 64-bit ciphertext block $C = c_1...c_{64}$

1. (key schedule) Compute sixteen 48-bit round keys K_i from K. (using DES key schedule algorithm - see below)

2. $(L_0, R_0) \leftarrow IP(m_1 m_2...m_{64})$

3. (16 rounds) for i from 1 to 16, compute L_i and R_i computing $f(R_{i-1}, K_i) = P(S(E(R_{i-1i}))$ as follows:

(a) Expand $R_{i-1} = r_1 r_2...r_3 2$ from 32 to 48 bits
$T \leftarrow E(R_{i-1})$. (Thus $T = r_{32} r_1 r_2...r_{32} r_1$.) (b) $T\prime_i$. Represent $T\prime$ as eight 6-bit character strings: $(B_1, ..., B8) = T\prime$.
(c) $T\prime\prime \leftarrow (S_1(B_1), S_2(B_2), ..., S_s(B_s))$. Here $S_i(B_i)$ maps $B_i = b_1 b_2...b_6$
(d) $T\prime\prime\prime \leftarrow P(T\prime\prime)$

4. $b_1 b_2...b_6 4 \leftarrow (R_1 6, L_1 6)$. (Exchange final blocks $L_1 6, R_1 6$).

5. $C \leftarrow IP^{-1}(b_1 b_2 ... b_6 4)$.

DES Key Schedule Algorithm

Input: 64-bit key $K = k_1 ... k_6 4$ (including 8 odd-parity bits).

Output: sixteen 48-bit keys $K_i, 1 \leq i \leq 16$.

1. Define $v_i, 1 \leq i \leq 16$ as follows: $v_i = 1$ for $i \in 1, 2, 9, 16; v_i = 2$ otherwise. (These are left-shift values for 28-bit circular rotations below.)

2. $T \leftarrow PCI(K)$; represent T as 28-bit halves (C_0, D_0).

3. For i from 1 to 16 compute K_i as follows: $C_i \leftarrow (C_{i-1} \leftarrow v_i), D_i \leftarrow (D_{i-1} \leftarrow v_i), K_i \leftarrow PC2(C_i, D_i)$

3.8.3 Security of DES

Since its invention in the 1970s, the security of DES has been studied intensively. Special techniques such as differential and linear cyptanalysis have been used to attack DES, but the most successful attack has been an exhaustive search of the key space. With special hardware of large networks and workstations, it is now possible to decrypt DES ciphertexts in a few days or even hours. In addition, the Electronic Frontier Foundation (EFF)

have sponsored the development of a crypto chip named *Deep Crack* that can process 88 billion DES keys per second and has successfully cracked 56-bit DES in less that 3 days (J Buchmann 2001). In the paper *Six Ways to Break DES* [48], Junod outlines various methods that can be used to break the encryption.

Today, DES can only be considered secure if triple encryption is used. In this context, it is important to know that DES is not a group. This means that for the two DES keys k_1 and k_2 there is, in general, not a third DES k_3 such that $\text{DES}(k_1) \circ \text{DES}(k_2) = \text{DES}(k_3)$. If DES were a group, then multiple encryption would not lead to increased security. In fact, the subgroup that the DES encryption permutations generate in the permutation group $S_{64!}$ is at least of order the order of 10^{2499}. Hence, triple encryption DES or DES3 has become the preferred standard for symmetric encryption systems world wide since the 2001.

3.9 Rijndael

Rijndael is an iterated block cipher with a variable block length and a variable key length. The block length and key length can be independently specified at 128, 192, and 256 bits. The Rijndael Block Cipher was selected by the National Institute of Science and Technology (NIST), mainly because DES was an aging standard and no longer addresses today's needs for strong encryption.

The designers of the Rijndael Cipher had the following criteria taken into account:

- resistance against all known attacks;
- speed and code of compactness on a wide range of platforms;
- design simplicity.

In most ciphers, the round transformation has the Feistel Structure. In this structure, typically, part of the bits of the intermediate state are simply transposed unchanged to another position. The round transformation of Rijndael does *not* have the Feistel structure. Instead, the round transformation is composed of three distinct invertible uniform transformations called *layers*. By *uniform*, we mean that every bit of the state is treated in a similar way.

The specific choices for the different layers are for a large part based on the application of a Wide Trail Strategy, a design method used to provide resistance against linear and differential cryptanalysis. In the Wide Trail Strategy, every layer has its own function:

The linear mixing layer: guarantees high diffusion of multiple rounds.
The non-linear layer: parallel application of S-boxes that have optimum worst-case non-linearity properties.
The key additional layer: a simple XOR of the round key to the intermediate State.

3.9.1 The State and the Cipher

The State can be pictured as a rectangular array of bytes. This array has four rows, the number of columns is denoted by Nb and is equal to the block

length divided by 32. The Cipher key is similarly pictured as a rectangular array with four rows. The number of columns of the Cipher Key is denoted by Nk and is equal to the key length divided by 32.

Encryption takes place using four different stages:

1. Substitute bytes: Uses S-box to perform byte by byte substitution of the block.

2. Shift rows: A simple permutation.

3. Mix Columns: Substitution over $GF(2^8)$.

4. Add round key: Bitwise XOR of current block and portion of the expanded key.

3.9.2 Hardware Implementation

The Rijndael cipher is suited for effective implementation on a wide range of processors with dedicated hardware. Below are some examples of 8- and 32-bit processors.

8-Bit Processors

On an 8-bit processor, Rijndael can be programmed by simply implementing the different component transformations. This is straight forward for RowShift and for the Round Key addition. The implementation of a ByteSub requires a table of 256 bytes. The Round Key addition, ByteSub and RowShift can be effectively combined and executed serially per State byte.

Indexing overhead is minimised by explicitly coding the operation for every State byte.

32-Bit Processors

The different steps of the round transformation can be combined in a single set of lookup tables, allowing for very fast implementations on processors with word lengths of 32 or above.

Hardware Suitability

The cipher is suited to be implemented in dedicated hardware. There are several trade-offs between area and speed that are possible. Because the implementation in software on general purpose processors is already very fast, the need for hardware implementations is usually limited to two specific cases:

(i) Extremely high speed chips with no area restrictions: the T tables can be hardwired and the EXORs can be conducted in parallel.

(ii) Compact co-processors on a Smart Card to speed up Rijindael execution: for this platform, typically, the S-box (or the complete MixColumn) operation can be hardwired.

3.9.3 The Inverse Cipher

In the table look-up implementation, it is essential that the only non-linear step (ByteSub) is the first transformation in a round that the rows are shifted before MixColumn is applied. In the Inverse of a round, the order of the transformations in the round is reversed and, consequently, the non-linear setup will end up being the last step of the Inverse round and the rows are shifted after the application of (the inverse of) MixColumn.

3.9.4 Strength of AES

This system is expected to perform strongly for all key lengths and block lengths defined. The most efficient key recovery attack for AES is exhaustive key search. This is the most efficient way of obtaining information from a given plaintext-ciphertext pairs. The expected effort of exhaustive key search depends on the length of the Cipher Key:

For a 16-byte key, 2^{127} applications of Rijndael;
For a 24-byte key, 2^{191} applications of Rijndael;
For a 32-byte key, 2^{255} applications of Rijndael.

The rationale for this is that a considerable safety margin is taken with respect to all known attacks. It is, however, impossible to make non-speculative statements on unknown matters.

3.9.5 Advantages and Limitations

Advantages

The cipher does not base its security or part of it on obscure and poorly understood interactions between arithmetic operations. The variable block lengths of 192 and 256 bits allow the construction of a collision-resistant iterated hash function using Rijndael as a compression function. The block length of 128 bits is not considered sufficient for this purpose nowadays.

Although the number of rounds is fixed in the specifications, it can be modified as a parameter to enhance security.

Limitations

In software, the cipher and its inverse make use of different codes and/or tables. In hardware, the inverse cipher can only partially re-use the circuitry that implements the cipher. Encryption is performed at the Add Round Key stage; this is the only stage in which the key is used. Thus, ciphering always begins with this round. The other three stages provide confusion, diffusion and non-linearity. Since the key is not used in these stages, no security is provided. The ciphering process can be viewed as alternating operations of XOR encryption (Add Round Key) of a block followed by scrambling of the block (the other three stages) followed by XOR encryption. This provides for efficiency and strong encryption.

3.10 Lucifer

Lucifer is generally considered to be the first civilian block cipher, developed in the 1970s based on work done by Horst Feistel [14]. A revised version of the algorithm was adapted as a FIPS (Federal Information Processing Standard) standard, the Data Encryption Standard (DES). It was chosen by the US National Bureau of Standards (NBS) after public invitation for submissions and some internal changes by NBS. DES was publicly released in 1976 and has been widely used ever since. Lucifer's S-boxes have 4-bit inputs and 4-bit outputs; the input of the S-boxes is the bit permuted output of the S-boxes of the previous round; the input of the S-boxes of the first round is the plaintext.

Using differential cryptanalysis against the initial version of Lucifer, Biham and Shamir showed that Lucifer, with 32-bit blocks and 8 rounds, can be broken with 40 chosen plaintexts and 2^{29} steps; the same attack can break Lucifer with 128-bit blocks and 8 rounds with 60 chosen plaintexts and 2^{53} steps. Lucifer has been around for a long time. It has now been succeeded by DES and AES and all of Lucifer's US patents have now expired.

3.11 FEAL

FEAL was designed in Japan by Shimizu and Miyaguchi from NTT, Japan [13] as a replacement to DES. It was originally built as a four-round cryptosystem with a 64-bit block size and a 64-bit key size. This was done in order to give high performance in software. However, soon a number of attacks against FEAL-4 were announced including one attack that required

only 20 chosen plaintexts. This led the designers to introduce a revised version, i.e. FEAL-N, where N denotes a number of rounds.

FEAL was designed for speed and simplicity, especially for software on 8-bit microprocessors (e.g. chipcards). It uses byte oriented operations (8-bit addition mod 256, 2-bit left rotation and XOR), avoids bit-permutations and table look-ups and offers small code size.

Basic Algorithm

Input: 64-bit plaintext $M = m_1...m_{64}$; 64-bit key $K = k_1...k_{64}$

Output: 64-bit ciphertext block $C = c_1...c_{64}$

1. (key schedule) Compute sixteen 16-bit subkeys K_i from K

2. Define $M_L = m_1...m_{32}$, $M_R = m_{33}...m_{64}$.

3. $(L_0, R_0) \leftarrow (M_L, M_R) \oplus ((K_8, K_9), (K_{10}, K_{11}))$. (XOR initial subkeys).

4. $R_0 \leftarrow R_0 \oplus L_0$

5. For i is 1 to 8 do: $L_i \leftarrow R_{i-1}, R_i \leftarrow L_{i-1} \oplus f(R_{i-1}, K_{i-1})$. [61]

6. $L_8 \leftarrow L_8 \oplus R_8$.

7. $(R_s, L_s) \leftarrow (R_s, L_s) \oplus ((K_12, K_13), (K_14, K_15))$. (XOR final subkeys.)

8. $C \leftarrow (R_s, L_s)$. (Note the order of the final blocks is exchanged.)

The same algorithm can be used for decryption, but with the key schedule reversed. Cryptanalysis is repoted in [7].

3.12 IDEA

IDEA works on 64-bit blocks. Developed in Zurich, Switzerland by Xuejia Lai and James Massey, it is generally regarded to be one of the best and most secure block algorithms available to the public today. It utilizes a 128-bit key and is designed to be resistant to differential cryptanalysis [83], [59].

While IDEA is not a Feistel cipher, decryption is carried out in the same manner as encryption once the decryption subkeys have been calculated from the encryption subkeys. The designers have taken great care in making a structure that is easily implemented in both software and hardware. The security of IDEA relies on the use of three incompatible types of arithmetic operations on 16-bit words: XOR, addition modulo 2^{16}, and multiplication modulo $2^{16} + 1$. Its speed in software can be compared to that of DES.

One of the principles during the design of IDEA was to facilitate analysis of its strength against differential cryptanalysis; IDEA is considered to be immune from differential cryptanalysis. In addition, no linear cryptanalytic attacks on IDEA have been reported and there is no known algebraic weakness in IDEA. The most significant cryptanalytic result is due to Daemen. He discovered a large class of 251 weak keys for which the use of such a key during encryption could be detected and the key recovered. However, since there are 2^{128} possible keys, this result has no impact on the practical security of the cipher for encryption. IDEA is generally considered secure and both the cipher development and its theoretical basis have been openly and

widely discussed.

3.13 Skipjack

Skipjack is the encryption algorithm contained in the Clipper chip [83], [12], and it was designed by the NSA. It uses an 80-bit key to encrypt 64-bit blocks of data. Skipjack can be more secure than DES, since it uses 80-bit keys with 32 rounds. By contrast, DES uses 56-bit keys with only 16 rounds.

Since its release in 1987, the Skipjack algorithm has remained secret and a number of cryptographers were suspicious of the fact. Some thought it might be insecure, others were not contented with the fact that NSA had inserted a trapdoor. The government, aware of such criticism, decided to invite a small group of independent cryptographers to examine the Skipjack algorithm. The cryptographers issued a report which stated that, although their study was too limited to reach a definitive conclusion, they nevertheless believed that Skipjack was secure. The following report was issued by the independent committee:

'Under the assumption that the cost of processsing power is halved every 18 months, it will be 36 years before the difficulty of breaking Skipjack by exhaustive search will be equal to the difficulty of breaking DES today. Thus, there is no significant risk that Skipjack will be broken by exhaustive search in the next 30-40 years.

There is no significant risk that Skipjack can be broken through a shortcut method of attack, including differential cryptanalysis. There are no weak keys; there is no complementation property. The experts, not having time to evaluate the algorithm to any great extent, instead evaluated NSA's own design and evaluation process.

The strength of Skipjack against cryptanalytic attack does not depend on the secrecy of the algorithm.'

In 1998 the US government decided to de-classify Skipjack.

3.14 GOST

GOST is a symmetric block cipher designed by the former government of Soviet Union [53], [21].It is a 64-bit block cipher with a 256-bit key. The iteration for the GOST algorithm is 32 rounds. To encrypt, a block is divided into two halves, left, L, and right, R. The sub-key for round i is K_i. A typical GOST round i is:

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

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

The right half and the i^{th} subkey are added to modulo 2^{32}. The output is then divided into 8 4-bit data blocks. Each block becomes the input to a different S-box. There is a total of eight S-boxes and thus, each four bits go to one S-box. Each S-box is a permutation of the numbers 0 to 15. For example, an S-box might look like:

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

The outputs of all eight S-boxes are combined into a 32-bit word. The word is then circular shifted 11 bits to the left. The result is XORed to the left half to become the new right half, and the right half becomes the new left half. This process is repeated 32 times.

The are some major differences between GOST and DES: [83]

- DES has a complicated procedure for generating subkeys from the keys. GOST has a very simple procedure.

- DES has a 56-bit key; GOST has a 256-bit key. If you add in the secret S-box permutations, GOST has a total of about 610 bits of secret information.

- The S-boxes in DES have 6-bit inputs and 4-bit outputs, the S-boxes in GOST have 4-bit inputs and outputs. Both algorithms have eight S-boxes, but an S-box in GOST is one-fourth the size of an S-box in DES.

- DES has an irregular permutation, called a P-box; GOST uses an 11-bit left circular shift.

- DES has 16 rounds; GOST has 32 rounds.

GOST's designers tried to achieve the balance between efficiency and security. They modified DES's basic design to create an algorithm which will work better for software implementation. Basically, the security of GOST has been increased by making the key very large, keeping the S-boxes secret, and doubling the number of iterations.

3.15 Blowfish

Blowfish is a 64-bit block cipher with a variable key length [86]. It was designed to meet the following criteria: speed, compactness, simplicity, and above all, security. It is optimized for applications where the key is mainly static, like communication lines, or automatic file encryption.

Blowfish is a Feistel network consisting of 16 rounds. The input is a 64-bit data element, x. The basic algorithm is given below. Full description of the algorithm can be found in [84].

Algorithm

x is divided into (x_L, x_R)

For $i = 1$ to 16:

$x_L = x_L \oplus P_i$

$x_R = F(x_L) \oplus x_R$

Swap x_l and x_R (undo the last swap)

$x_R = x_R \oplus P_{17}$

$x_L = x_L \oplus P_{18}$

Recombine x_L and x_R.

Decryption is the same as encryption except that $P_1 ... P_{18}$ are in reverse order.

3.16 Cryptography using Chaos

The use of deterministic chaos for encrypting data was based on the work of Blackledge and Ptitsyn [8] [9] [69]. It follows the same basic approach as that discussed earlier with regard to the application of modular based

pseudo random number generation. Pseudo chaotic numbers are in principle, ideal for cryptography because they produce number streams that are ultra-sensitive to the initial value (the key). However, instead of using iterative based maps using modular arithmetic with integer operations, here, we require the application of principally nonlinear maps using floating point arithmetic. Thus, the first drawback concerning the application of deterministic chaos for encryption concerns the processing speed, i.e. pseudo random number generators (PRNGs) generate integer streams using integer arithmetic where as pseudo chaotic number generators (PCNGs) produce floating point streams using floating point arithmetic. Another drawback of chaos based cryptography is that the cycle length (i.e. the period over which the number stream repeats itself) is relatively short when compared to the cycle length available using conventional PRNGs (e.g. additive generators). Thus, compared with conventional approaches, the application of deterministic chaos has (at least) two distinct disadvantages. However, providing the application of chaos in this field has some valuable advantages, the computational overheads can be enhanced through the use of appropriate real time DSP units (essentially, high performance floating point accelerators). Moreover, the lower cycle lengths can be overcome by designing block ciphers which is where an iterator produces a cipher stream only over a block of data whose length is significantly less than that of the cycle length of the iterator, each block being encrypted using a different key and/or algorithm. So are there any advantages to using deterministic chaos? One advantage is compounded in Figure 3.7 which qualitatively illustrates complexity as a function of information showing regions associated with ordered, random and chaotic fields. Imagine that an algorithm can output a number stream which can be ordered, chaotic or random. In the case of an ordered number stream (those

generated from a discretized piecewise continuous functions for example), the complexity of the field is clearly low. Moreover, the information and specifically the information Entropy (the lack of information we have about the exact state of the number stream) is low as is the information content that can be conveyed by such a number stream. A random number stream (taken to have a uniform distribution for example) will provide a sequence from which, under ideal circumstances, it is not possible to predict any number in the sequence from the previous values. All we can say is that the probability of any number occurring between a specified range is equally likely. In this case, the information entropy is high. However, the complexity of the field, in terms its erratic transitions from one type of localized behaviour to another, is low.

Thus, in comparison to a random field, a chaotic field is high in complexity but its information entropy, while naturally higher than an ordered field, is lower than that of a random field, e.g. chaotic fields which exhibit uniform number distributions are rare.

From the discussion above, the application of deterministic chaos to encryption has a number of disadvantages relative to the application of PRNGs. However, the increased level of complexity can be used to provide complexity driven block ciphers. One method of approach is to use well known maps and modify them to extend the region of chaos. For example, the Matthews

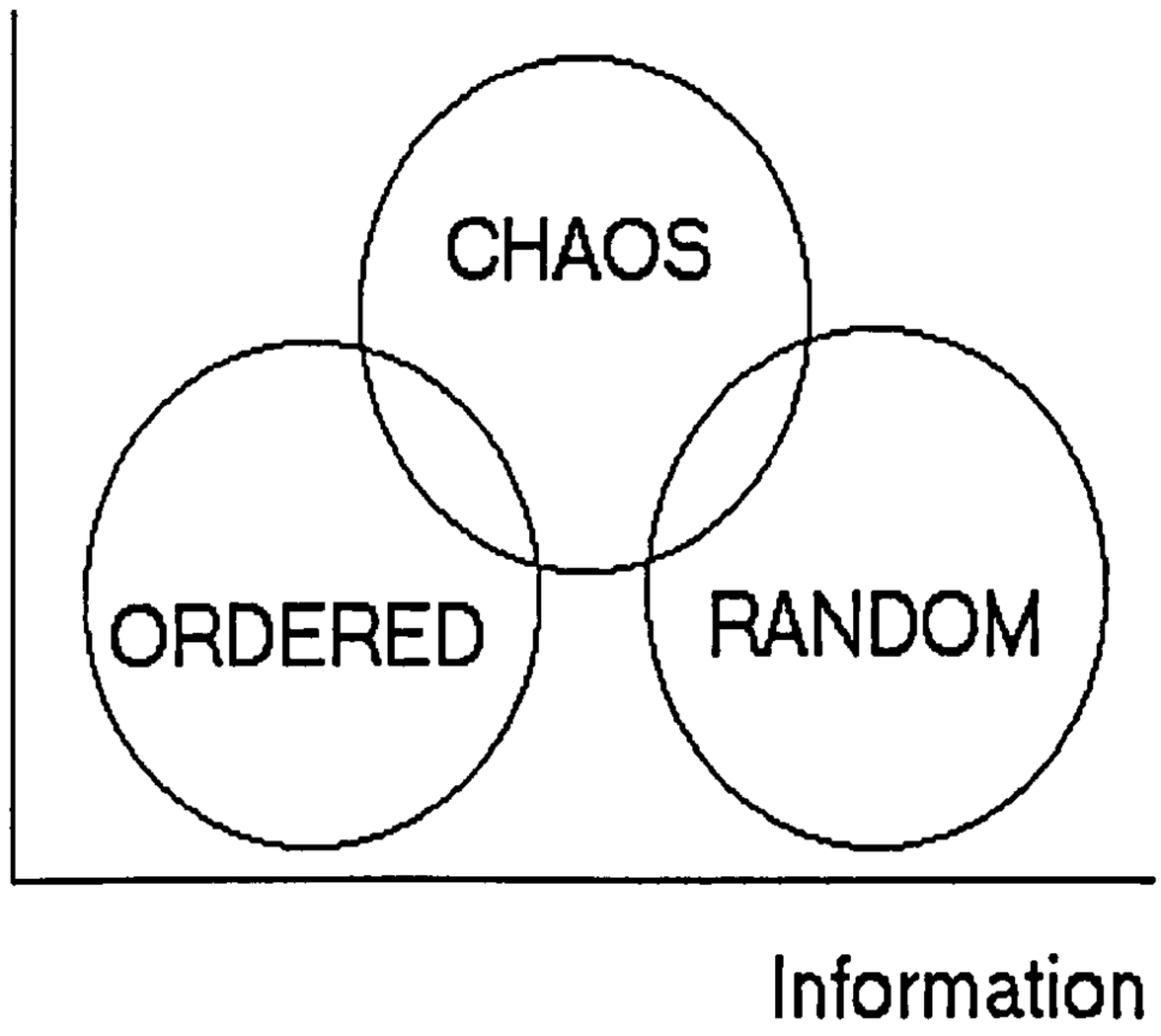


Figure 3.2: Qualitative comparison of ordered, random and chaotic fields in terms of their complexity and information content.

cipher is a modification of the logistic map to

$$x_{n+1} = (1+r)\left(1+\frac{1}{r}\right)^{r} x_n(1-x_n)^r, \quad r \in (0,4].$$

The effect of this generalization is seen in Figure 3.8 which shows the Feigenbaum diagram for values of r between 1 and 4. Compared to the conventional logistic map $x_{n+1} = rx_n(1-x_n), \quad r \in (0,4]$ which yields full chaos at $r = 4$, the chaotic behaviour of the Matthews map is clearly more extensive providing full chaos for the majority (but not all) of values of r between approximately 0.5 and 4. In the conventional case, the key is the value of x_0 (the initial condition). In addition, because there is a wide range chaotic behaviour for the Matthews map, the value of r itself can be used as a primary or secondary key.

The approach to using deterministic chaos for encryption has to date, been based on using conventional and other well known chaotic models of the type discussed above with modifications such as the Matthew map as required. However, in cryptography, the physical model from which a chaotic map has been derived is not important; only the fact that the map provides a cipher that is 'good' at scrambling the plaintext. This point leads to an approach which exploits two basic features of chaotic maps: (i) they increase the complexity of the cipher; (ii) there are an unlimited number of maps of the form $x_{n+1} = f(x_n)$ that can be literally 'invented' and then tested for chaoticity to produce a database of algorithms.

3.16.1 Block Ciphers using Deterministic Chaos

The low cycle lengths that are inherent in chaotic maps leads naturally to consider their application to block ciphers. However, instead of using a single algorithm to encrypted data over a series of blocks using different (block) keys, here we can use different algorithms, i.e. chaotic maps. Two maps can be used to generate the length of each block and the maps that are used to encrypt the plaintext over each block. Thus, suppose we have designed a database consisting of 100 chaotic maps say consisting of iterative functions $f_1, f_2, f_3, ..., f_{100}$, each of which generates a floating point number stream

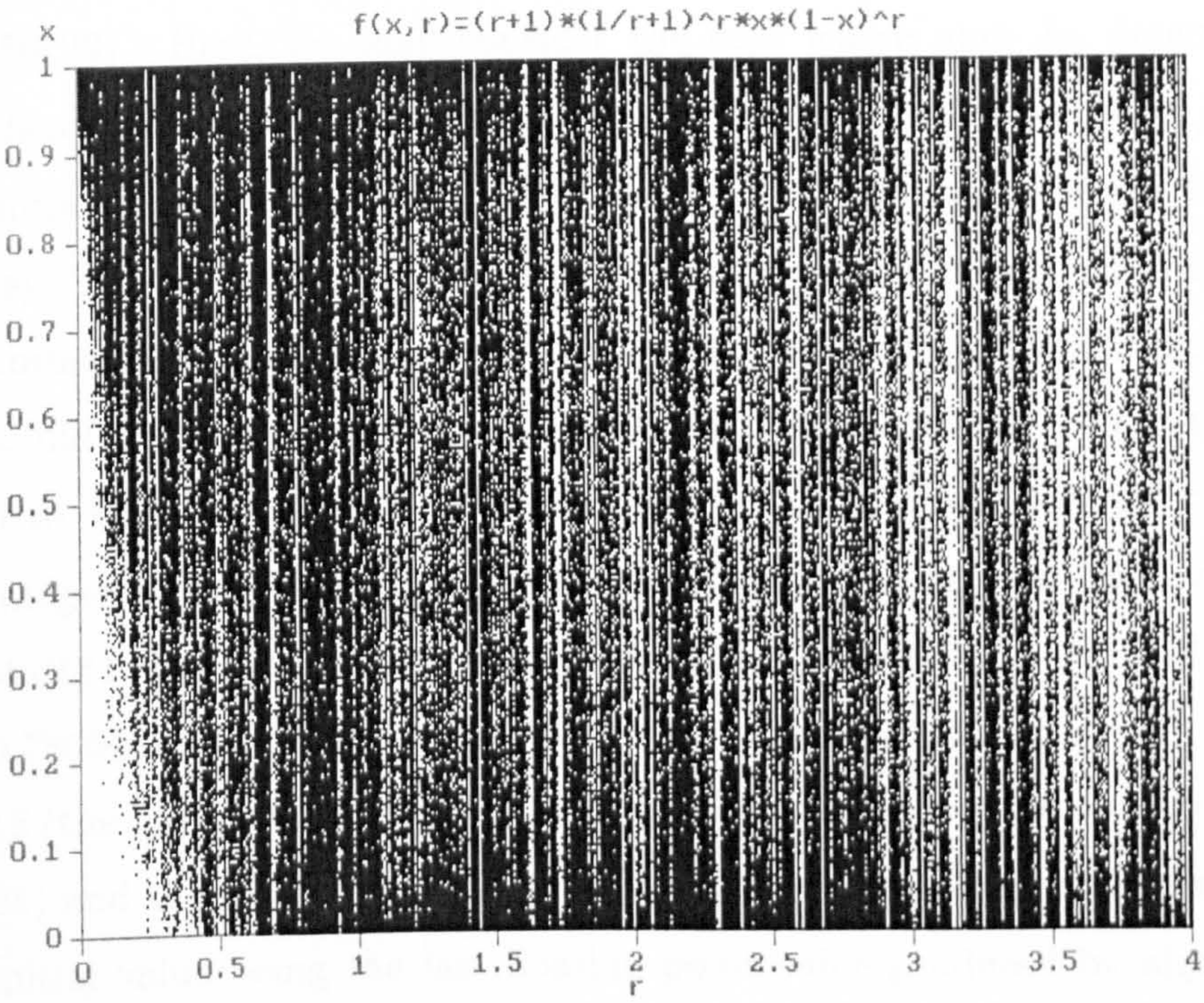


Figure 3.3: Feigenbaum map of the Matthews cipher

through the operation

$$x_{n+1} = f_m(x_n, p_1, p_2, ...)$$

where the parameters $p_1, p_2, ...$ are pre-set or 'hard-wired' to produce chaos for any initial value $x_0 \in (0,1)$ say. An 'algorithm selection key' is then introduced in which two algorithms (or the same algorithm) are chosen to 'drive' the block cipher - f_{50} and f_{29} say, the session key in this case being $(50,29)$. Here, we shall consider the case where map f_{50} determines the algorithm selection and map f_{29} determines the block size. Map f_{50} is then initiated with the key 0.26735625 say and map f_{29} with the key 0.65376301 say. The output from these maps (floating point number streams) are then normalized, multiplied by 100 and 1000 respectively for example and then rounded to produce integer streams with values ranging from 0 to 100 and 0 to 1000 respectively. Let us suppose that the first few values of these integer streams are $28,58,3,61$ and $202,38,785,426$. The block encryption starts by using map 28 to encrypt 202 elements of the plaintext using the key 0.78654876 say. The second block of 38 elements is then encrypted using map 58 (the initial value being the last floating point value produced by algorithm 28) and the third block of 785 elements is encrypted using algorithm 3 (the initial value being the last floating point value produced by algorithm 58) and so on. The process continues until the plaintext has been fully encrypted with the 'session key' (50,29,0.26735625,0.65376301,0.78654876).

3.16.2 Encrypting Processes

The encryption can be undertaken using a binary representation of the plaintext and applying an XOR operation using a binary representation of the cipher stream. This can be constructed using a variety of ways. For example,

one could extract the last significant bits from the floating point format of x_n for example. Another approach, is to divide the floating point range of the cipher into two compact regions and apply a suitable threshold. For example, suppose that the output x_n from a map operating over a given block consists of floating point value between 0 and 1, then, with the application of a threshold of 0.5, we can consider generating the bit stream

$$b(x_n) = \begin{cases} 1, & x_n \in (0.5, 1]; \\ 0, & x_n \in [0, 0.5). \end{cases}$$

However, in applying such a scheme, we are assuming that the distribution of x_n is uniform and this is rarely the case with chaotic maps. Figure 3.9 shows the PDF for the logistic map $x_{n+1} = 4x_n(1 - x_n)$ which reveals a non-uniform distribution with a bias for floating point number approach 0 and 1. However, the mid range (i.e. for $x_n \in [0.3, 0.7]$) is relatively flat indicating that the probability for the occurrence of different numbers generated by the logistic map in the mid range is the same. In order to apply the threshold partitioning method discussed above in a way that provides an output that is uniformly distributed for a any chaotic map, it is necessary to introduce appropriate conditions and modify the above to the form

$$b(x_n) = \begin{cases} 1, & x_n \in [T, T + \Delta_+); \\ 0, & x_n \in [T - \Delta_-, T); \\ -1, & \text{otherwise.} \end{cases}$$

where T is the threshold and Δ_+ and Δ_- are those values which characterize (to a good approximation) a uniform distribution. For example, in the case of the logistic map $T = 0.5$ and $\Delta_+ = \Delta_- = 0.2$. This aspect of the application of deterministic chaos to cryptography, together with the search

for a parameter or set of parameters that provides full chaos for an 'invented' map determines the overall suitability of the function that has been 'invented' for this application. The 'filtering' of a chaotic field to generate a uniformly distributed output is equivalent to maximizing the entropy of the cipher stream (i.e. generating a cipher stream with a uniform PDF) which is an essential condition in cryptography.

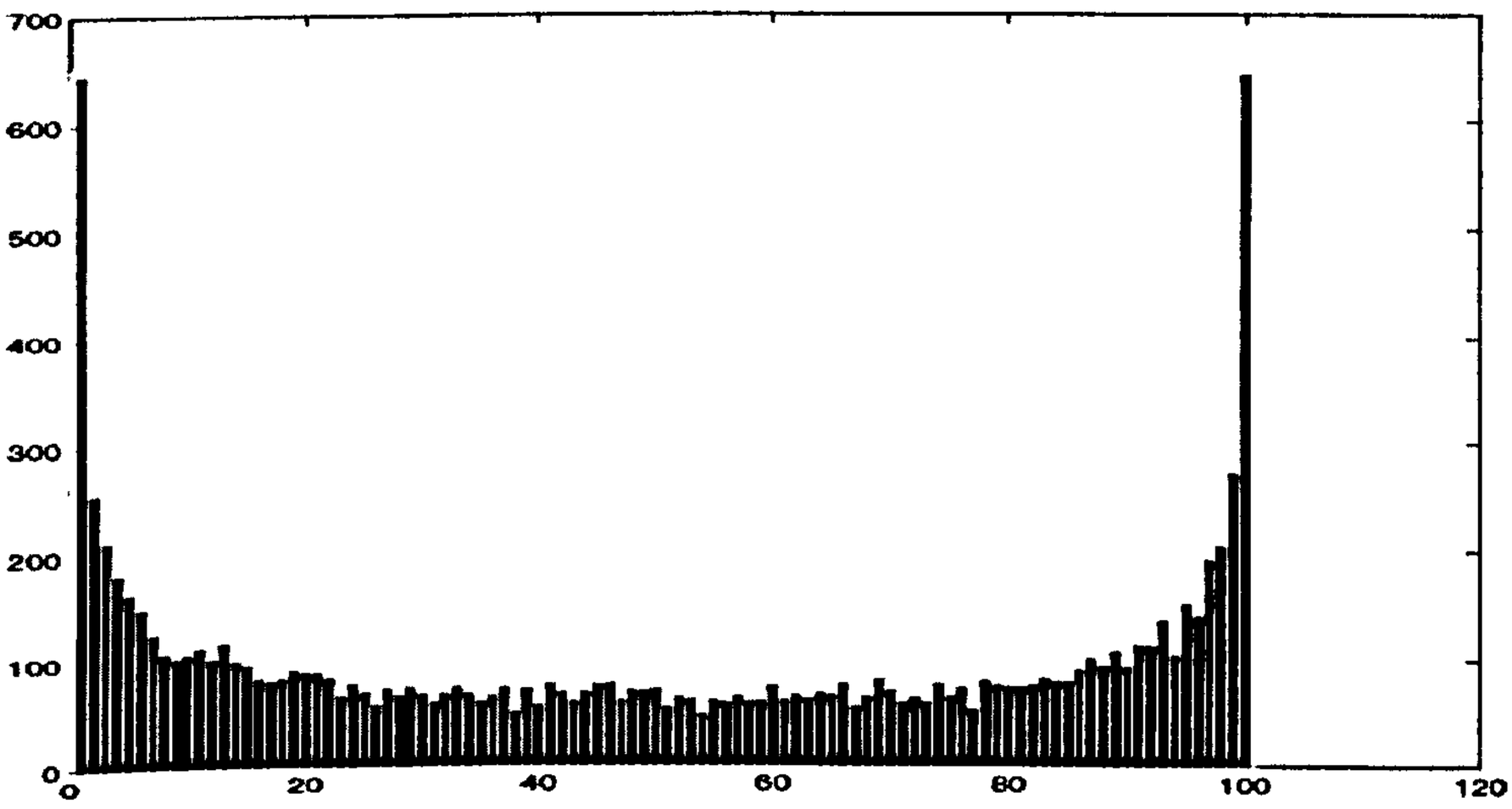


Figure 3.4: Probability density function (with 100 bins) of the output from the logistic map for 10000 iterations.

In terms of cryptanalysis and attack, the multi-algorithmic approach to designing a block cipher discussed here introduces a new 'dimension' to the attack problem. The conventional problem associated with an attack on a symmetric cipher is to search for the private key(s) given knowledge of the algorithm. Here, the problem is to search not only for the session key(s), but the algorithms they 'drive'. One over-riding issue concerning cryptol-

ogy in general, is that algorithm secrecy is weak. In other words, a cryptographic system should not rely of the secrecy of its algorithms and all such algorithms should be openly published.[3]. The system described here is multi-algorithmic, relying on many different chaotic maps to scramble the data. Here, publication of the algorithms can be done in the knowledge that many more maps can be invented as required (subject to appropriate conditions in terms of generating a fully chaotic field with a uniform PDF) by a programmer, or possibly with appropriate 'training' of a digital computer.

The idea of using chaotic encryption has been implemented using crypstic. (See Appendix B).

3.16.3 Key Exchange and Authentication

The process of 'scrambling' data using PCNGs or PRNGs is just one aspect of cryptography. The other major aspects are (i) key exchange; (ii) authentication. Without developing secure ways of transferring the keys from sender to receiver, there is little virtue in developing sophisticated methods of 'scrambling'. Further, the ability for a receiver to decrypt a transmission can lead to a false sense of confidence with regard to its content and authentication of a decrypted message is often necessary, particularly when a system is being attacked through the promotion of disinformation for example by searching for a crib, i.e. forcing an encrypted communication whose plaintext in known to have certain key words, phases or quotations for example.

With regard to chaotic block ciphers, one can apply the RSA algorithm dis-

[3]Except for some algorithms developed by certain federal government agencies. Perhaps they have something to hide!

cussed earlier, not to encrypt the plaintext, but to encrypt the sessions keys and the algorithm database. With regard to authentication of a message, one approach is to use a key that is plaintext dependent for which the chirp coding approach discussed previously can be used (with appropriate modifications). Further, application of chirp coding can be used to transfer a plaintext key in the cipher text, a key that is one of those used to encrypt/decrypt the data, but in contributing to the decryption, provides an authentication of the original plaintext. In effect, provided that appropriate protocols and procedures have been introduced, this approach, not only provides a method of authentication but does so, using a one time pad technique.

The history and development of encryption is a subject that has and continues to use a range of methods and approaches. However, there are some basic concepts that are easy to grasp and sometimes tend to get lost in the detail. The first of these is that the recipient of any encrypted message must have some form of *a priori* knowledge on the method (the algorithm for example) and the operational conditions (the public and/or private keys) used to encrypt a message. Otherwise, the recipient is in no better a 'state of preparation' than the potential attacker. The idea is to keep this *a priori* information to the bare minimum but in such a way that it is super critical to the decryption process. Another important reality is that in an attack, if the information transmitted is not deciphered in good time, then it is typically redundant. Coupled with the fact, that an attack usually has to focus on a particular approach (a specific algorithm for example), one way to enhance the security of a communications channel is to continually change the encryption algorithm and/or process offered by the technology currently available. This is the basis for a new commercial system called *Crypstic*, details of which are provided in Appendix B which includes the system development software.

3.17 Stream Ciphers

3.17.1 SEAL

SEAL is a software efficient stream cipher designed by Rogaway and Coppersmith [74]. SEAL is a pseudorandom function family (PRF): under a control of a key, first preprocessed into a set of tables. SEAL stretches a 32-bit 'position index' into a keystream of essential arbitrary length. It then encrypts by XORing this keystream with the plaintext, in the manner of a Vernam cipher. As with any Vernam cipher, it is imperative that the keystream only be used once. On a modern 32-bit processor, SEAL can encrypt messages at a rate of about 5 instructions per byte. In comparison, the DES algorithm is some 10-30 times as expensive.

SEAL is a length increasing PRF: under control of a 160-bit key a, SEAL maps a 32-bit string n to an L-bit string SEAL(a, n, L). The number L can be made as large or as small as is needed for a target application, but output lengths ranging from a few bytes to a few thousand bytes are anticipated.

A PRF can be used to make a good stream cipher. In a stream cipher the encryption of a message depends not only on the key a and the message x but also on the message's 'position' n in the data stream. This position is often a counter (sequence number) which indicates which message is being ciphered. The encryption of string x at position n is given by $(n, x \oplus SEAL(a, n, L))$, where $L = |x|$. In other applications n may indicate the address of a piece of data on disk.

SEAL has been designed with the following features which enhance its strength: [83]

1. Use of a large, secret, key-derived S-box.

2. Alternate arithmetic operations which do not commute (addition and XOR).

3. Use of an internal state maintained by the cipher which is not directly manifest in the data stream.

4. Varying the round function according to the round number, and varying the iteration function according to the iteration number.

One way to assess performance in a table-based cipher like SEAL is to simply count the number of S-box look-ups per byte generated output. SEAL uses 0.5 look-ups per byte of output. Merkle's 16-round Khufu uses 2 table look-ups per byte, while the S/P permutations of a software DES require 16 or 32 look-ups per byte. These comparisons ignore the rest of the work which each cipher does, and this work is in fact greater in SEAL than in Khufu or DES.

Even though SEAL provides a fast strong encryption, it does not, by itself, provide data authenticity. If there is a need, SEAL-encrypted message, can be accompanied by message authentication code (MAC).

3.17.2 RC4

RC4 is a variable key size stream cipher developed by Rivest in 1987 [58]. The keystream is independent of the plaintext. It has an 8-bit S-box: $S_0, S_1, ...S_{255}$. The entries are used as numbers 0 through 255 and permutated. The permutation itself is a function of the variable length key. The two counters i

and j are both initialised to zero.

The following function generates a random byte:

$i = (i + 1) \bmod 256$

$j = (j + S_i) \bmod 256$

swap S_i and S_j

$t = (S_i + S_j) \bmod 256$

$K = S_i$

Encryption takes place by XORing the byte K with the plaintext and decryption is the reverse. Encryption is 10 times faster than DES. RC4 is quite a strong encryption, even though its algorithm looks so simple that most experienced programmers can code it from memory.

3.17.3 FSAngo

FSAngo [82] is a Japanese high speed stream cipher which works on symmetric key systems. Developed in conjunction with Fujisoft ABC Inc. and Tokyo Denki University, the random key is generated using the FSRansu random number generator. Designers of FSRansu claim that the sequence provided by the PRNG do not provide enough information to identify the keys. The key space is over 10^{600}. The random numbers have been tested for frequency linear complexity and statistical distribution; the result could not determine any helpful pattern.

The program works at high speed and is compatible with all processors. It can also be easily implemented in hardware with minor modifications.

The table below compares different types of ciphers with the key strength and speed [5]

Cipher	Patented	Max Key Size	Block Size	Speed
RC6	Yes	2048 bits	128 bits	1.66 mb/s
Twofish	No	256 bits	128 bits	2.12 mb/s
Mars	Yes	1248 bits	128 bits	1.38 mb/s
Rijndael	No	256 bits	128 bits	2.12 mb/s
Blowfish	No	448 bits	64 bits	2.46 mb/s
Idea	Yes	128 bits	64 bits	0.75 mb/s
Gost	No	256 bits	64 bits	1.63 mb/s
Cast256	Yes	256 bits	64 bits	1.68 mb/s
Cast128	No	128 bits	64 bits	2.60 mb/s
Misty1	Yes	128 bits	64 bits	1.01 mb/s

Table 3.3: Comparison on different algorithms on key sizes and speeds.

Chapter 4

Digital Watermarking

4.1 Background to Watermarking

In this thesis, digital watermarking techniques are used to hide encryption keys within the ciphertext data that is transmitted. This overcomes the need to implement key exchange algorithms prior to the use of a symmetric encryption system. It also means that the key can be changed dynamically every time a ciphertext is transmitted and thus, provides a solution to implementing a one-time pad in practice, provided the algorithm is kept secure.

Watermarking is defined as the practice of hiding a message in an image, audio clip, video clip, or media within that work itself. The practice of using watermarks has existed for long time. Watermarking was first used by Italians around 1282. The marks were made by adding thin wire patterns to the paper moulds. Their use was picked up in the eighteenth century when the Europeans and Americans started using them as trademarks to indicate the date and the size of paper being manufactured. It was also about this

time that watermarks began to be used as anticounterfeiting measures on money and other documents. Many people started utilising watermarks for different applications. One of the problems banks were facing at the time was counterfeiting; it was easy to duplicate pound notes or dollar bills, as there was no sophistication in creating those notes. To overcome this problem, William Congreve invented a technique for making colour watermarks by inserting dyed material in the middle of the paper during the paper making process. The resulting marks must have been extremely difficult to forge, because the Bank of England itself declined to use them on the grounds that they were too difficult to make. In 1848, a more practical technology was invented by William Henry Smith [43], also from England. This replaced the fine wire patterns used to make earlier marks with a sort of shallow relief sculpture, pressed into the paper mould. The resulting variation on the surface of the mould produced beautiful watermarks with varying shades of gray. This is the basic technique used today for the face of Queen Elizabeth in the UK currency (5, 10, 20 and 50 notes).

Watermarks have been used on various occasions, depending on the situation. For example, in 1981, Margaret Thatcher arranged to distribute uniquely identifiable copies of sensitive documents to her ministers. Each copy had a different word spacing that was used to encode the identity of the recipient. When the confidential documents leaked to the press, it easily enabled the former British prime minister to identify the culprit. This example shows that one can be creative in using the technology in its simplest form.

Digital Watermarking techniques surfaced around 1995. Before that, the first who appear to have used the term *digital watermaking* were Komatsu and Tominaga. Watermarking, like cryptography, also uses secret keys to

map information to its owners, although the way this mapping is actually performed differs considerably from what is done in cryptography, mainly because the watermarked object should keep its intelligibility. In practice, a watermarked object may be altered either on purpose or accidentally, so the watermarking system should still be able to detect and extract the watermark. In cryptography, an object is protected for transmission and archiving, once decrypted protection is gone. Watermarking, on the other hand should protect the object beyond this. A number of attacks can be used against a watermarked object such as:

Filtering. Low-pass filtering does not introduce considerable degradation in watermarked objects, but can dramatically affect the performance, since spread-spectrum-like watermarks have a non negligible high-frequency spectral contents.

Cropping. In this attack the attacker is interested in only a small portion of the watermarked object, such as parts of a certain picture or text. Therefore, it is essential, when inserting a watermark, to spread the watermark over the dimensions of the data where this attack takes place.

Compression. This is an unintentional attack which appears very often in multimedia applications. Normally images that can be downloaded from the Internet have been compressed. If the watermark is required to resist different levels of compression, it is usually advisable to perform the watermark insertion task in the same domain where the compression takes place. For instance, DCT-domain image watermarking is more robust to JPEG compression than spatial-domain watermarking.

Multiple Watermarking. An attacker may take the object that has been watermarked, watermark it, and then claims his/her watermark was there

first. One way to prevent such attack is to timestamp the hidden information by a certification authority.

4.2 Applications of Watermarking

The popularity of watermarking is increasing rapidly since it gained recognition about a decade ago. Many are recognizing its usefulness and this has spun out a number of applications. We mention here a few obvious applications including some that have made impact to our own lives. In working with watermarking, one has to consider a balance between robustness and transparency; both are fundamentally opposed requirements, a tradeoff between the two must then be made.

Video Watermarking. Watermarks can be created either in spatial or in the DCT (Discrete Cosine Transform) domains[1]. If created in the DCT domain the results can be directly extrapolated to MPEG-2 sequences, although different actions must be taken for different frames.

Audio Watermarking. Here, the inability of a human ear to listen to certain frequencies are used to conceal the watermark and make it inaudible. In the context of standard audio processing and broadcast systems, the audio content may undergo various band-limiting stages. An audio watermark is expected to persist through such manipulations and therefore, it is imperative for the watermarking technology not to rely solely on portions of the audio spectrum that are perceptually less relevant. Of course, this should be done in a way that does not degrade the value of the content and the process of

[1]The reason for the DCT domain is that standard still and video compression schemes such as JPEG and MPEG are based on algorithms that utilise the properties of the DCT.

embedding a watermark should leave no perceivable audio artifacts.

Text Watermarking. Text documents can be watermarked by patterning the inter-word spaces. The words are classified using some features. Several adjacent words are grouped into a segment, and the segments are also classified using the word class information. The same amount of information is inserted into each of the segment classes. The information is encoded by modifying some statistics of inter-word spaces of the segments belonging to the same class.

Fingerprinting. This application allows devices such as a video cameras to insert information about itself, e.g. ID number and/or creation date. This can be done conventionally by using digital signature techniques or by using watermarking techniques. In the latter case, it is more secure, hence altering or removing the signature is a difficult task.

Broadcast Monitoring. In broadcasting, watermarks are usually inserted to programs that are widely broadcast. In this way, advertisers can be assured that they are getting the air time they paid for, and musicians can feel protected by not having their music being pirated and re-broadcasted.

4.3 The Matched Filter

The method of watermarking research for this thesis is based on application of a specific function - the chirp - coupled with a well defined processes - the matched filter. We now discuss the background to the technique, providing theoretical and algorithmic details on the approach taken.

The matched filter is a result of finding a solution to the following problem:

Given that

$$s_i = p_{i-j} f_j + n_i,$$

find an estimate for the Impulse Response Function (IRF) given by

$$\hat{f}_i = q_j s_{i-j}$$

where

$$r = \frac{|\sum_i Q_i P_i|^2}{\sum_i |N_i|^2 |Q_i|^2}$$

is a maximum. The ratio defining r is a measure of the signal-to-noise ratio. In this sense, the matched filter maximizes the signal-to-noise ratio of the output. Assuming that the noise n_i has a 'white' or uniform power spectrum, the filter Q_i which maximizes the SNR defined by r is given by

$$Q_i = P_i^*$$

and the required solution is therefore

$$\hat{f}_i = \text{IDFT}(P_i^* S_i).$$

Using the correlation theorem, we then have

$$\hat{f}_i = p_{j-i} s_j.$$

The matched filter is therefore based on correlating the signal s_i with the IRF p_i. This filter is frequently used in systems that employ linear frequency modulated (FM) pulses - 'chirped pulses' - which will be discussed later.

4.3.1 Derivation of the Matched Filter

With the problem specified as above, the matched filter is essentially a 'by-product' of the 'Schwarz inequality', i.e.

$$\left|\sum_i Q_i P_i\right|^2 \leq \sum_i |\, Q_i \,|^2 \sum_i |\, P_i \,|^2$$

The principal 'trick' is to write

$$Q_i P_i = |\, N_i \,|\, Q_i \times \frac{P_i}{|\, N_i \,|}$$

so that the above inequality becomes

$$\left|\sum_i Q_i P_i\right|^2 = \left|\sum_i |\, N_i \,|\, Q_i \frac{P_i}{|\, N_i \,|}\right|^2 \leq \sum_i |\, N_i \,|^2 |\, Q_i \,|^2 \sum_i \frac{|\, P_i \,|^2}{|\, N_i \,|^2}.$$

From this result, using the definition of r given above, we see that

$$r \leq \sum_i \frac{|\, P_i \,|^2}{|\, N_i \,|^2}.$$

Now, if r is to be a maximum, then we want

$$r = \sum_i \frac{|\, P_i \,|^2}{|\, N_i \,|^2}$$

or

$$\left|\sum_i |\, N_i \,|\, Q_i \frac{P_i}{|\, N_i \,|}\right|^2 = \sum_i |\, N_i \,|^2 |\, Q_i \,|^2 \sum_i \frac{|\, P_i \,|^2}{|\, N_i \,|^2}.$$

But this is only true if

$$|\, N_i \,|\, Q_i = \frac{P_i^*}{|\, N_i \,|}$$

and hence, r is a maximum when

$$Q_i = \frac{P_i^*}{|\, N_i \,|^2}.$$

4.3.2 White Noise Condition

If the noise n_i is white noise, then its power spectrum $\mid N_i \mid^2$ is uniformly distributed. In particular, under the condition

$$\mid N_i \mid^2 = 1 \quad \forall i = 0, 1, ..., N-1$$

then

$$Q_i = P_i^*.$$

4.3.3 FFT Algorithm for the Matched Filter

Using pseudo code, the algorithm for the matched filter is

```
for i=1, 2, ..., n; do:
   sr(i)=signal(i)
   si(i)=0.
   pr(i)=IRF(i)
   pi(i)=0.

       forward_fft(sr,si)
       forward_fft(pr,pi)

 for i=1, 2, ..., n; do:
   fr(i)=pr(i)*sr(i)+pi(i)*si(i)
   fi(i)=pr(i)*si(i)-pi(i)*sr(i)

      inverse_fft(fr,fi)
```

```
for i=1, 2, ..., n; do:
hatf(i)=fr(i)
```

4.3.4 Deconvolution of Frequency Modulated Signals

The matched filter is frequently used in systems that utilize linear frequency modulated (FM) pulses. IRF's of this type are known as chirped pulses. Examples of where this particular type of pulse is used include real and synthetic aperture radar, active sonar and some forms of seismic prospecting for example. Interestingly, some mammals (dolphins, whales and bats for example) use frequency modulation for communication and detection. The reason for this is the unique properties that FM IRFs provide in terms of the quality of extracting information from signals with very low signal-to-noise ratios and the simplicity of the process that is required to do this (i.e. correlation). The invention and use of FM IRFs for man made communications and imaging systems dates back to the early 1960s (the application of FM to radar for example); mother nature appears to have 'discovered' the idea some time ago.

Linear FM Pulses

The linear FM pulse is given (in complex form) by

$$p(t) = \exp(-i\alpha t^2), \quad \mid t \mid \leq T/2$$

where α is a constant and T is the length of the pulse. The phase of this pulse is αt^2 and the instantaneous frequency is given by

$$\frac{d}{dt}(\alpha t^2) = 2\alpha t$$

which varies linearly with t. Hence, the frequency modulations are linear which is why the pulse is referred to as a linear FM pulse. In this case, the signal that is recorded is given by (neglecting additive noise)

$$s(t) = \exp(-i\alpha t^2) \otimes f(t).$$

Matched filtering, we have

$$\hat{f}(t) = \exp(i\alpha t^2) \odot \exp(-i\alpha t^2) \otimes f(t).$$

Evaluating the correlation integral,

$$\exp(i\alpha t^2) \odot \exp(-i\alpha t^2) = \int_{-T/2}^{T/2} \exp[i\alpha(t+\tau)^2] \exp(-i\alpha\tau^2) d\tau$$

$$= \exp(i\alpha t^2) \int_{-T/2}^{T/2} \exp(2i\alpha\tau t) d\tau$$

and computing the integral over τ, we have

$$\exp(i\alpha t^2) \odot \exp(-i\alpha t^2) = T \exp(i\alpha t^2) \operatorname{sinc}(\alpha T t)$$

and hence

$$\hat{f}(t) = T \exp(i\alpha t^2) \operatorname{sinc}(\alpha T t) \otimes f(t).$$

In some systems, the length of the linear FM pulse is relatively long. In such cases,

$$\cos(\alpha t^2) \operatorname{sinc}(\alpha T t) \simeq \operatorname{sinc}(\alpha T t)$$

and

$$\sin(\alpha t^2) \operatorname{sinc}(\alpha T t) \simeq 0$$

and so

$$\hat{f}(t) \simeq T \operatorname{sinc}(\alpha T t) \otimes f(t).$$

Now, in Fourier space, this last equation can be written as

$$\hat{F}(\omega) = \begin{cases} \frac{\pi}{\alpha} F(\omega), & |\omega| \leq \alpha T; \\ 0, & \text{otherwise.} \end{cases}$$

The estimate $\hat{f}$ is therefore a band limited estimate of f whose bandwidth is determined by the product of the chirping parameter α with the length of the pulse T. An example of the matched filter in action is given in Figure 4.1 obtained using the MATLAB code given below. Here, two spikes have been convolved with a linear FM chirp whose width or pulse length T is significantly greater than that of the input signal. The output signal has been generated using an SNR of 1 and it is remarkable that such an excellent restoration of the input is recovered using a relatively simple operation for processing data that has been so badly distorted by additive noise. The remarkable ability for the matched filter to accurately recover information from linear FM type signals with very low SNRs leads naturally to consider its use for covert information embedding. This is the subject of the section that follows which investigates the use of chirp coding for covertly watermarking digital signals for the purpose of signal authentication.

```
function MATCH(T,snr)

%Input:
%       T - width of chirp IRF
%       snr - signa-to-noise ratio of signal
%
n=512;     %Set size of array (arbitrary)
nn=1+n/2; %Set mid point of array
```

```
%Compute input function (two spikes of width m centered
%at the mid point of the array.
m=10; %Set width of the spikes (arbitrary)
for i=1:n
   f(i)=0.0; %Initialize input
   p(i)=0.0; %Initialize IRF
end
   f(nn-m)=1.0;
   f(nn+m)=1.0;

%Plot result
figure(1);
subplot(2,2,1), plot(f);

%Compute the (real) IRF, i.e. the linear FM chirp using a
%sine function. (N.B. Could also use a cosine function.)
m=T/2;
k=1;
for i=1:m
   p(nn-m+i)=sin(2*pi*(k-1)*(k-1)/n);
   k=k+1;
end

%Plot result
subplot(2,2,2), plot(p);
```

```
%Convolve f with p using the convolution theorem and normalize to unity.
f=fft(f); p=fft(p);
          f=p.*f;
          f=ifft(f); f=fftshift(f); f=real(f);
f=f./max(f); %N.B. No check on case when f=0.

%Compute random Gaussian noise field and normalize to unity.
noise=randn(1,n);
noise=noise./max(noise);

%Compute signal with signal-to-noise ratio defined by snr.
s=f+noise./snr;

%Plot result
subplot(2,2,3), plot(s);

%Restore signal using Matched filter.

%Transform to Fourier space.
s=fft(s);

%Compute Matched filter.
rest=conj(p).*s;
rest=ifft(rest); rest=fftshift(rest); rest=real(rest);

%Plot result
subplot(2,2,4), plot(rest);
```

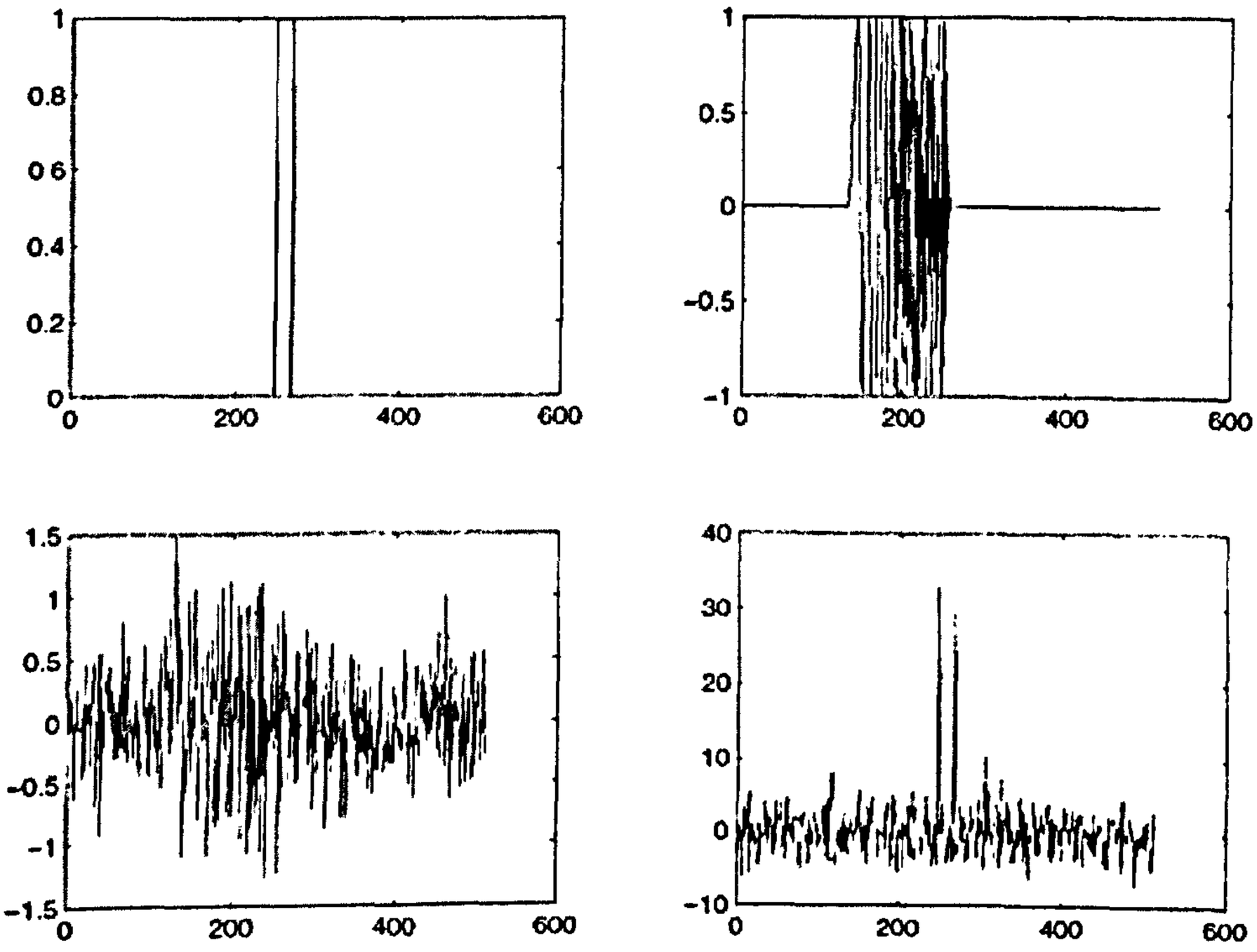


Figure 4.1: Example of a matched filter in action (bottom right) by recovering information from a noisy signal (bottom left) generated by the convolution of an input consisting of two spikes (top left) with a linear FM chirp IRF (top right). The simulation and restoration of the signal given in this example is accomplished using the MATLAB function MATCH(256,1).

4.4 Watermarking using Chirp Coding

In this section, we discusses a new approach to 'watermarking' digital signals using linear frequency modulated 'chirp coding'. The principle underlying this approach is based on the use of a matched filter to provide a reconstruction of a chirped code that is uniquely robust, i.e. in the case of very low signal-to-noise ratios.

Chirp coding for authenticating data is generic in the sense that it can be used for a range of data types and applications (the authentication of speech and audio signals for example). The theoretical and computational aspects of the matched filter and the properties of a chirp are briefly revisited to provide the essential background to the method. Signal code generating schemes are then addressed and details of the coding and decoding techniques considered.

4.4.1 Basic concepts

Methods of watermarking digital data have applications in a wide range of areas. Digital watermarking of images has been researched for many years in order to achieve methods which provide both anti-counterfeiting and authentication facilities. One of the principle equations that underpins this technology is based on the 'fundamental model' for a signal which is given by

$$s = \hat{P}f + n$$

where f is the information content for the signal (the watermark), $\hat{P}$ is some linear operator, n is the noise and s is the output signal. This equation is usually taken to describe a stationary process in which the noise n is characterized by stationary statistics (i.e. the probability density or distribution function of n is invariant of time). In the field of cryptology, the operation $\hat{P}f$ is referred to as the processes of 'diffusion' and the process of adding noise (i.e. $\hat{P}f + n$) is referred to as the process of 'confusion'. In cryptography and steganography (the process of hiding secret information in images) the principal 'art' is to develop methods in which the processes of diffusion and confusion are maximized, an important criterion being that the output s should be dominated by the noise n which in turn should be characterized by a maximum[2] (i.e. a uniform statistical distribution).

Digital watermarking and steganography can be considered to form part of the same field of study, namely, cryptology. Being able to recover f from s provides a way of authenticating the signal. If, in addition, it is possible to determine that a copy of s has been made leading to some form of data degradation and/or corruption that can be conveyed through an appropriate analysis of f, then a scheme can be developed that provides a check on: (i) the authenticity of the data s; (ii) its fidelity.

Formally, the recovery of f from s is based on the inverse process

$$f = \hat{P}^{-1}(s - n)$$

where $\hat{P}^{-1}$ is the inverse operator. Clearly, this requires the field n to be known *a priori*. If this field has been generated by a pseudo random number generator for example, then the seed used to generate this field must be known *a priori* in order to recover the data f. In this case, the seed represents

[2]A measure of the lack of information on the exact state of a system

the private key required to recover f. However, in principle, n can be any field that is considered appropriate for confusing the information $\hat{P}f$ including a pre-selected signal. Further, if the process of confusion is undertaken in which the signal-to-noise ratio is set to be very low (i.e. $\|n\| >> \|\hat{P}f\|$), then the watermark f can be hidden covertly in the data n provided the inverse process $\hat{P}^{-1}$ is well defined and computationally stable. In this case, it is clear that the host signal n must be known in order to recover the watermark f leading to a private watermarking scheme in which the field n represents a key. This field can of course be (lossless) compressed and encrypted as required. In addition, the operator $\hat{P}$ (and its inverse $\hat{P}^{-1}$) can be key dependent. The value of this operator key dependency relies on the nature and properties of the operator that is used and whether it is compounded in an algorithm that is required to be in the public domain for example.

Another approach is to consider the case in which the field n is unknown and to consider the problem of extracting the watermark f in the absence of this field. In this case, the reconstruction is based on the result

$$f = \hat{P}^{-1}s + m$$

where

$$m = -\hat{P}^{-1}n.$$

Now, if a process $\hat{P}$ is available in which $\|\hat{P}^{-1}s\| >> \|m\|$, then an approximate (noisy) reconstruction of f can be obtained in which the noise m is determined by the original signal-to-noise ratio of the data s and hence, the level of covertness of the diffused watermark $\hat{P}f$. In this case, it may be possible to post-process the reconstruction (de-noising for example) and recover a relatively high-fidelity version of the watermark, i.e.

$$f \sim \hat{P}^{-1}s.$$

This approach (if available) does not rely on a private key (assuming $\hat{P}$ is not key dependent). The ability to recover the watermark only requires knowledge of the operator $\hat{P}$ (and its inverse) and post-processing options as required. The problem here is to find an operator that is able to recover the watermark effectively in the presence of the field n. Ideally, we require an operator $\hat{P}$ with properties such that $\hat{P}^{-1}n \rightarrow 0$.

In this application, the operator is based on a chirp function, specifically, a linear Frequency Modulated (FM) chirp of the (complex) type $\exp(-i\alpha t^2)$ where α is the chirp parameter and t is the independent variable. This function is then convolved with f. The inverse process is undertaken by correlating with the (complex) conjugate of the chirp $\exp(i\alpha t^2)$. This provides a reconstruction for f in the presence of the field n that is accurate and robust with very low signal-to-noise ratios. Further, we consider a watermark based on a coding scheme in which the field n is the input. The watermark f is therefore n-dependent. This allows an authentication scheme to be developed in which the watermark is generated from the field in which it is to be hidden. Authentication of the watermarked data is then based on comparing the code generated from $s = \hat{P}f + n$ and that reconstructed by processing s when $\|\hat{P}f\| >> \|n\|$. This is an example of a self-generated coding scheme which avoids the use, distribution and application of reference codes. Here, the coding scheme is based on the application of Daubechies wavelets. There are numerous applications of this technique in areas such as telecommunications and speech recognition where authentication is mandatory. For example, the method can readily be applied to audio data with no detectable differences in the audio quality of the data. The watermark code is able to be recovered accurately and changes relatively significantly if the data is distorted through cropping, filtering, noise or a compression system

for example. Thus, it provides a way making a signal tamper proof.

4.4.2 Matched Filter Reconstruction

Given that

$$s(t) = \exp(-i\alpha t^2) \otimes f(t) + n(t),$$

after matched filtering, we obtain the estimate

$$\hat{f}(t) \simeq T\,\mathrm{sinc}(\alpha T t) \otimes f(t) + \exp(i\alpha t^2) \odot n(t).$$

The correlation function produced by the correlation of $\exp(i\alpha t)$ with $n(t)$ will in general be relatively low in amplitude since $n(t)$ will not normally have features that match those of a chirp. Thus, it is reasonable to assume that

$$\|T\,\mathrm{sinc}(\alpha T t) \otimes f(t)\| >> \|\exp(i\alpha t^2) \odot n(t)\|$$

and that in practice, $\hat{f}$ is a band-limited reconstruction of f with high SNR. Thus, the process of using chirp signals with matched filtering for the purpose of reconstructing the input in the presence of additive noise provides a relatively simple and computationally reliable method of 'diffusing' and reconstructing information encoded in the input function f. This is the underlying principle behind the method of watermarking described here.

4.4.3 The Fresnel Transform

Ignoring scaling, we can define the Fresnel transform as

$$s(x, y) = \exp[-i\alpha(x^2 + y^2)] \otimes \otimes f(x, y).$$

This result is just a 2D version of the 'chirp transform' discussed earlier. The reconstruction of f from s follows the same principles and can be accomplished using a correlation of s with the function $\exp[i\alpha(x^2+y^2)]$. This result leads directly to a method of digital image watermarking using the Fresnel transform to 'diffuse' the watermark f. In particular, reverting to the operator notation used previously, our Fresnel transform based watermarking model becomes

$$s(x,y) = \hat{P}f(x,y) + n(x,y)$$

where the operator $\hat{P}$ is given by

$$\hat{P} = \exp[-i\alpha(x^2+y^2)] \otimes\otimes$$

and the inverse operator is given by

$$\hat{P}^{-1} = \exp[i\alpha(x^2+y^2)] \odot\odot \ .$$

Note, that $\otimes\otimes$ denotes 2D convolution and $\odot\odot$ denotes 2D correlation. Also, in practice, only values ≥ 0 can be used for application to digital images so that we must consider a function of the normalized form $(1+\exp[i\alpha(x^2+y^2)])/2$ for example.

A covert watermarking procedure involves the addition of a (diffused) watermark to a host image with a very low watermark-to-signal ratio, i.e.

$$\|\hat{P}f(x,y)\| << \|n(x,y)\|.$$

Recovery of the watermark is then based on the result

$$f(x,y) = \hat{P}^{-1}[s(x,y) - n(x,y)].$$

4.4.4 Chirp Coding, Decoding and Watermarking

We now return to the issue of watermarking using chirp functions. The basic model for the watermarked signal (which is real) is

$$s(t) = \text{chirp}(t) \otimes f(t) + n(t)$$

where

$$\text{chirp}(t) = \sin(\alpha t^2).$$

We consider the field $n(t)$ to be some pre-defined signal to which a watermark is to be 'added' to generate $s(t)$. In principle, any watermark described by the function $f(t)$ can be used. On the other hand, for the purpose of authentication we require two criterion: (i) $f(t)$ should represent a code which can be reconstructed accurately and robustly; (ii) the watermark code should be sensitive (and ideally ultra-sensitive) to any degradation in the field $n(t)$ due to lossy compression, cropping or highpass and lowpass filtering for example. To satisfy condition (i), it is reasonable to consider $f(t)$ to represent a bit stream, i.e. to consider the discretized version of $f(t)$ - the vector f_i - to be composed of a set of elements with values 0 or 1 and only 0 or 1. This binary code can of course be based on a key or set of keys which, when reconstructed, is compared to the key(s) for the purpose of authenticating the data. However, this requires the distribution of such keys (public and/or private). Instead, we consider the case where a binary sequence is generated from the field $n(t)$. There are a number of approaches that can be considered based on the spectral characteristics of $n(t)$ for example. These are discussed later on, in which binary sequences are produced from the application of wavelet decomposition.

Chirp Coding

Given that a binary sequence has been generated from $n(t)$, we now consider the method of chirp coding. The purpose of chirp coding is to 'diffuse' each bit over a range of compact support T. However, it is necessary to differentiate between 0 and 1 in the sequences. The simplest way to achieve this is to change the polarity of the chirp. Thus, for 1 we apply the chirp $\sin(\alpha t^2)$, $t \in T$ and for 0 we apply the chirp -$\sin(\alpha t^2)$, $t \in T$ where T is the chirp length. The chirps are then concatenated to produce a contiguous stream of data, i.e. a signal composed of ±chirps. Thus, the binary sequence 010 for example is transformed to the signal

$$s(t) = \begin{cases} -\text{chirp}(t), & t \in [0, T); \\ +\text{chirp}(t), & t \in [T, 2T); \\ -\text{chirp}(t), & t \in [2T, 3T). \end{cases}$$

The period over which the chirp is applied depends on the length of the signal to which the watermark is to be applied and the length of the binary sequence. In the example given above, the length of the signal is taken to be $3T$. In practice, care must be taken over the chirping parameter α that is applied for a period T in order to avoid aliasing and in some cases it is of value to apply a logarithmic sweep instead of a linear sweep. The instantaneous frequency of a logarithmic chirp is given by

$$\psi(t) = \psi_0 + 10^{at}$$

where

$$a = \frac{1}{T}\log_{10}(\psi_1 - psi_0)$$

ψ_0 is the initial frequency and ψ_1 is the final frequency at time T. In this case, the final frequency should be greater than the initial frequency.

Decoding

Decoding or reconstruction of the binary sequence requires the application of a correlator using the function chirp(t), $t \in [0, T)$. This produces a correlation function that is either -1 or +1 depending upon whether $-$chirp(t) or +chirp(t) has been applied respectively. For example, after correlating the chirp coded sequence 010 given above, the correlation function $c(t)$becomes

$$c(t) = \begin{cases} -1, & t \in [0, T); \\ +1, & t \in [T, 2T); \\ -1, & t \in [2T, 3T). \end{cases}$$

from which the original sequence 010 is easily inferred, the change in sign of the correlation function identifying a bit change (from 0 to 1 or from 1 to 0). Note, that in practice the correlation function may not be exactly 1 or -1 when reconstruction is undertaken and the binary sequence is effectively recovered by searching the correlation function for changes in sign. The chirp used to recover the watermark must of course have the same parameters (inclusive of its length) as those used to generate the chirp coded sequence. These parameters can be used to define part of a private key.

Watermarking

The watermarking process is based on adding the chirp coded data to the signal $n(t)$. Let the chirp coded signal be given by the function $h(t)$, then the watermarking process is described by the equation

$$s(t) = a\left[\frac{bh(t)}{\|h(t)\|_\infty} + \frac{n(t)}{\|n(t)\|_\infty}\right]$$

and the coefficients $a > 0$ and $0 < b < 1$ determine the amplitude and the

SNR of s where

$$a = \|n(t)\|_\infty.$$

The coefficient a is required to provide a watermarked signal whose amplitude is compatible with the original signal n. The value of b is adjusted to provide an output that is acceptable in the application to be considered and to provide a robust reconstruction of the binary sequence by correlating $s(t)$ with chirp$(t), t \in [0, T)$. To improve the robustness of the reconstruction, the value of b can be increased, but this has to be off-set with regard to the perceptual quality of the output, i.e. the perturbation of n by h should be as small as possible.

4.4.5 Code Generation

In the previous section, the method of chirp coding a binary sequence and watermarking the signal $n(t)$ has been discussed where it is assumed that the sequence is generated from this same signal. In this section, the details of this method are presented. The problem is to convert the salient characteristics of the signal $n(t)$ into a sequence of bits that is relatively short and conveys information on the signal that is unique to its overall properties. In principle, there are a number of ways of undertaking this. For example, in practice the digital signal n_i, which will normally be composed of an array of floating point numbers, could be expressed in binary form and each element concatenated to form a contiguous bit stream. However, the length of the code (i.e. the total number of bits in the stream) will tend to be large leading to high computational costs in terms of the application of chirp coding/decoding. What is required, is a process that yields a relatively short binary sequence (when compared with the original signal) that reflects the important prop-

erties of the signal in its entirety. Two approaches are considered here: (i) power spectral density decomposition and (ii) wavelet decomposition.

Power Spectral Density Decomposition

Let $N(\omega)$ be the Fourier transform $n(t)$ and define the Power Spectrum $P(\omega)$ as

$$P(\omega) = \mid N(\omega) \mid^2 .$$

An important property of the binary sequence is that it should describe the spectral characteristics of the signal in its entirety. Thus, if for example, the binary sequence is based on just the low frequency components of the signal, then any distortion of the high frequencies components will not affect the watermark and the signal will be authenticated. Hence, we consider the case where the power spectrum is decomposed into N components, i.e.

$$P_1(\omega) = P(\omega), \quad \omega \in [0, \Omega_1);$$

$$P_2(\omega) = P(\omega), \quad \omega \in [\Omega_1, \Omega_2);$$

$$\vdots$$

$$P_N(\omega) = P(\omega), \quad \omega \in [\Omega_{N-1}, \Omega_N).$$

Note, that it is assumed that the signal $n(t)$ is band-limited with a bandwidth of Ω_N.

The set of the functions $P_1, P_2, ..., P_N$ now reflect the complete spectral characteristics of the signal $n(t)$. Since each of these functions represents a unique part of the spectrum, we can consider a single measure as an identifier or tag. A natural measure to consider is the energy which is given by the integral of the functions over their frequency range. In particular, we consider the

energy values in terms of their contribution to the spectrum as a percentage, i.e.

$$E_1 = \frac{100}{E} \int_0^{\Omega_1} P_1(\omega) d\omega,$$

$$E_2 = \frac{100}{E} \int_{\Omega_1}^{\Omega_2} P_2(\omega) d\omega,$$

$$\vdots$$

$$E_N = \frac{100}{E} \int_{\Omega_{N-1}}^{\Omega_N} P_N(\omega) d\omega,$$

where

$$E = \frac{100}{E} \int_0^{\Omega_N} P(\omega) d\omega.$$

Code generation is then based on the following steps:

(i) Rounding to the nearest integer the (floating point) values of E_i to decimal integer form:

$$e_i = \text{round}(E_i), \quad \forall i.$$

(ii) Decimal integer to binary string conversion: conversion

$$b_i = \text{binary}(e_i).$$

(iii) Concatenation of the binary string array b_i to a binary sequence:

$$f_j = \text{cat}(b_i).$$

The watermark f_j is then chirp coded as discussed previously.

Wavelet Decomposition

The wavelet transform is defined by

$$\hat{W}[f(t)] = F_L(t) = \int f(\tau) w_L(t,\tau) d\tau$$

where

$$w_L(t,\tau) = \frac{1}{\sqrt{\mid L \mid}} w\left(\frac{t-\tau}{L}\right).$$

The wavelet transformation is essentially a convolution transform in which $w(t)$ is the convolution kernel but with a factor L introduced. The introduction of this factor provides dilation and translation properties into the convolution integral (which is now a function of L) that gives it the ability to analyse signals in a multi-resolution role.

The code generating method is based on computing the energies of the wavelet transformation over N levels. Thus, the signal $f(t)$ is decomposed into wavelet space to yield the following set of functions:

$$F_{L_1}(\tau), \quad F_{L_2}(\tau), \quad \ldots \quad F_{L_N}(\tau).$$

The (percentage) energies of these functions are then computed, i.e.

$$E_1 = \frac{100}{E} \int \mid F_{L_1}(\tau) \mid^2 d\tau,$$

$$E_2 = \frac{100}{E} \int \mid F_{L_1}(\tau) \mid^2 d\tau,$$

$$\vdots$$

$$E_N = \frac{100}{E} \int \mid F_{L_N}(\tau) \mid^2 d\tau,$$

where

$$E = \sum_{i=1}^{N} E_i.$$

The method of computing the binary sequence for chirp coding from these energy values follows that described in the method of power spectral decomposition. Clearly, whether applying the power spectral decomposition

method or wavelet decomposition, the computations are undertaken in digital form using a DFT and a DWT (Discrete Wavelet Transform) respectively.

4.4.6 MATLAB Application Programs

Two prototype MATLAB programs have been developed to implement the watermarking method discussed. The *coding process* reads in a named file, applies the watermark to the data using wavelet decomposition and writes out a new file using the same file format. The *Decoding process* reads a named file (assumed to contain the watermark or otherwise), recovers the code from the watermarked data and then recovers the (same or otherwise) code from the watermark. The coding program displays the decimal integer and binary codes for analysis. The decoding program displays the decimal integer streams generated by the wavelet analysis of the input signal and the stream obtained by processing the signal to extract the watermark code or otherwise. This process also provides an error measure based on the result

$$e = \frac{\sum_i | x_i - y_i |}{\sum_i | x_i + y_i |}$$

where x_i and y_i are the decimal integer arrays obtained from the input signal and the watermark (or otherwise). In the application considered here, the watermarking method has been applied to audio (.wav) files in order to test the method on data which requires that the watermark does not affect the fidelity of the output (i.e. audio quality). Only a specified segment of the data is extracted for watermarking which is equivalent to applying and offset to the data. The segment can be user defined and if required, form the basis for a (private) key system. In this application, the watermarked segment has been 'hard-wired' and represents a public key. The wavelets

used are Daubechies wavelets computed using the MATLAB wavelet toolbox. However, in principle, any wavelets can be used for this process and the actual wavelet used yields another feature that can form part of the private key required to extract the watermark.

Coding Process

The coding process is compounded in the following basic steps:

Step 1: Read a .wav file.

Step 2: Extract a section of a single vector of the data (note that a .wav contains stereo data, i.e. two vectors).

Step 3: Apply wavelet decomposition using Daubechies wavelets with 7 levels. Note, that in addition to wavelet decomposition, the approximation coefficients for the input signal are computed to provide a measure on the global effect of introducing the watermark into the signal. Thus, 8 decomposition vectors in total are generated.

Step 4: Compute the (percentage) 'energy values'.

Step 5: Round to the nearest integer and convert to binary form.

Step 6: Concatenate both the decimal and binary integer arrays.

Step 7: Chirp code the binary sequence.

Step 8: Scale the output and add to the original input signal.

Step 9: Re-scale the watermarked signal.

Step 10: Write to a file.

The above procedure has been implemented where the parameters for segmenting and processing data of a specific size have been 'hard wired'. The Matlab code (encode.m) has been included in the accompanying CD.

Decoding process

The decoding process is as follows:

Step 1: Steps 1-6 in the coding processes are repeated.

Step 2: Correlate the data with a chirp identical to that used for chirp coding.

Step 3: Extract the binary sequence.

Step 4: Convert from binary to decimal.

Step 5: Display the original and reconstructed decimal sequence.

Step 6: Display the error.

A complete Matlab code showing decoding process (dccode.m) has beeu included in the accompanying CD.

4.4.7 Discussion

In a practical application of this method for authenticating audio files for example, a threshold can be applied to the error value. If and only if the error lies below this threshold is the data taken to be authentic.

The prototype MATLAB programs provided have been developed to explore the applications of the method for different signals and systems of interest to the user. Note that in the decoding program, the correlation process is carried out using a spatial cross-correlation scheme (using the MATLAB function *xcorr*), i.e. the watermark is recovered using the process chirp(t) $\odot$ $s(t)$ instead of the Fourier equivalent CHIRP$^*(\omega)S(\omega)$ where CHIRP and S are the Fourier transforms of chirp and s respectively (in digital form of course). This is due to the fact that the 'length' of the chirp function is significantly less than that of the signal. Application of a spatial correlator therefore provides greater computational efficiency.

The method of digital watermarking discussed here makes specific use of the chirp function. This function is unique in terms of its properties for reconstructing information (via application of the Matched Filter) that has been 'diffused' through the convolution process,i.e. the watermark extracted is, in theory, an exact band-limited version of the original watermark as defined in the presence of significant additive noise, in this case, the signal into which the watermark is 'embedded'. The method has a close relationship with the Fresnel transform and can be used for digital image watermarking in an entirely equivalent way. The approach considered here allows a code to be generated directly from the input signal and that same code used to watermark the signal. The code used to watermark the signal is therefore self-generating. Reconstruction of the code only requires a correlation process

with the watermarked signal to be undertaken. This means that the signal can be authenticated without access to an external reference code. In other words, the method can be seen as a way of authenticating data by extracting a code (the watermark) within a code (the signal).

Audio data watermarking schemes rely on the imperfections of the human audio system. They exploit the fact that the human auditory system is insensitive to small amplitude changes, either in the time or frequency domains, as well as insertion of low amplitude time domain echo's. Spread spectrum techniques augment a low amplitude spreading sequence which can be detected via correlation techniques. Usually, embedding is performed in high amplitude portions of the signal, either in the time or frequency domains. A common pitfall for both types of watermarking systems is their intolerance to detector de-synchronization and deficiency of adequate methods to address this problem during the decoding process. Although other applications are possible, chirp coding provides a new and novel technique for fragile audio watermarking. In this case, the watermarked signal does not change the perceptual quality of the signal. In order to make the watermark inaudible, the chirp generated is of very low frequency and amplitude. Using audio files with sampling frequencies of over 1000Hz, a logarithmic chirp can be generated in the frequency band of 1-100Hz. Since the human ear has low sensitivity in this band, the embedded watermark will not be perceptible. Depending upon the band and amplitude of the chirp, the signal-to-watermark ratio can be in excess of 40dB. Various forms of attacks can be applied which change the distribution of the percentage sub-band energies originally present in the signal including filtering (both low pass and high pass), cropping and lossy compression (MP3 compression) with both constant and variable bit rates. In each case, the signal and/or the watermark is distorted enough to reg-

ister the fact that the data has been tampered with. Further, chirp based watermarks are difficult to remove from the signal since the initial and the final frequency is at the discretion of the user and its position in the data stream can be varied through application of an offset, all such parameters being combined to form a private key.

4.5 Echelon

When working with encryption, one has to realise that once the data is encrypted, ultimate protection has been practiced. The rest is to assess the strength and techniques used. The strength may come from the encryption itself, it could have strong algorithm and therefore cannot easily be broken, for example. On the other hand, the adversary may not be an average hacker. In order to have encryption 'muscle', powerful computers are needed. Moreover, the best of cryptographers are needed to be disposed. This can only happen in government organisations.

Some of the best cryptographers in the world can be found working for the government. Most of the military secrets have to be closely guarded since this information has to be shared and sent to the military allies across the globe. Also military communication secrets are well guarded. There is no room for compromise when it comes to transmitting or gathering military intelligence.

Even though they may not openly admit to it, most government organisations work hard at eavesdropping all communications around their borders and beyond. In the USA, the National Security Agency (NSA) performs highly specialized activities to protect US information systems and produce

foreign intelligence information. The NSA is also responsible for creating the encryption algorithms for messages used for secret communications. These algorithms are kept secret and are never published. The agency has created under its wings a global spy system codenamed ECHELON, which captures and analyzes virtually every phone call, fax, email and telex message sent anywhere in the world. ECHELON is controlled by the NSA and is operated in conjunction with the Government Communications Head Quarters (GCHQ) of England, the Communications Security Establishment (CSE) of Canada, the Australian Defense Security Directorate (DSD), and the General Communications Security Bureau (GCSB) of New Zealand. These organisations are bound together under a secret 1948 agreement, UK-USA, whose terms and text remain under wraps even today. The NSA was established in 1952 by president Harry Truman. After it was setup, the facilities were kept secret and the government did not openly admit to its existence until 1957. The ECHELON centre is Headquartered at Fort George Meade, located between Washington D.C. and Baltimore, Maryland. To date, NSA is the largest employer of mathematicians and cryptographers.

The ECHELON system deploys the largest spy station in the world, with over twenty-five satellite receiving stations and 1,400 American NSA personnel working with 350 UK Ministry of Defense staff on site. The power of ECHELON resides in its ability to decrypt, filter, examine and codify these messages into selective categories for further analysis by intelligence agents from the various UKUSA agencies. Once the data gets sifted, it is given to the cryptographers for breaking and cracking the codes. The work involves intercepting and decoding messages in over 100 languages.

The ECHELON system does not only cover text and images. It also inter-

cepts voice communication through satellite links. In the UK, as the electronic signals are brought into the station, they are fed through the massive computer systems, such as Menwith Hills SILKWORTH, where voice recognition, optical character recognition (OCR) and data information engines get to work on the messages. By using powerful voice recognition systems, voice patterns of *interest* are stored and picked up whenever they appear in the conversation. This allows the tracking of certain known individuals. Each station maintains a list of key words (The Dictionary). The managers at each station are free to add and delete the keywords according to their needs.

Politicians are known to have manipulated the system to their advantage. Margaret Thatcher used Echelon to spy on her two cabinet members she suspected were disloyal to her. In order to avoid any legal implications, the request was undertaken by the Canadian CSE.

ECHELON has been used beyond political motivation; governments have used the system for commercial interests. In 1990, the German magazine, Der Speigel, revealed that the NSA had intercepted messages about an impending $200 million deal between Indonesia and the Japanese satellite manufacturer NEC Corp. After President Bush intervened in the negotiations on behalf of American manufacturers, the contract was split between NEC and AT&T.

The ECHELON system has placed its *listening devices* all across the globe. This enables them to intercept all the communications between all satellite systems. There has been great concern from the watch groups that ECHELON does not serve the purpose it was originally intended for. According the echelon watch website [30]:

'Echelon is perhaps the most powerful intelligence gathering organization in the world.

Several credible reports suggest that this global electronic communications surveillance system presents an extreme threat to the privacy of people all over the world. According to these reports, ECHELON attempts to capture staggering volumes of satellite, microwave, cellular and fiber-optic traffic, including communications to and from North America. This vast quantity of voice and data communications are then processed through sophisticated filtering technologies.

This massive surveillance system apparently operates with little oversight. Moreover, the agencies that purportedly run ECHELON have provided few details as to the legal guidelines for the project. Because of this, there is no way of knowing if ECHELON is being used illegally to spy on private citizens.

This site is designed to encourage public discussion of this potential threat to civil liberties, and to urge the governments of the world to protect our rights.

ECHELON has huge listening facilities and its network is directed at Intelsat and Inmarsat satellites. These two satellites are responsible for the vast majority of phone and fax communications traffic within and between countries and continents.

Most of the work done in these projects is a closely guarded secret. Therefore it is hard to determine what methods these governments employ when it comes to intercepting and trying to crack the supposedly malicious message. One of the methods used on encrypted messages is to scan all packets going through the channel. Once an encrypted message is encountered, it is then sifted for further scrutiny. Normally when a file is processed, be it a text file or a binary file, the processing object will leave a mark in the file. This mark should enable the processing object, or program, to recognize the file if it is encountered again. For example, if we open a new Microsoft Word

™document, the output when viewed in binary format will show that the document indeed was created in MS Word. There will be a file signature which will list all properties of the document.

Most encryption software creates a signature on the data that is output. Of course for stronger encryption, the signature may not be obvious. But to hackers or experienced cryptographers, these can be easily rooted out. Some may visually appear on the binary files, while others may appear in forms of patterns which are unique to a certain algorithm. The NSA is one of the most experienced organisations when it comes to cryptography and cryptanalysis. It has been around for a long time, and this has enabled it to gather as much data as possible on ALL types on cryptographic data that pass through the Internet. Also, that fact that NSA works with other organisations across the globe allows it to have access to different types of encrypted data and in different languages.

The filtering system first goes through each packet to determine the type of data. If it is plaintext for example, it will be classified and go through its assigned process. The same goes for still images, video and audio clips. Once the *filtering* system has determined that the data is encrypted, the first thing it will look for is a signature. The signature will determine the type of encrypted data. As mentioned earlier, the signature may not be obvious and therefore the difficulty in deciphering the signature depends on part the complexity of the encryption algorithm. Once it has been ascertained, the output will be moved to the next *bin* for further analysis.

Different methods of cryptanalysis will be used to crack the code. Once broken, the output text will be compared to each of the over 100 language dictionaries available. This will go on until matching pattern is found. the

next section explains how DBX can be implemented to avoid detection when data is transmitted on open channel.

4.6 Embedding Ciphertext into an Image

Most of the encryption algorithms discussed above lead to ciphertext output. However, there are other messages that will not lead to text output, those messages are video, audio clips, and still images. If the message leads to a still image, then it is obvious that this message is just an image and not a text. Therefore there will not be much interest in the image. The basic principle is illustrated in Figure 4.2.

One of the proposals for this thesis is that after encrypting the message, using *Dynamic Block Encryption* (DBX) for example - which will be covered in detail in the next chapter - the next step is to take the output, watermark an image with the output and then encrypt the image. This makes the encrypted image appear as a normal photograph (if a positive decrypt is achieved) while encrypted data is actually embedded within.

The method can be used for not only encrypting files to an image, but also to other types of output, for example, the output can be a music file. Once in a music file, it is obvious to a listener that this is just a piece of music and nothing else has been *added* to it. This illustrates the one advantage that watermarking has over encryption, namely, that encrypted information flags the fact that important information is being communicated. Communicating information (encrypted or otherwise) by watermarking an entirely

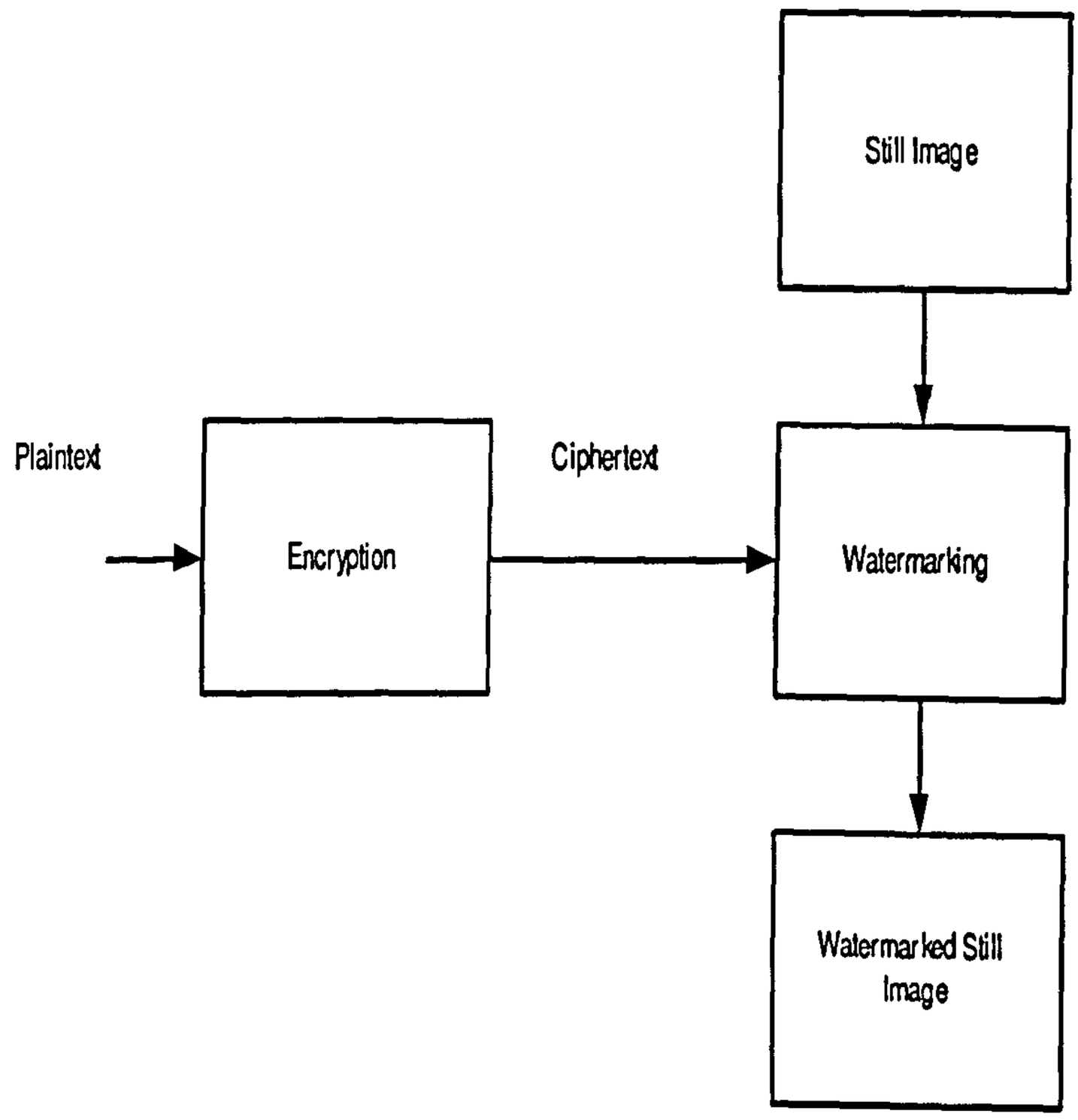


Figure 4.2: Block diagram showing basic function of encryption/watermarking system.

independent file (which is then encrypted or otherwise) provides a level of covertness that encrypted data cannot achieve.

In the following chapter, we consider the use of the watermarking technique discussed in this chapter for covertly exchanging keys by watermarking the ciphertext with the key used to generate the ciphertext.

Chapter 5

Dynamic Block Encryption Algorithm (DBX)

5.1 Introduction

This chapter details with the first of two encryption engines developed for this thesis. It considers each model in details, covering important aspects and properties. The algorithm takes data on a block by block basis and xor's it with random numbers obtained from running a Blum Blum Shub Pseudo Random Number Generator. The key for encryption is generated from the text itself and hence changes for each message and is communicated by watermarking the ciphertext using the approach developed in the previous chapter. This leads to the design of a unique and novel algorithm that forms the central kernel to the research reported in this.

The algorithm has been divided into five modules:

- The key generation module: This module generates a key from the input text; hence, the key generated is text dependent.

- The encryption module: This module encrypts text (or images or any other data in any format) using random sized text blocks.

- The key exchange module: After encryption the text (or image) is watermarked and the key is hidden within the text for transmission.

- The key extraction module: When this module is run, it removes the watermark and recovers decryption key.

- The decryption module: Decrypts the contents after removal of the watermark.

5.2 Two Modes of Operation

This software is designed to operate in two modes. In the first mode, we use one time parameter exchange, after the initial software setup, there is no exchange of parameters between the sender and the recipient. Apart from the encrypted file with the embedded key.

In the second mode, the parameters are passed using secure channel every-time a new file is transmitted.

In this chapter, we cover the main functionality of the DBX, which is independent on the mode of operation. The next chapter will cover in more details.

5.3 Key Generation Module

This module takes in the text, video or image file and generates a key which is used for encryption. For simplicity, all examples used here refer to text files; however, the program has been tested with still images and other data types. It has, also been tested with video. There are a number of ways to generate keys from the text. Four methods of increasing complexity have been considered.

5.3.1 Data Summation Method

After reading the file, a random noise is generated and padded at the beginning of the file. Then, all its characters are summed in ASCII mode. Thus, if a text file, for example, contained text *microsoft*, each character is first converted to its ASCII equivalent and then added up, i.e. m + i + c + r + o + s + o + f + t yields 109 + 105 + 99 + 114 + 111 + 115 + 111 + 102 + 116 = 982. Now each digit is again converted to its ASCII equivalent. So 9 + 8 + 2 becomes 57 + 56 + 50, which is 163. Finally 163 is converted into binary and forms 10100011. This binary string forms the key that is used to encrypt the text microsoft.

In the technique used here, it is difficult for the attacker to figure out the key on an unsecure communication channels. The key here is changed everytime the message changes. Hence, if an attacker manages to crack one message to get the key, once the next message comes a long, the key will be rendered useless. If the same message is used more than once, the key will be different, this is due to the introduction of random numbers used to pad the file.

Even though this method is simple, extracting the key from the text is a complicated idea in itself. Significant effort goes into securing the keys, while in this case the key is used only once, and the chances of guessing it are zero (in a reasonable amount of time).

5.3.2 Hash Algorithm Method

A hash function is a one way function. By using one way functions, the output of the function cannot be converted back to its original input form. i.e. the function is non-invertible. The analogy to a one way function is mixing of paints. If we have paints of two different colours, for example yellow and red, by pouring half of each content into a container, we end up with an orange colour. So orange was obtained as a result of mixing two different colours. Now, if we want to go back from orange to yellow and red, it is effectively impossible. This principle is concerned with the physical effect of diffusion which is a one-way process.

A number of applications make use of hash algorithms. For example, Unix passwords are stored in a file in hashed form. This makes it difficult even for Unix system managers to guess what the passwords are. Once a user wants to log in, he/she types his/her password which then gets 'hashed', and compared with the stored hashed password in the system. If the two hashes match, then permission to the user is granted. This method is not the most secure way of protecting passwords. If someone manages to get hold of the Unix password file, then he/she needs to try and guess different passwords, hash them, and see if the output can match any of the passwords. Once a match is found the hashed password will match the guessed password leaving the system open to an attack.

Hash algorithms are also used to authenticate downloaded files, and also if they have not been tampered with. Some Internet sites provide hash values for each file they post. If you download a file from any of those sites, you may compare the hash value of the downloaded file to the value you get after downloading and hashing. If they both match, then the download has worked correctly and there is no file corruption. On the other hand, if the download has partially failed, the hash value obtained after download will be different to the one of the web site. This is also helpful to avoid downloading files which have been replaced. For example, if, for some reason, someone has changed the file to be downloaded and replaced it with another file, of the same size, the hash value will still change indicating inconsistency.

Network managers have found 'perfection' in the use of hash function. In order to detect any changes to the critical system files, companies like *tripwire.com* have managed to utilise the hash functions features and design products to help network managers deal with any malicious attempts of breaking into the network. The manager has first to identify files which need to be changed and which remain static. Also he/she then identifies authorised and un-authorised changes. All tagged files have an associated hash value. If there is any changes on any of the files, the hash value will change. Any critical changes will alert the network manager. This is quite a useful tool, as it helps identify any critical file that has changed in case of a network attack [95] [94].

The method of hash functions to generate keys is used here to create a plain-text dependent key. There are a number of algorithms available for hash function. Some of the more common function are: SHA-0, SHA-256, SHA-512, MD4, MD5, HAVAL-128, and RIPEMD [63] [24]. The so called SHA

family, acronym for Secure Hash Algorithm, are created by the National Security Agency (NSA). The original algorithm SHA-0 was first published in 1993. It has since developed into SHA-1 (produces a 160 bit output for messages of any length less than 2^{64} bits), SHA-224 (produces 224-bits), SHA-256 (produces 256 bits), SHA-384 (produces 384 bits), and SHA-512 (produces 512 bits), sometimes called SHA-2. The last three variants were published by NIST in 2001. Message Digest, MD2, MD4, and MD5, have been developed by Ronald Rivest, co-founder of RSA Security. It takes plaintext input of arbitrary size and outputs a 128-bit message digest.

A MATLAB m-code function *key_gen_hashxx.m* is used to generate key using a hash function. The program can read the file of any format and any size. The m-function is used to run the hash function with a few changes to the output according to that needed by the DBX input. Once the file has been read, the first step is to make a system call to a hash program *rehash.exe*. The hash program reads the file and outputs a hash value of 256 bits. The hash value is stored in a variable name *hashv*. The value is stored as follows:

```
File: <testfile.txt>
SHA-256      : 1511A24E FC375C25 C44F5880 79D2C0A6 5B1ACEAD 18F390AF
               0AF2803D 20A1FD16
```

The output data is arranged in the above manner. It contains the file name, the type of SHA used, in this case SHA-256, a colon, and finally the hash value is arranged in groups of eight characters. In order to manipulate the above data, we need to strip all unwanted characters; this includes the file name, SHA type, colon, all spaces, carriage returns and line breaks. Once this is done we end up with pure hexadecimal numbers in the following form:

```
1511A24EFC375C25C44F5880.... 7B8A928E53164B7337B5F07AAA4243CA
```

Once the data is in this format, it is easier to manipulate, including the total length for the hash value is obtained. This is necessary to ensure that there has been no data corruption during the hashing procedure or stripping the unwanted data. By using SHA-256 we get 64 hexadecimal digits. Each hexadecimal digit represents 4 binary bits; we therefore have a total of 256 bits. This is a uniquely developed value only for this file. Of course since the number of files that can be used to generate hash values can be quite large, there is a possibility that more than one file can yield the same hash value. This is termed as collision. Even though it is possible to obtain similar values by hashing different type of files, the chances of this happening are quite minimal. To date, there is no known collision for this algorithm.

5.3.3 Wavelet Decomposition Method

The next method of key generation developed for this thesis is based on the *wavelet decomposition method.* Wavelets are correlation integrals that include a scaling parameter and were originally developed for the analysis of seismic data in the 1980s. Since then, they have found application in many areas of data analysis and have been discussed in Chapter 4. Based on the material presented in Chapter 4, the code for wavelet decomposition starts by reading a file and applying a wavelet with 7 levels. The approximation coefficients for the input signal are computed and the energy values computed and converted into a binary stream. The result is then used as a key for the encryption module. In particular, we apply wavelet decomposition using Daubechies wavelets with 7 levels using the m-code

```
[ca cl] = wavedec(au2(:,1), 7, 'db4')
```

that produces *ca* and *cl* which are the approximation and detailed coefficients. We then extract the detail coefficients at each of the 7 levels and add them, thus obtaining the energy coefficients. The energy coefficients are then added and rounded up to obtain the total energy coefficients. We again round up to the nearest integer the percentage energy of each set. Concatenating, we obtain a 150-bit binary string. This is finally converted to decimal in 50-bit segments:

```
keya = bin2dec(b_string(1:50));
keyb = bin2dec(b_string(51:100));
keyc = bin2dec(b_string(101:150));
```

By implementing the following functions,

```
keya = mod(keya,100000);
keyb = mod(keyb,100000);
keyc = mod(keyc,100000);
```

a final output is obtained that is a 15 digit decimal integer and is used in the encryption module as a key.

5.3.4 Convolution Method

The fourth and final method used for key generation is by applying the convolution integral. This is done by reading the file, and then generate a

seed. Once the seed is obtained, convolution is applied between the data and the seed. The output is then taken and modulated with a 15 digit decimal number.

5.4 Encryption Module

This module is based on encrypting data using a variable block size. All the encryption algorithms used to date employ either stream or block ciphers. With the stream cipher, the characters are taken one at a time and XORed with a data stream from a random number generator. With the block cipher a block of characters is ciphered depending on the block size.

The Dynamic Block Encryption (DBX) software developed for this thesis takes blocks of data and then separately encrypts them - see Figure 5.1.

Unlike block cipher algorithms, where the block sizes are static, thus, giving the cryptanalyst a point of attack, DBX does not have a fixed block size. The block size is dynamic and can be adjusted to any length, which cannot be pre-determined. Currently, the block size is fixed to a range between 5 and 50 bytes. The range selected is quite small compared to block ciphers which use 128, 256, or even 512 bits blocks. One reason for this is that the algorithm uses random number generators to generate numbers used in the XOR operation. The algorithm uses Blum Blum Shub (BBS) which is one of the secured random number generator. After every block, the generator is re-initialised and re-started. This enhances security since it makes it difficult to try and attack the random sequence that is generated.

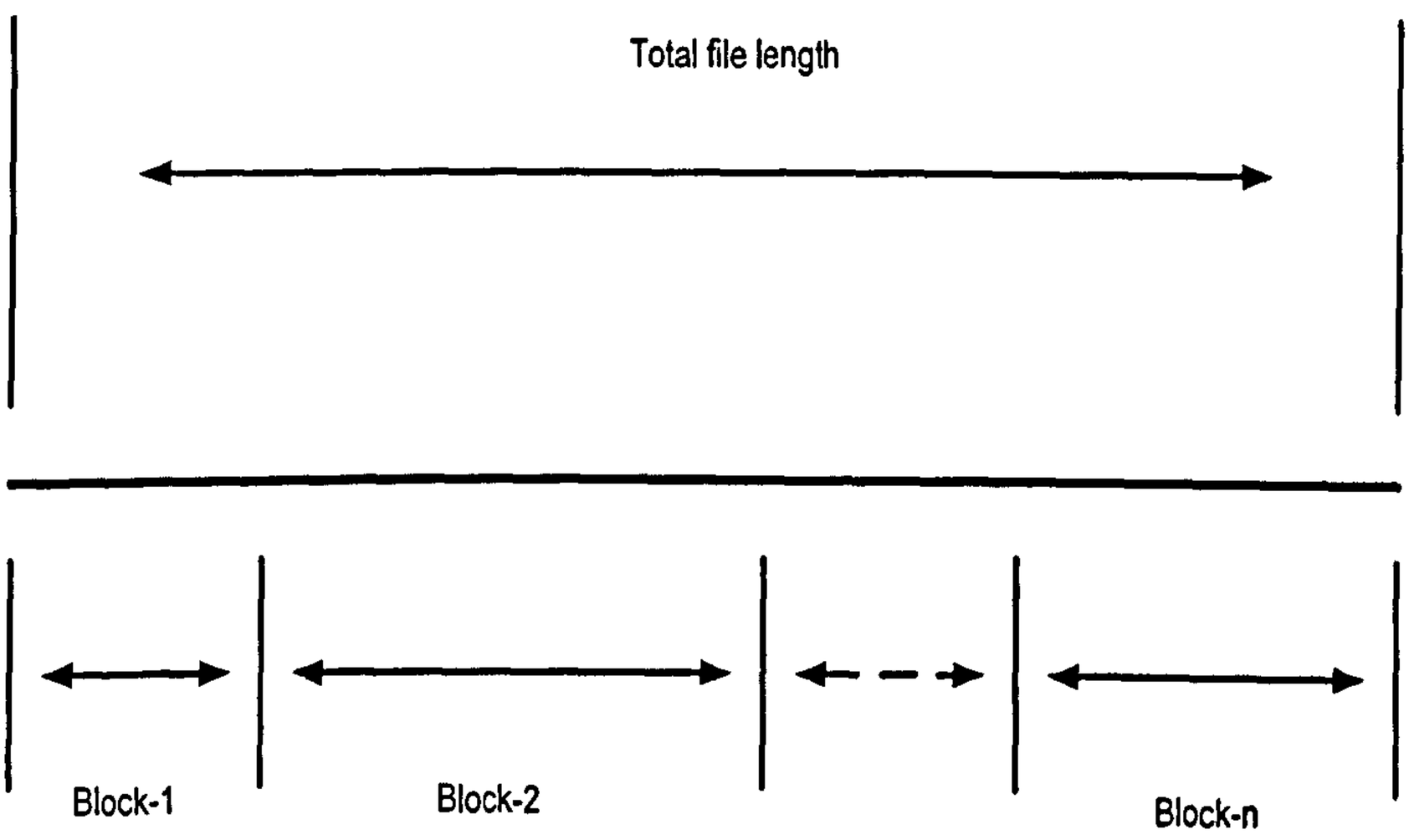


Figure 5.1: Illustration of dynamic blocking

Even though this block size modulation scheme is difficult to break, most of the random number generators are actually pseudo random number generators, i.e. they are deterministic. Note that all such generators have a pattern, which after a certain period is repeated - the characteristic cycle length. Secure generators will have a lower frequency pattern, while less secure generators have a higher frequency. [47]

The algorithm developed here works in the following way. After reading the file to be encrypted; either a text file, an image file, an audio file, or a movie clip, the program reads in a key as well. The key has already been generated in the previous module, which is text (or image, sound, video clip) dependent. The program calculates the length of the file in bytes which is needed for processing random numbers and deriving encryption blocks. A database of 10,000 prime numbers (which can be increased to over a million and manipulated using disk I/O and memory) is loaded into memory; again, loading it into memory causes the execution to be much faster. Once the initial values have been set-up the actual program execution follows.

Once the key is obtained, it will be broken down into a number of parts, for example, an 80 bit key can be broken down into 10 parts 8 bit each. Each of this part will be used as a seed for random number generator each time it is initialised. Once the key is exhausted, the seed will be obtained from the last block size. This process will continue until the whole file is encrypted.

The first random number generator used, a Linear Congruential Generator (LCG) of the form

$$x_{n+1} = x_n 7^5 \bmod P$$

where P is a prime number. This generator is used to randomly select two prime numbers from the 10,000 prime data base. We first feed in the parameters in order to run the above model. Here, x_0 is the initial value of x_n which is taken to be the key obtained after modulus 10,000. This number is used as an index to the prime database, and is used to pick a prime number from the database. For example, if the number obtained is 879, it picks a prime number corresponding to that prime in the database which is them used to set used P. The LCG is run twice to produce two random numbers. This same number, 879 is also used to randomly create the first block size. Since the maximum block size is 50, the number is modulated by 50. The two random numbers are again used as index to the prime number database to select two corresponding prime numbers needed to execute the Blum-Blum Shub (BBS) random number generator, i.e.

$$x_{n+1} = x^2 \bmod n$$

where

$$n = pq$$

Here, p and q are the prime numbers obtained above. The initial value of x_n is the same as the value used in LGC. This value is only used once, during initialisation; the seed changes for every new block. In essence, the encryption method is based on the application of a LCG to 'prime number seed' the BBS.

Having obtained the block size, the program then runs BBS to generate random numbers, depending on the size of a block, e.g. if the block size was 29, then we generate 29 random numbers. These numbers are then modulated by 255 to limit them to 8-bit ASCII characters. Once the first block of numbers has been generated, the algorithm fetches the same number

of characters from the plaintext file. These characters are XORed with the modulated random numbers to provide an encrypted text. As soon as the first block is encrypted, the seed for BBS is intialised, by taking the last value of the block length and using it to initialise the BBS generator. A different seed is used to initialize the BBS for each block of characters to be encrypted. This process continues over the whole length of the file. An example of the encrypted data generated by his process is given in Figure 5.3 which shows the 8-bit ASCII integer streams derived from that given in Figure 5.2.

5.5 Key Exchange Module

This module is based on the chirp coding method discussed in the previous chapter. A chirp is generated and used to hide information in a data. At the receiving end, the same chirp is re-constructed, data extraction being undertaken by negation. This concept is used to hide the key in the text. Depending on the text size, and the size of the key (password), a number of chirps are generated. For example, if the text size is 100 bytes, and the key length is 20 bytes, we use 5 chirps to cover the whole text.

Having encrypted the data, we are required to transmit it together with the encryption key. Under normal circumstances the key will be transmitted separately, but in this case the key is embedded within the encrypted data. This feature is the single most important contribution to the field as reported in this thesis. The key exchange module reads in the encrypted data and

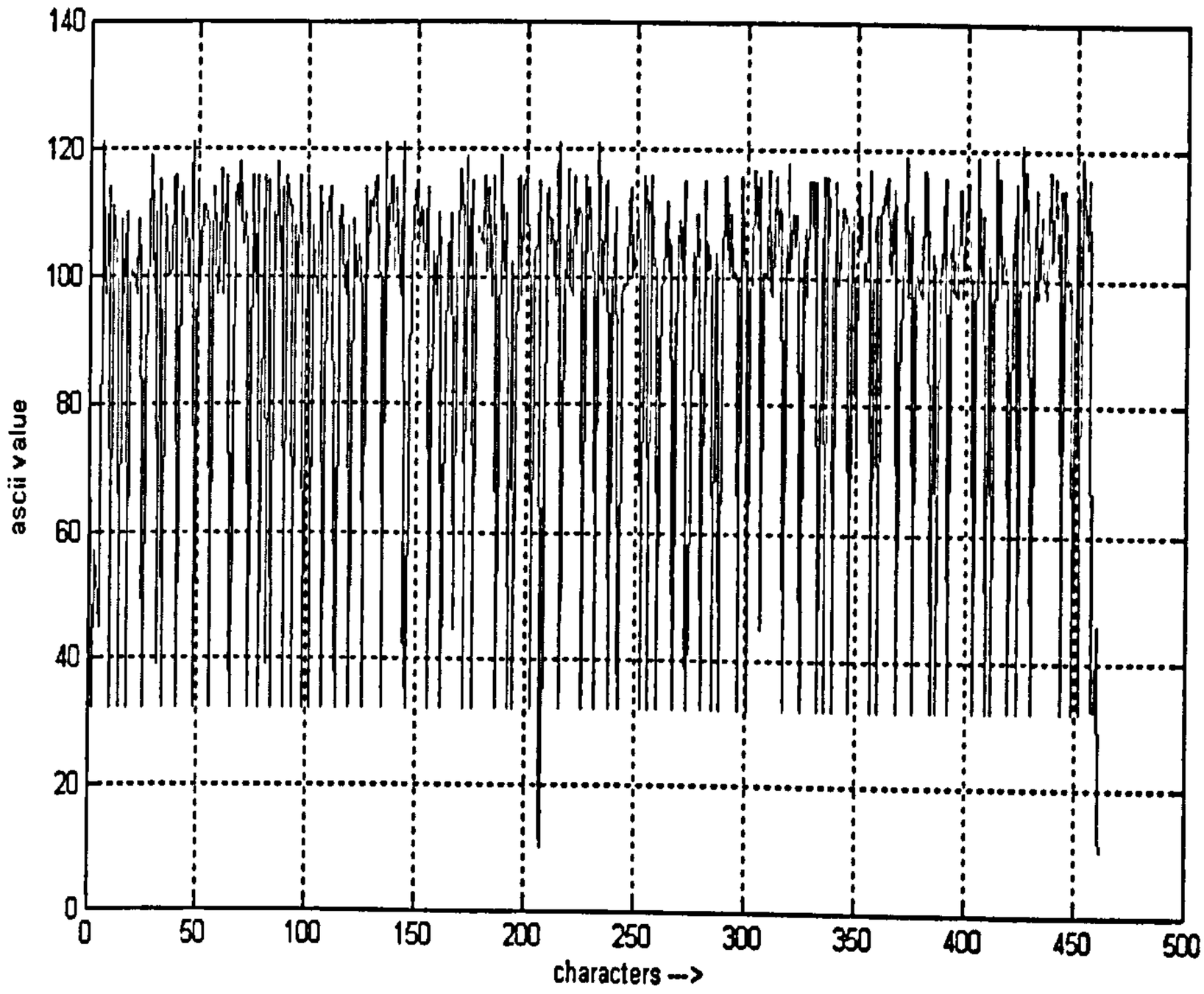


Figure 5.2: Plaintext 8-bit ASCII integer stream.

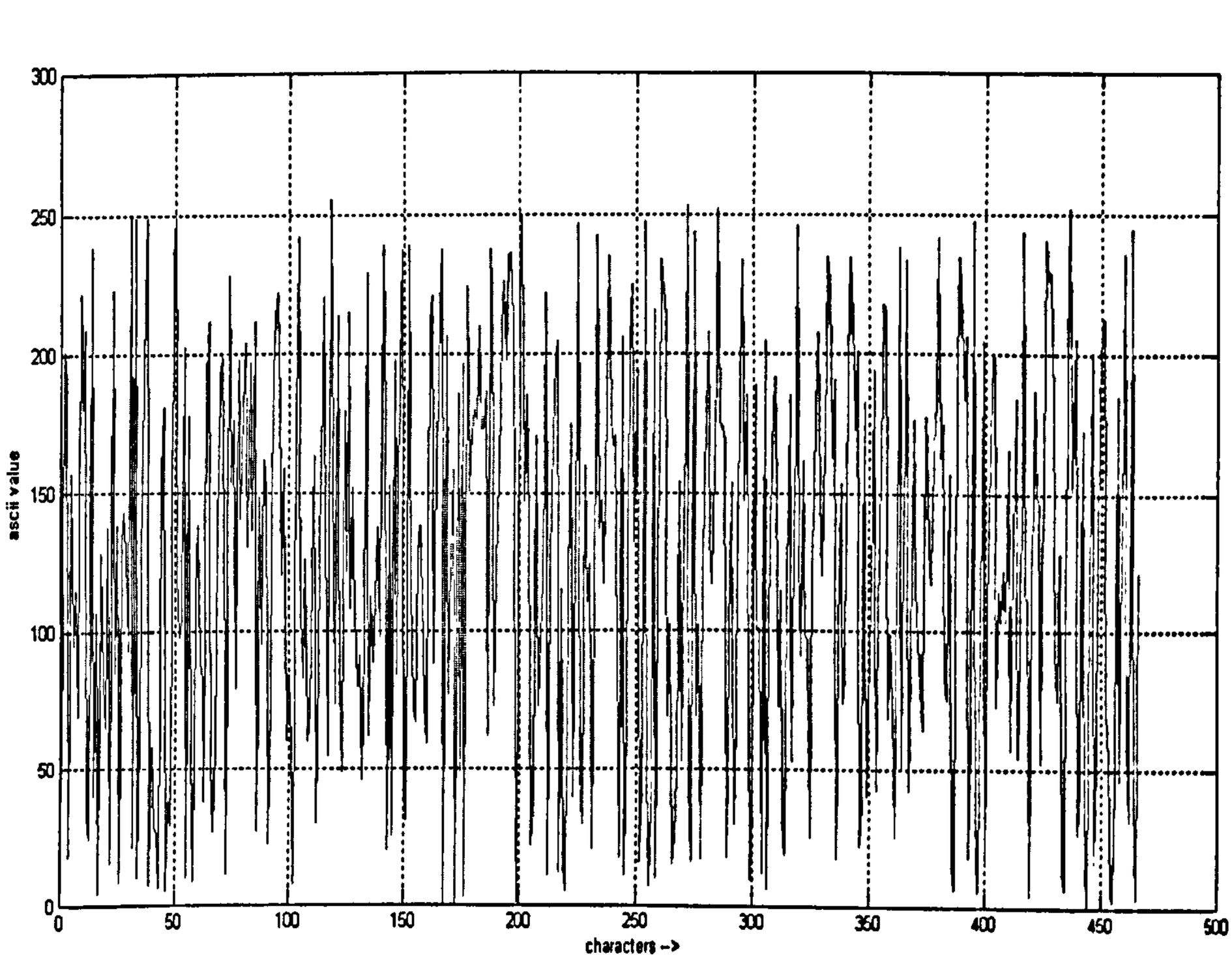


Figure 5.3: Encrypted data - 8-bit ASCII integer stream.

the key that was generated using *key-generation module.* This key is first displayed as a binary stream. The objective of the exercise is to multiply each binary bit with a chirp code. (We introduce the purpose and use of chirp code later on in the chapter.) For illustrative purposes, consider a key in the form of a binary stream 1101110001011010. The first step is to transform this binary stream into a series of one's and minus one's (as discussed in Chapter 4). Hence the above stream is transformed into 1 1 -1 1 1 -1 -1 1 -1 1 1 1 -1 1 -1. We then compute the chirp function. The first function is in the form of $sin(\alpha x_i^2)$. By varying the values of α and the length of the chirp, we obtain different forms of the chirp function. This feature is quite useful since it makes it hard for the attacker to guess the parameters used or the type of chirp implemented. The main affect of the chirp function is to modulate the frequency. Each chirp can be of a fixed pre-determined length. For example, Figure 5.4 shows a single positive chirp.

Clearly, if we multiply the above chirp by -1 we obtain a negative chirp as used to encode -1.

Since chirps are periodic and of a fixed pre-determined length, we can use a series of positive and negative chirps to represent a certain combination of binary strings. In this case we can represent the key binary stream so that the first few chirps of the binary stream

```
1  1  -1  1  1  -1  -1  1  -1  1  -1  1  1  -1  1  -1
```

will be as appears as illustrated in Figure 5.5.

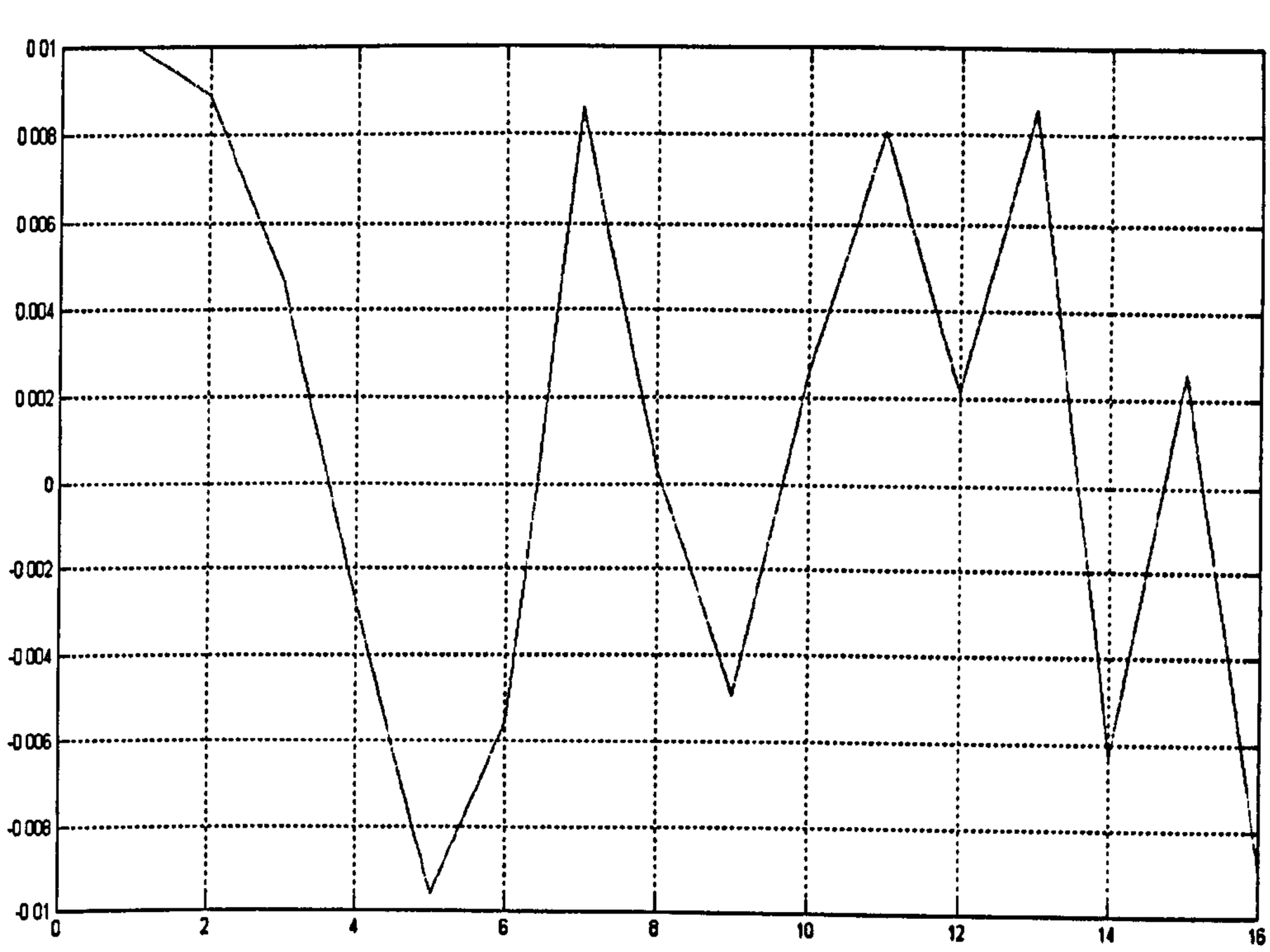


Figure 5.4: Example of a single positive chirp

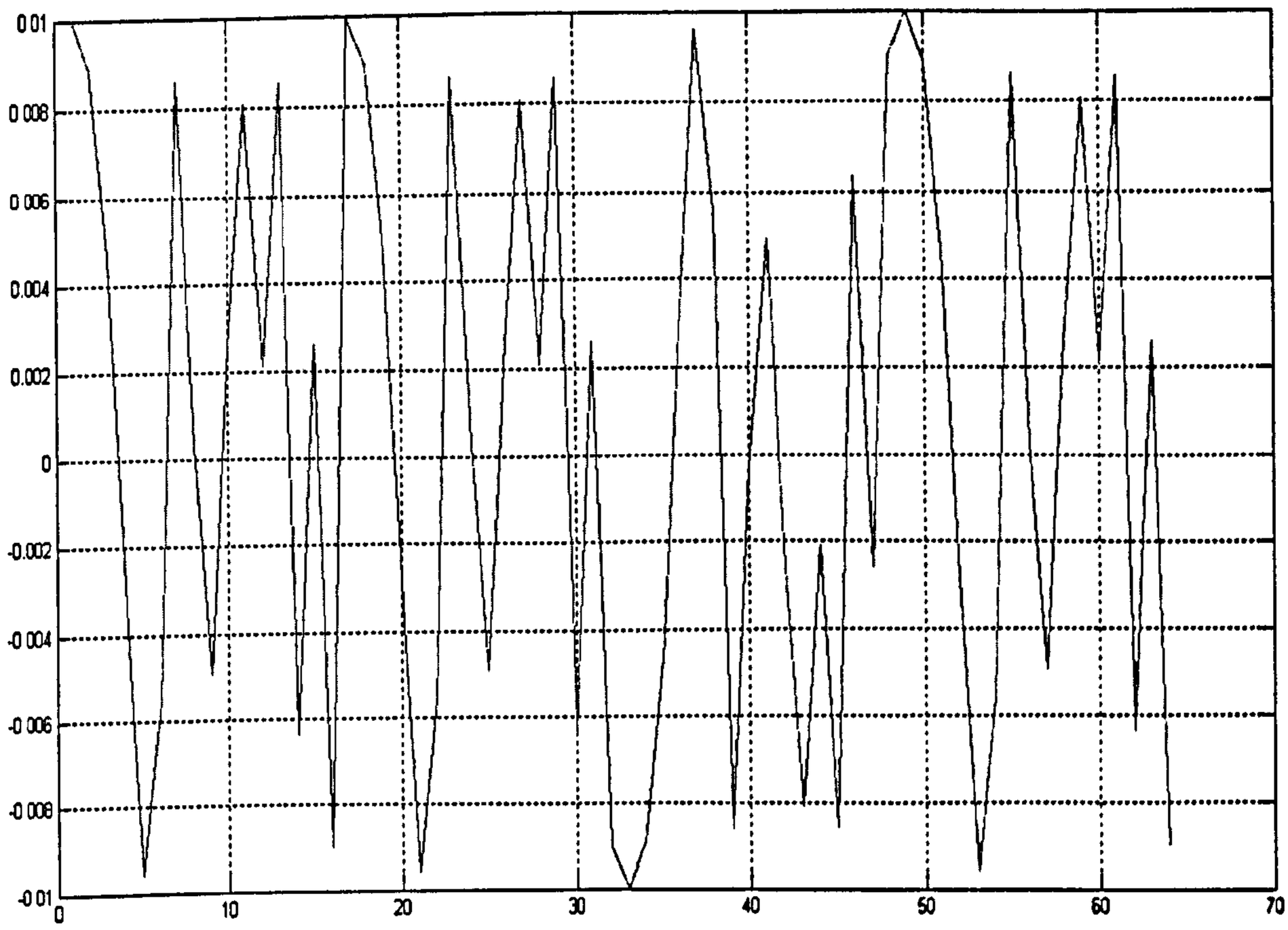


Figure 5.5: Example of a chirp stream consisting of four chirps

What we observe is the representation of a binary stream in terms of a 'chirp stream'. By multiplying each positive binary bit by a chirp we get a positive chirp; each negative binary bit results in a negative chirp. If we combine all the bits together we end up with a series of positive and negative chirps. In a sense, we have managed to transform the binary key into a sequence of positive and negative chirps - a series of continuous frequency modulated waveforms.

In order to perfectly embed the key in a file, we need to calculate the chirp length. Noting that the file length and key length have already been obtained from the modules discussed earlier, we can calculate the chirp length as

chirp length = (file length)/(key length).

If, for example, the file length is 200 bytes and the key length is 10 bytes, the chirp length will be 20 bytes long. This means that each chirp will have a length of 20 bytes and will vary depending on whether it is a positive or negative chirp.

The final output will be the same length as the file itself. The encrypted data, which has been read earlier, is normalised to limit the range of the data stream to 1, and thus makes it more accurate when it comes to using the xcorr function. For data above 1, the results are unpredictable. An example of the chirp stream in its entirety is given in Figure 5.6.

Finally, the data is added to the output of chirp functions - the chirp stream,

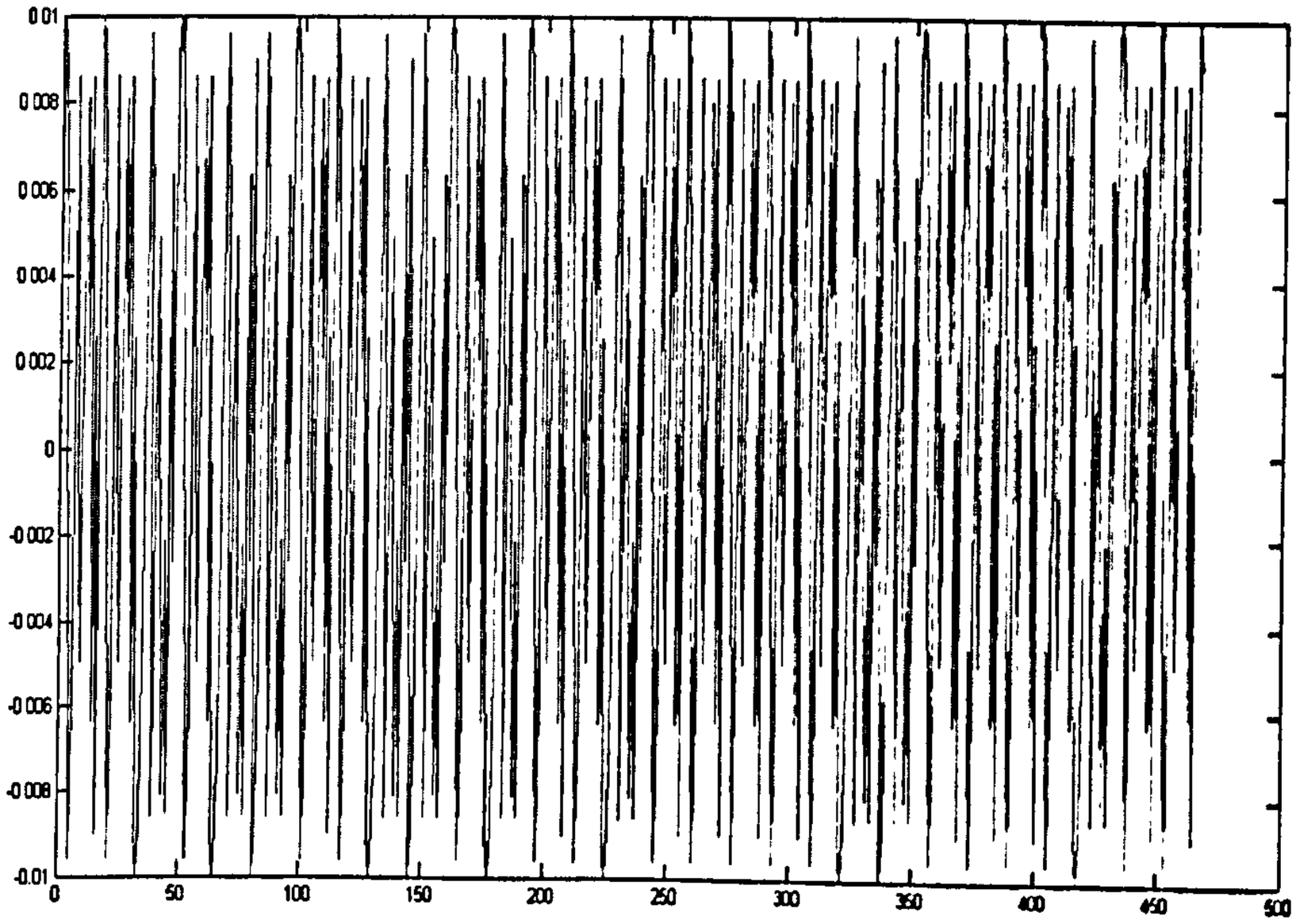


Figure 5.6: Complete chirp stream used for key exchange.

and can be prepared for transmission. This data contains the encrypted data and chirp coded key and is shown in Figure 5.7. Observe, that the chirp stream is, in effect, hidden in the ciphertext stream; the chirp stream is a small perturbation of the ciphertext.

5.6 Key Extraction Module

This module is run by the recipient. Once recipient receives an encrypted data with a hidden key, he/she then needs other parameters so that the key can be extracted from the ciphertext. The parameters needed here are the noise length and chirp length. Once obtained, the correct chirp length is then computed.

The encrypted data is normally received electronically through email. In this case, it is data with the hidden key. If an attacker manages to intercept this data, he/she will only have access to the data with a partial key. Without knowledge of the chirp length an attacker cannot extract the key. The recipient receives chirp parameters through specially customised hardware: Crypstic (see Appendix B). The Crypstic is a USB flash disk with a hidden memory. Each user is equipped with a crypstic that is tailored to their particular needs and applications. This type of encryption is not for general users and is normally used by company executives. Unlike other types of one-to-many encryption systems where everybody has access to the encryption engine, this type of encryption is accessible only to a few people in the organisations, where this software will be used.

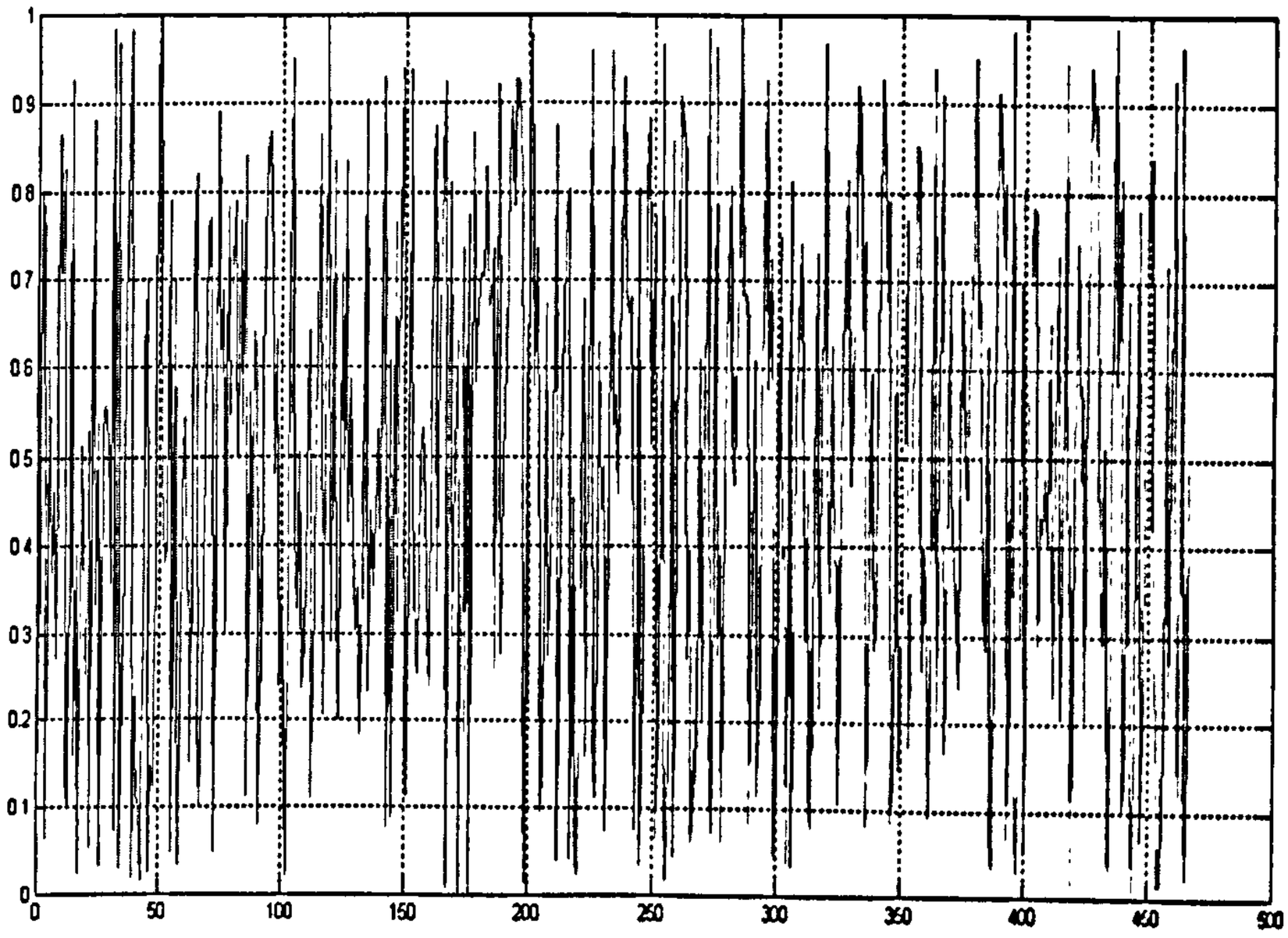


Figure 5.7: Ciphertext stream with embedded chirp stream.

Once both the file length and the key length have been received, the first step is to re-construct chirp length, which is given as

chirp length = (file length) / (key length)

Once the chirp length is obtained, the appropriate chirp can be constructed; a chirp that is exactly the same as the chirp created initially. All bits can be recovered with no loss of data. The data is then correlated with the newly created chirp which allows all original chirps to be identified within the signal. The correlation process identifies the positive and negative chirps of the chirp stream which in turn yields a positive or negative binary bit, respectively. The negative binary bits are converted to zero and the positive bits convert to ones. The final output is the key to be used for decryption.

5.7 Decryption Module

This module receives a key from the previous module. The encrypted file contains the file itself with the embedded key. When the key is extracted, we are left with encrypted data which can be then be decrypted. The decryption program is similar to the encryption module in functionality.

5.8 Discussion

The use of chirp coding can be applied quite generally in a way that is, in principle, independent of the encryption engine. In this chapter, the encryption engine uses a LCG to 'drive' BBS by modulating the prime numbers

that BBS relies upon on a dynamic block basis. Thus, the entire approach is based on modulation, i.e. prime number modulation, dynamic block size modulation and frequency modulation (chirp coding). However, in addition to the modulation method considered, another approach can be developed that is based on modulating the encryption algorithms themselves rather than modulating the parameters that drive' them as considered in this chapter. In order to design such an encryption engine, it is not possible to use prime number based random number generators because of the inherent limitation placed on the form of the generators through the use of prime numbers and their properties. Instead, we resort to the application of deterministic chaos [69]. Deterministic chaos has three fundamental advantages:

(i) it dispenses with prime numbers altogether and therefore eradicates any prime number based attacks;

(ii) it provides a wealth of iteration function sequences that can be literally 'invented' for applications in cryptography;

(iii) because of (ii) above, it provides the ability to design encryption engines that are multi-algorithmic, i.e. multi-dynamic algorithm selection.

The basis for this approach to encryption engine design is discussed in the following chapter.

Chapter 6

M-Code Development and Test Results

6.1 Introduction

In previous chapters we have considered the overall methodology developed for this research thesis. This chapter covers in greater detail each of the modules involved. All modules are interrelated through data I/O and are therefore weakly coupled. The functions and modules have been prototyped in MATLAB and the m-code is discussed in Appendix A which relates the m-code to that given in the CD at the back of this thesis. In this chapter, we highlight the structural operation of the system referring to the lines of the m-code that are sourced in its entirety on the CD.

The software can be operated on two modes. In the first mode, there is a one time agreement of parameters to be exchanged between the sender and the recipient. These parameters are fixed and once exchange of files

takes place there is nothing passed except the file itself. Of course the key, which changes for every transmission, is also embedded within the encrypted text. This allows for greater security because the method for extracting the key from the text is so complex. Further, in order to extract the key using brute force attack it requires a significant amount of computer power and time. Even though there is a one time agreement of parameters, there is an option for sender to change these parameters. Parameters can be changed at different intervals depending on the required security level, some usage may require changes to take place every 90 days. For more secure environment, parameters may be changed every day.

The encryption and decryption process takes place in the following sequence:

Sender

- Selects parameters for encryption process, or accepts current parameters.
- Runs key generation module. There are four options to key generation.
- Runs encryption module.
- Runs key exchange module.

Recipient

- Runs key extraction module
- Runs decryption module

6.2 Fixed Length Parameters

In this chapter, we start with the details on key generation of the first mode of operation. In both modes, as stated earlier, four approaches in key generation are discussed. Even though the methods of key generation are different, the output is similar. Each of the four modules has been configured to produce 80 bit key length.

6.2.1 Parameter Selection

The program comes with pre-configured parameters. These parameters are used for padding length, and chirp initialisation.

The chirp function in matlab needs four parameters to run

$$y = chirp(t, f0, t1, f1)$$

. The value of t determines the length (period) of the chirp itself. This is automatically calculated as the ratio of file length to the key length, will therefore change for different file lengths. The functions of $f0$, $t1$, and $f1$ will be explained later in the chapter in details.

In parameter selection, the following process takes place:

1. Read in the existing parameters.

```
fid = fopen('parameters.txt'. 'r');
a = fread(fid,'int64');
fclose(fid);
```

2. Prompt the user to accept or change any parameter. This goes for all other parameters.

```
disp(['current f0: ' num2str(a(2))]);
answer2 = input('Would you like to change f0 (y/n): ', 's');
if answer2 == 'y'
   new_f0 = input('Enter new f0: ', 's');
   a(2) = str2num(new_f0);
end
```

3. Save new values to a file.

```
fid = fopen('parameters.txt', 'w');
fwrite(fid,a,'int64');
```

6.2.2 Summation Method

This program takes the value of noise length, generates noise, and pads the noise at the beginning of the file. It finally generates a key by summing up all characters in a file and manipulates them.

1. Accept a file to be encrypted.

```
filename=input('Enter a plaintext file (any format): ','s');
fid = fopen(filename,'rb');
data = fread(fid);
fclose(fid);
```

2. Generate noise based on previous value of noise length. The minimum value is 10, 145 is the maximum size a random number can be generated. Both chosen arbitrarily, and after some random testing.

```
noise = 10+round(145*rand(1,noise_length));
```

3. Pad the noise value in front of the data (concatenation).

```
new_file = [noise,data'];
```

4. Add up all characters in a file.

```
data = new_file;
key_sum = sum(data);
```

5. Break them up into small sizes and add up again, this is done to make the output more random and unpredictable.

```
key_sum = sum(data);
key_sum2 = sum(data(1:5000));
key_sum3 = sum(data(5000:10000));
key_sum4 = sum(data(10000:15000));
```

6. Convert to string

```
key_sum = num2str(key_sum);
key_sum2 = num2str(key_sum2);
key_sum3 = num2str(key_sum3);
key_sum4 = num2str(key_sum4);
```

7. Get the length of the key in decimal,

   e.g. if key_sum = 3451, key_length = 4

```
key_length = length(key_sum);
key_length2 = length(key_sum2);
key_length3 = length(key_sum3);
key_length4 = length(key_sum4);
```

8. Get ASCII value for each of the decimal number.

```
key_ascii = abs(key_sum);
key_ascii2 = abs(key_sum2);
key_ascii3 = abs(key_sum3);
key_ascii4 = abs(key_sum4);
```

9. Add them up

```
tot_key_ascii = [key_ascii, key_ascii2, key_ascii3, key_ascii4];
tot_key_length = key_length + key_length2 + key_length3 + key_length4;
```

10. Convert to binary.

```
total_key_bin = [];
for i = 1:tot_key_length
    key_bin = dec2bin(tot_key_ascii(i));
    total_key_bin = cat(2,total_key_bin,key_bin);
end
```

11. The final output is an 80 bit binary key.

6.2.3 Convolution Integral method

The key generated by this module is done by creating a small set of random data, then convoluting it with the sum of the total data. The main difference lies in the method used, otherwise everything else remains the same.

1. Add up all characters in a file, after padding with noise.

```
key_sum = sum(new_file);
```

2. Get the decimal length

```
s = length(num2str(key_sum));
```

3. Generate a random number based on the length.

```
x = rand(1,s);
x = abs(x(1));
```

4. Do a convolution taking the value of *new_file* and x.

5. The rest of the program is manipulating the output value (xx) and finally converting to 80 bit binary stream.

```
xx = convn(new_file,x,'valid');
```

6.2.4 Wavelet Decomposition Method

In this module Daubechies wavelets are used to decompose the plaintext into two coefficients: approximation coefficient and detailed coefficient. The values of the detailed coefficient are extracted and used to generate the keys (see chapter 5). The output is set to be 80 binary bit stream.

6.2.5 Hash Function Method

In this method, a hash algorithm is used to create keys. The algorithm used here is the *secure hash algorithm* with 256 bits, SHA-256, mentioned earlier in this thesis (see Chapter 3). A hash function H is a transformation that takes an input m and returns a fixed-size string, which is called the hash value h (that is, $h = H(m)$).

In this module, we use a readily available hash function algorithm. It is then incorporated into a MATLAB routine. The input parameters are formatted in such a way as be suitable for inputting in the module. The output given is also modified in such a way as to make it compatible with the rest of the module.

1. Read in a file, concatenate with the noise.

2. Run hash function giving the above file. The output hash value will be stored in variable *hashv* and the status in variable *stat.*

```
[stat, hashv] = system(['rehash -none -sha-256 ' filename]);
```

3. The output will be in this format.

```
File: <hashtest2.m>
SHA-256      : 1511A24E FC375C25 C44F5880 79D2C0A6 5B1ACEAD 18F390AF
               0AF2803D 20A1FD16
```

4. Compute the actual hash length.

```
hash_length = length(hashv);
```

```
hash = hashv(26+(file_length):hash_length)';
new_hash_length = length(hash);
```

5. Strip the header, spaces (ASCII value 32), line feeds (ASCII value 10) and carriage returns (ASCII value 13).

```
tot_hash_value = [];
for i = 1:new_hash_length
    if (hash(i) ~= 32 && hash(i) ~= 10 && hash(i) ~= 13)
        hash_value = hash(i);
        tot_hash_value = [tot_hash_value;hash_value];
    end
end
```

6. Make sure the length is correct.

```
tot_hash_value = tot_hash_value(1:64);
```

7. Convert to binary.

```
hash_bin = dec2bin(hex2dec(tot_hash_value(1:64)));
```

8. Set the output to 80 bits

```
watermark_binary = hash_bin(1:80);
```

6.3 Encryption Module

The background to the design of this module has been explained in detail in Chapter 5. Here, we present the m-code for the encryption module. Since this is a symmetric encryption algorithm, the encryption and decryption modules are both the same.

By this stage, the file will have been read in. The key already generated and ready for encrypting the file.

1. Take the key length, which is 80 and divide it into equal number of parts. This can be achieved by using modulus arithmetic. In this case, the seed length is 13, any suitable number can be used.

```
seed_length = 13;
rem = mod(length(watermark_binary),seed_length);
```

2. Discard the remainder, obtain exact multiples of the key.

```
for bin_key_counter = 1:seed_length:length(watermark_binary)
    bin_key_counter;
    counter = counter + 1;
    yy = watermark_binary(bin_key_counter:bin_key_counter+seed_length-1);
    y = bin2dec(watermark_binary(bin_key_counter:bin_key_counter+...
        seed_length-1))
    y_tot = [y_tot;y];
end
```

3. Get the first seed to be used for Linear Congruential Generator

```
seed = y_tot(1);
```

4. Load the prime number database. The database currently hold the first 10,000 prime numbers. These can be increased and partly read as I/O. Once prime numbers are loaded, let the variable *seed* be an index to the first prime. For example, if *seed* was 234, it should pick 1481 from the prime number database which is 234th prime.

```
prms = load('primes3.m');
prm = prms(seed);
```

5. Set the initial value to be used in Linear Congruential Generator

```
xn = seed;
```

6. Generate two random numbers using the linear congruential generator.

```
for i = 1:2
    xn = 1 + mod(xn*7^5,prm);
    xn = 1 + mod(xn,9999);
    tot_xn=[tot_xn;xn];
    count = count + 1;
end
```

7. Compute two blum primes. By using blum primes, the strength of the random numbers sequence produced is enhanced. Blum primes, p and q are chosen so that $p \bmod 4 = 3$, and $q \bmod 4 = 3$. A prime number is read and tested; if it is found to be a blum prime, it is stored in a variable, if not, the next number is read. The process continues until two blum primes are obtained.

```
tot_blum = [];
blum_count = 0;
for i = 1:length(prms)
a = tot_xn(1);
   if i == 1
    prm1 = prms(a);
   else
    if (a + 1) > length(prms) - 1;
      prm1 = prms(round(length(prms)/2));
     else
      prm1 = prms(a+i);
     end
end

if mod(prm1,4) == 3
    blum_prime = prm1;
     blum_count = blum_count + 1;
    tot_blum = [tot_blum;blum_prime];
if blum_count == 2
     break
    end
  end
  if blum_count == 2
      break
   end
end
tot_blum;
```

```
p = tot_blum(1);
q = tot_blum(2);
```

8. Compute block length. The size has been set from 5 to 50 characters.

```
block_length = 5 + mod(xnb,46);
```

9. *Counter* stores the number of values that the key has been split into. For example, if the key length is 80 bits, and the key has been split into 8 different values of 10 bits each, the *counter* has a value of 8. The first value is allocated to the a variable *xnb* and so on.

```
if main_i <= counter
    xnb = y_tot(main_i);
```

10. Move the value to be used by Blum-Blum Shub

```
xnb2 = xnb;
```

11. Run BBS to generate random numbers. The set of random numbers generated is the same as the block length. Since the numbers can be huge, they are normalised to 8-bit ASCII so that they do not exceed 255.

```
for i = 1:block_length
xnb2 = mod(xnb2^2,p*q);
xornum = xnb2;
  xornum = 6 + mod(xnb2,250);
  tot_xornum = [tot_xornum;xornum];
end
```

12. Continue reading the data as long as the remaining text is larger than the assigned block length.

```
flag = 1;

if text_length > block_length
block_data = data(offset:offset+block_length-1);
  offset = offset + block_length;
```

13. Once the length of the remaining text becomes less than the assigned block length, a new block length is created that is the same as the remaining text length. The flag is set to zero.

```
else
block_length = text_length;
new_block_length = block_length;
  block_data = data(offset:original_text_length);
  new_tot_xornum = (tot_xornum(1:length(block_data)));
  tot_xornum = new_tot_xornum;
flag = 0;
end
```

14. Compare the text length to the block length and if they are the same, end file.

```
new_text_length = text_length - block_length;
if new_text_length == 0
new_text_length = text_length;
end
```

15. Do a bit exclusive-or for each block between the random numbers *(tot_xornum)* and plaintext *(block_data)*. Check the flag everytime; once it is set to zero, the *break* statement 'breaks' out of the loop. The resultant output is an encrypted data.

```
crypted_text = bitxor(tot_xornum, block_data);
total_crypted_text = [total_crypted_text; crypted_text];
if flag == 0
   break
  end
```

6.4 Key Exchange Module

A major challenge for cryptographers is key exchange. In this research a principal concentration has been the mechanism for key exchange through application of chirp coding as discussed in Chapter 4. Once data has been encrypted, the key is sent to the recipient embedded in the encrypted text itself. The recipient, upon receiving the ciphertext, extracts the key from the data by correlating it with a replica of the chirp function. The m-code for embedding the key this way is outlined below.

The module starts by reading the input file and undertaking the necessary initialisations. The principal components of the m-code are then as follows:

1. The variables *file_length* and *length_watermark* (key length) are computed in previous modules. Here, they are used to compute *chirp_length*.

```
chirp_length = floor(file_length/length_watermark)
```

2. In order to multiply each binary bit by a chirp, we cannot introduce zeros. Hence this script is designed to transpose all zeros to -1.

```
for j = 1:length_watermark
if str2num(watermark_binary(j)) == 0
   x(j) = -1;
  else
    x(j) = 1;
  end;
end;
```

3. Initialise the chirp function. Here, we use a log chirp instead of a linear chirp in order to reduce aliasing, i.e. under sampling.

```
t = 0:1/chirp_length:1;
y=chirp(t,0,1,40,'log');
```

4. Multiply each binary bit by a chirp, so +1 will yield a positive chirp and −1 will yield a negative chirp as discussed in Chapter 4.

```
znew = 0;
for j = 1:length_watermark
z=x(j)*y;
  znew=cat(2,znew,z);
end
```

5. Divide the data by 255 so that the maximum value of 1; equivalent to applying uniform normalisation.

```
data = data./255;
```

6. Adds the two signals together; *new_data* is the same size as the original signal. This variable is written into the output file to be used for transmission. In effect, the recipient receives and encrypted data together with a series of positive and negative chirps.

```
new_data = (znew'+data);
```

6.4.1 Key Extraction Module

In this module, the embedded key is extracted and then passed on to the decryption module. In order to extract the key, the user must have exact parameters used by the recipient, to reconstruct the chirp. Once the chirp is reconstructed, it is then correlated with the encrypted text for key extraction.

1. Open a file and read in parameters needed to construct a chirp.

```
fid = fopen('parameters.txt', 'r');
a = fread(fid,'int64');
fclose(fid);
```

2. Assign read in values to the chirp function

```
f0 = a(2);
t1 = a(3);
f1 = a(4);
```

3. Compute *chirp_length*.

```
chirp_length = floor(file_length/length_watermark);
```

4. Compute t.

```
t = 0:1/chirp_length:1;
```

5. Initialise chirp function.

```
y=chirp(t,f0,t1,f1,'log');
```

6. Correlate to recover the key from ciphertext. The key is recovered in terms of a set of positive and negative integers. The interest does not lie in the integers themselves, but in their sign; positive values are denoted by 1 and negative numbers by -1.

```
k=1;
for i=1:length_watermark
  yzcorr=xcorr(new_data(k:k+chirp_length-1),y,0)
  k=k+chirp_length;
  r(i)=sign(yzcorr);
end
```

7. Recover bit stream. Assign 1 to a positive number and -1 to a negative number.

```
for i=1:length_watermark
    if r(i)==-1
        recov(i)=0;
    else
        recov(i)=1;
    end
end
```

8. Compute *znew*. *znew* is reconstructed by multiplying the chirp signal with each value of the key, in this case, with each +1 and −1 stored in r, which was obtained from the correlation of the two signals.

```
znew = 0;
for j = 1:length_watermark
    z=r(j)*y;
    znew=cat(2,znew,z);
end;
znew = znew(2:length(znew));
```

9. Separate the data, leaving encrypted data without key or chirp signal.

```
data = new_data - znew
```

10. Multiply by 255 to bring back the original value, for decryption by the next module.

```
data = round(data.*255);
```

6.4.2 Decryption Module

This module is the reverse of the encryption module, since we are working with a symmetric cipher. The only difference is that after decryption, noise is subtracted from the file and the result is the actual decrypted file.

6.4.3 Input Parameters

The following tables show parameters needed for input to the program. These parameters have been thoroughly tested. When the program is first launched,

the user is given an option to change or keep the existing parameters.

f0	t1	f1	f0	t1	f1	f0	t1	f1
0	1	40	1	10^4	65	4	10^{18}	65
0	1	50	1	10^5	65	4	10^{18}	70
0	1	60	1	10^6	65	4	10^{18}	80
0	1	65	1	10^7	65	4	10^{18}	100
0	2	65	1	10^8	65	4	10^{18}	1,000
0	5	65	1	10^9	65	4	10^{18}	10^4
0	10	65	1	10^{12}	65	4	10^{18}	10^6
0	20	65	1	10^{18}	65	4	10^{18}	10^9
0	50	65	2	10^{18}	65	4	10^{18}	10^{12}
0	100	65	3	10^{18}	65	4	10^{18}	10^{18}
0	1,000	65						

Table 6.1: Chirp parameters tested for successful key exchange.

6.5 Changing Parameters Mode

As we have mentioned at the beginning of this chapter, DBX can be run in two different modes, the first mode is a one time parameter exchange mode where once the parameters are set, there is no need to change them. The same set of parameters are used by sender and recipient. The key, of course is different for every message transmitted.

All functionality for both modes is the same. However, due to the way they are used, there are slight differences in some modules which we will describe below. To start with, program runs in the following sequence.

Sender

- Runs key generation module. There are four options to key generation.
- Runs encryption module.
- Runs key exchange module.
- Runs save parameters module.

Recipient

- Runs load parameters module.
- Runs key extraction module.
- Runs decryption module.

6.6 Key Generation Module

In key generation modules, the padding size varies. So the first thing is to generate the actual padding size, 10,000 bytes, and then generate 10,000 random numbers. For each run, the size will be randomly generated and hence different.

Generate random numbers, minimum 99983 characters. This number has been arbitrarily chosen. The aim is to start with a padding size of around 100,000 characters. Since 100,000 is a *good* number, it is always better to avoid. We have therefore chosen 99983 + 10000 which gives us 109983 characters. Unlike the previous mode, in this mode user cannot change these values, hence they are randomly selected.

```
y=99983+round(10000*rand);
noise = 10+round(145*rand(1,y));
```

This is the same for all the four modules used in key generation. The rest of the program is more or less the similar.

6.6.1 Encryption Module

There are no changes on this modules in both the modes.

6.6.2 Inserting Watermark

The main difference here is that the chirp parameters are fixed. They can only be changed if the need for enhancing security of the software arises, which is quite rare.

Initialise chirp.

```
t = 0:1/chirp_length:1;
y = chirp(t,0,1,40,'log');
```

6.6.3 Parameters

Chirp length and padding length are saved by the sender and loaded by the recipient on the other end.

6.6.4 Key Extraction Module

In this module, the chirp parameters remain the same except for chirp length. Therefore no major changes.

6.6.5 Decryption Module

The main difference in the decryption module is that the noise subtracted from the total file received is dynamic, so the value will change depending on the random number generated to create padding noise in the first place.

6.7 Analysis

The following tables show the time/speed relationship for a range of test cases. In both cases experimental tests were carried out. The data was obtained after running tests on a 2Gh Pentium 4 Laptop with 1 GB RAM. Table 6.4 shows a test on key generation, key embedding and key extraction. Table 6.5 shows a test for encryption, the decrypt is bit for bit perfect.

The average speed for the 18 items in table 6.4 is:

$$3050 \div 18 = 169 \text{ kbytes/s}$$

$$169 \times 8 = 1352 \text{ kbits/s}$$

$$= 1.35 \text{ mb/s}$$

File type	Size (bytes)	CPU Time	kbytes/s
Text	31,007	0.32 s	97
Text	111,659	0.60 s	186
Text	209,608	1.02 s	205
Jpeg	71,189	0.46 s	155
Jpeg	121,943	0.66 s	185
Jpeg	435,765	2.04 s	214
MS Word	25,600	0.33 s	78
MS Word	35,328	0.34 s	104
MS Word	321,536	1.07 s	301
Pdf	95,018	0.50 s	190
Pdf	1,443,863	6.40 s	226
Pdf	4,353,974	32.74 s	133
Exe	69,120	0.49 s	141
Exe	2,608,128	16.03	163
Exe	8,466,464	61.72 s	137
Avi	1,598,284	6.81 s	235
Avi	3,633,932	31.36 s	116
Avi	5,479,452	29.84 s	184

Table 6.2: Test for generating, embedding and extracting the key.

The encryption module includes added noise. For example if the file size is 60k and the noise is 100 k, then the new size will be 160k.

File type	Size (bytes)	CPU Time	kbytes/s
Text	111,659	79.73 s	1.40
Text	209,608	90.36 s	2.32
Jpeg	71,189	30.20 s	2.36
Jpeg	121,943	80.23 s	1.52
Jpeg	435,765	248.36 s	1.75
MS Word	25,600	60.47 s	0.42
MS Word	35,328	60.08 s	0.59
MS Word	321,536	177.92 s	1.81
Pdf	95,018	71.73 s	1.32
Pdf	1,443,863	28.28 s	51.06
Exe	69,120	64.39 s	1.07
Exe	2,608,128	63.13 s	41.31
Avi	1,598,284	25.83 s	61.88
Avi	3,633,932	31.36 s	115.88
Avi	5,479,452	29.84 s	183.63

Table 6.3: Test for encryption/decryption.

Average speed for encryption/decryption:

$$468.32 \div 15 = 31 \text{ kbytes/s}$$

$$31 \times 8 = 248 \text{ kbits/s}$$

6.8 Running DBX with Crypstic

The use of deterministic chaos for designing multi-algorithmic encryption engines has been discussed in Chapter 3. For the DBX system developed here, it can be used as a parameter file exchange system based on application of a user specific USB memory stick - Crypstic. By utilising the Crypstic, DBX functionality can also work with any type of algorithm, hence the main theme is the key exchange mechanism.

6.9 Security of DBX

DBX can be considered to be relatively secure and, coupled with the application of the Crypstic for parameter exchange, highly secure. Many factors a play role in enhancing DBX security.

Attack on DBX - One way DBX can be attacked is by brute force. The system is currently configured for 140 bits key space, but it can be increased. Conventional attacks strategies which exploit the fact that the algorithm is publish can not be applied in the same procedural way. This is because the approach considered in this thesis exploits the principle of multi-dynamicism where the encryption process is designed with a view to the constantly modifying the algorithms/parameters/keys etc.

Key Exchange. The method used for key exchange significantly enhances the security of the system. The key is passed to the recipient embedded in the encrypted data itself. It is very difficult to detect that two signals have been superimposed during the transmission. Even if the attacker is equipped with knowledge of the existence of the key within the signal, extraction of

the key is relatively complex and will not be easily accomplished.

Chirp Function. Once the key is generated and converted to binary, a chirp is initialised. Each binary bit is multiplied by a chirp. A 1 will produce a positive chirp and a 0 will produce negative chirps These chirps are then concatinated together to produce a signal the same length as the file itself.

The chirp function in *matlab* generates a swept-frequency cosine (chirp) signal. The chirp block outputs a swept-frequency cosine (chirp) signal with unity amplitude and continuous phase. To specify the desired output chirp signal, its instantaneous frequency function must be defined, also known as the output frequency sweep. The frequency sweep can be linear, quadratic, or logarithmic, and repeats once every sweep time by default.

By using chirp functions, we can obtain a variety of chirps which can be implemented differently. The main use of a chirp in DBX is to hide the key. Thus, given a chirp function with certain parameters, the recipient needs to construct the same parameters in order for him/her to recover the key. The sender has a wide range of values to use for parameters which the recipient has to have available.

The MATLAB chirp function used here has the general form

$$y = chirp(t, f_0, t_1, f_1)$$

and generates samples of a linear swept-frequency cosine signal at the time instances defined in array t where t is time instance (secs), f_0 is the instantaneous frequency at time $t = 0$ (Hz) and f_1 is the instantaneous frequency at time t_1 (Hz). The chirp function can be set to run in three different modes - linear, quadratic and logarithmic as follows:

Linear: $f_i(t) = f_0 + \beta$ (instantaneous frequency sweep) where $\beta = (f - 1 - f_0)/t_1$ which ensures that the desired frequency breakpoint f_1 at time t_1 is maintained.

Quadratic: $f_i(t) = f_0 + \beta t^2$ where $\beta = (f_1 - f_0)/t_1^2$. If $f_0 \geq f_1$, the output waveform is a downsweep, with a default shape that is convex. If $f_0 < f_1$, the output waveform is an upsweep, with a default shape that is concave.

Logarithmic: $f_i(t) = f_0 + 10^{\beta t}$ where $\beta = (\frac{f_1}{f_0})^{\frac{1}{t_1}}$

In the DBX system developed for this thesis, the chirp is set to run in logarithmic mode using four parameters where each parameter can takes on a large value. It is therefore difficult to re-create the exact chirp. This is mainly because the chirp parameters vary over a wide range, giving, in effect, a large key space. It is interesting to note, that this result has been developed in nature. For example, dolphins send a series of chirps, or clicks through water [62]. When the sound waves interact with an object, they bounce back and the echoed sound enables the dolphin to have a mental picture of an object by comparing the echo with the sound it already knows. As with the human visual system, there are a huge range of that the brain of a dolphin can generate. The template space in this application being equivalent to the key space available in the current application! Some scientists believe that dolphins may actually see acoustic images with their brains. This method is termed as echolocation. It has recently been discovered that the chirp has many practical uses in technology. For example, in a paper at the (2004) Next Generation Communications Network Conference, Mohsen Kavehrad stated that [49] '... multirate laser pulses with wave forms shaped like dolphin-chirp sound pulses offer a new way of helping free-space optical signals penetrate

clouds, fog, and other adverse weather conditions that sometimes hamper the success of this method.' One of the principal contributions of this research thesis is that chirps can be used effectively to solve the key exchange problem and authentication issues in cryptography.

Dynamic Blocks. Unlike other block ciphers, where the block length is fixed, DBX employs dynamic blocks, where the blocks to be encrypted change at every cycle. This makes it harder to attack because if the block length is known, an attacker will typically work on a particular block length and if he/she manages a successful attack, he/she will have, by default, recognised the pattern. There is no pattern in DBX.

Dynamic Key. The encryption key always changes. Even if the same plaintext is used, the key will be different. Thus, a brute force attack will only help in decrypting one message. Further, *a priori* knowledge of the key will not help because of its dynamic nature. A number of cryptanalysis methods dealing with plaintext attack will fail for the same reason.

Huge Keyspace. The security of DBX does not only lie in the key. It is a combination of other elements from which the key is composed, thus creating a huge keyspace. The key consists of noise length, key space and total chirp length and it is the combination of these three components that provides the huge key space, making the task of breaking it using brute force effectively impossible. The current keyspace has been set with the following parameters.

Noise length varies from 100,000 to 150,000 bytes, so the difference is 50,000 bytes. The four chirp parameters mentioned in the previous section, t, f0, t1, and f1 take on different range of values.

noise length: this is varied between 100 KB and 150 KB, yielding a difference of 50 KB.

t: this is calculated as the ratio of total file length to key length. By taking the commonly used file lengths which vary between 100 KB and 10 MB (after padding with a 100 KB random noise), the difference becomes: $99 * 10^4$.

f0: the value of f0 goes to a maximum of 4.

t1: t1 has been tested to a maximum of 10^{18}.

f1: this value also has been tested to a maximum of 10^{18}.

Finally, as mentioned above, there are three methods the chirp function can be used.

Maximum key space obtained from the above is:

$$(50 * 10^3) * (4) * 10^{18} * 10^{18} * (3) = 6 * 10^{41}$$

$$2^{140} = 1.4 * 10^{42}$$

Therefore the key space can be considered as 140 bit key space. Of course, this is by no means the maximum keyspace for DBX, the above values may be increased by changing the current settings of the program thus increasing the key space, taking into account the matlab and memory space constraints.

Blum-Blum Shub. BBS is a well known cryptographically secure random number generator. To make it more secure, the method used here calls for reinitialising the generator after every few rounds, obtaining a new seed each time. This makes it more difficult to try and predict the already complex pattern of a BBS output stream.

Chapter 7

Conclusion and Future Directions

The principal focus of this research thesis has been three fold:

- To investigate the use of prime number modulation for enhancing the effectiveness of conventional encryption engines.
- To design a key exchange method by covertly watermarking the cipher text with the key that has been used to generate it.
- To combine the current key exchange mechanism with encryption using deterministic chaos.

DBX can be applied to work with deterministic chaos using a multi-algorithmic approach (Chapter 3) coupled with the key exchange method discussed in Chapter 5. When coupled together, it forms the basis for a new product called Crystic^{TM} which has been developed through a joint venture between Loughborough University (Department of Computer Science and the Department

of Electronic and Electrical Engineering) and Lexicon Data Limited and is marketed by Cryptic Limited. Details of this product including a technical report, business plan, system application and documentation are given in Appendix B.

7.1 Authentication

One of the principal themes of current research into encryption involves the facility to authenticate a decrypted message. The ability for a receiver to decrypt a message can provide a false sense of confidence that the plain text obtained is authentic. This provides any potential attacker with the means of disseminating miss-information. The authentication method developed for this thesis is based on the use of chirp coding bit streams. The bit streams are generated from the plain text by application of a transformation. In this thesis, the transformations research have been based on: (i) a simple additive transform; (ii) convolution of the input with a random noise field; (iii) application of a hash function transformation; (iv) application of a wavelet transform. The fourth method is very general as, in addition to using standard wavelets, for this application, any wavelet function can be constructed. The reason for this is that in conventional wavelet based signal analysis, wavelets are designed to be strictly orthogonal in order for data, having been processed in 'wavelet space', to be inverted back into real space. However, in the present application this is not only unnecessary but, is strictly, not desired as the whole point of encryption processes is to ensure that the transformations used are non-invertible, i.e. based on the employment of one-way functions.

The watermarking method developed for this thesis is unique in that it provides a method for self-authentication. All current watermarking techniques require access to the original (non-watermarked) data in order to clarify the existence (or otherwise) of the watermark and the information it conveys. The use of chirp coding eliminates the need to have access to the original file and in so doing, eradicates the requirement of generating and managing a data base for verification and associated protocols. No other function except for a chirp provides the facility for self-authentication. The reason for this lies in the properties of the correlation of a chirp with itself, namely, that, for chirping parameter α,

$$\int_{-\infty}^{\infty} \exp(-i\alpha t^2) \exp[i\alpha(t+\tau)^2] dt = \delta(\tau)$$

This property is the reason for chirp coding being used in a range of man made (e.g. real and synthetic aperture radar, telecommunications, synthetic aperture radio wave imaging etc.) and natural communications system (e.g. active and passive communications by whales, dolphins and bats). This thesis provides the first example of its use for watermarking which can, in principle, be applied to any data including, audio and video.

7.2 Key Exchange

The specific use of chirp coding for watermarking digital signals depends upon the application. For audio and video signals the perturbation of the input data by the watermark is so insignificant that any distortion caused by its presence is not noticeable. Hence the watermark can remain covertly embedded in the data providing a digital seal or signature that can be read as

appropriate. However, for data that must remain bit perfect, application of a chirp coded watermark requires that the watermark is deleted from the data once it has been recovered. This is the basis for the key exchange method proposed in this thesis. In this case, a plain text dependent key (bit stream) is used to generate the ciphertext via application of a given encryption engine. The same bit stream is then transformed to the corresponding chirp stream which is then used to watermark the cipher stream. Upon reception, the bit stream is recovered and used to remove the watermark by regenerating the chirp stream. The same bit stream is then used to decrypt the cipher stream. Since the key is plaintext dependent, the process represents a practical solution to the problem of implementing a one-time pad, even though what is represented here is not a one-time pad, but this is the direction. Note that in order to 'cover' for the case when the same plaintext is used twice, the data is automatically padded with random bits to ensure that a different key is generated each time the system is executed.

7.3 Encryption using Deterministic Chaos

The large majority of conventional encryption engines are partially or entirely dependent on the use of prime numbers. Symmetric encryption algorithms depend on prime numbers that are used as input parameters together with a (non-prime) key to initiate the algorithm; typically a large integer whose size defines the length of the key (e.g. 64-bit and 128-bit keys). Key exchange algorithms are then required to exchange the key between sender and recipient prior to execution of the encryption engine. However, asymmetric encryption systems, which are based almost exclusively in the RSA algorithm, depend entirely on prime numbers (see section 3.7). Attempting to

break the system entails factoring huge numbers, even though this is theoretically possible, there is always a time constraint (a number of years in some cases). A shortcut method to bypass factoring may be out there, which will definitely change the whole outlook on prime numbers in cryptography.

A well known and computationally efficient method of testing for a prime is the Miller test, i.e.

If the extended Reimann hypothesis is true,

then if p is a SPRP (Strong Probable Prime Base)

for all integers n with $1 < n < 2(logp)2$, *then p is prime.*

Miller test has been known for some time and employed in a number of RSA attacks. Until relatively recently, Miller's test has been used in the knowledge that is it, in effect, a 'formula without a proof' and therefore doubt has remained as to its routine application. However, in October 2004, a proof of the extended Reimann hypothesis was published by Louis de Branges de Bourcia [27], a prominent Professor of Mathematics at Purdue University who has made finding the proof of the Reimann hypothesis his life's work. This has very serious potential implications for conventional RSA encryption and prime number based encryption systems in general. If this proof is confirmed and turns out to be correct, it will provide a short-cut to identifying primes without large lookup tables and thus significantly increase the vulnerability of prime number based encryption systems and the algorithms upon which they are based.

Although encryption using deterministic chaos has been investigated for some years, no commercial system has to date been implemented other than that being marketed by Crypstic Limited (see Appendix B). The reason for this

is that conventional prime number based pseudo random number generators have been considered secure enough for most applications and that a serious threat must exist in order to change the procedures and protocol associated with an entire secure communications network. Proof of the extended Riemann hypothesis means that Miller's test for evaluating whether a number is prime or otherwise can now be used routinely and thus, seriously undermines the basis for the majority of encryptions systems in use today.

Chaos based encryption does not make use of primes which is a principal characteristics. Moreover, unlike conventional pseudo random number generators, pseudo chaotic number generators have an unlimited number of forms, i.e. there are an unlimited number of iteration function sequences that can be designed (see section 3.17). Although the iterated function sequences must be tailored to provide a statistically uniform output, the fact that there are so many that can be used, means that a specific encryption engine can be designed that is unique to a given sender/receiver. The current platform for implementing this approach is based on utilizing a pair of USB memory sticks as discussed in Appendix B.

7.4 Discussion

There are three golden rules of Cryptology:

- no cryptographic system is invulnerable to attack;
- no cryptographic system is invulnerable to attack;
- no cryptographic system is invulnerable to attack.

Cryptology is a field of study that, like any other, has a legacy of attempts to derive the ultimate (theoretical) solution which has failed in practice. One of the principal mistakes that is made is to believe that a single solution is possible, e.g. application of a specific algorithm, which can be applied over a relative long period of time without any significant changes. This is the principal of mono-staticism and is one of the main reasons for the continued failure of information security. The failure of the Enigma cipher systems used in Germany from the late 1930s to the end of the second world war remains one of the best examples of mono-staticism. Nearly 10,000 of these encryption machines were manufactured between 1938 and 1945, all with the same specification and operating procedures (although a fourth rotor was introduced for use by the U-boat fleet in 1943, which caused a 'black' period for the allied side until a four rotor machine was captured together with the code books without the knowledge of U-boat high command!).

A new underlying philosophy is now beginning to emerge that is based on the principal of developing multiple solutions which constantly changed - the principal of multi-dynamicism. This approach involves developing algorithms, systems and applications together with the procedures and protocol used in practice which change continuously. This includes the principal of dynamic algorithm management in which encryption engines are constructed based on multi-algorithmicity or 'paracryption' like the one presented in Section 3.17. This thesis has also investigated a method of key exchange that is also dynamic and dependent on the plain text together with a transformation that can in principle, be chosen from an unlimited list, especially through application of (non-orthogonal) wavelet functions. Also, the use of self-authentication via chirp coding, negates the need for a database (dynamic or otherwise). In this sense, the entire approach for this thesis is

based on a multi-dynamic paradigm.

The watermarking method developed for this thesis has a range of applications in addition to key exchange for which it was originally conceived. It should be appreciated that watermarking information has some advantages over encrypting it. First and foremost is the fact that any encrypted data will immediately alert an interceptor to the fact that there may be valuable information worth intercepted and attacking. On the other hand, a covert watermark is interceptor illusive because the data appears to be of an original plain text form. This issue points to an approach in which the watermark is the cipher text which is then transmitted within the body of data that is unaffected by the presence of the watermark, e.g. audiovisual data (e.g. wav and/or avi files).

7.5 Future Directions

There are a number of directions that are possible to undertake based on the work herein. However, many of these will inevitably be 'driven' by the demands and constraints associated with a given system and its specific application. One such applications specific product is Crypstic - see Appendix B. In this case, a unique multi-algorithmic encryption engine (unique to two sticks and only two sticks) is provided in a private area of the USB memory stick that is password protected.

7.5.1 Covert Access

The fact that access to the Crypstic system is based on a pop up GUI which requires a user defined password means that the system (not the algorithm) potentially vulnerable to (password) attack. In order to overcome this problem it should be possible to design an approach in which the hidden area remains entirely covert. Access to this area of memory can then be achieved by modification of one of a number of different files stored in the public area. Modification can, for example, be undertaken by renaming a specific file (from a field of size N to a field of size M where $M < N$) with use of just the mouse and the delete key in order to overcome any potential attack based on the use of key loggers (which to date, do not record mouse movements).

7.5.2 Copy Protection

Assuming that an attack has been successful in terms of obtaining access to the encryption engine on a Crypstic, the executable file can then be copied from the Crypstic to another platform. In order to prevent use of the engine on another platform, a small component of the process can be reserved for the CPU (with a unique serial number) on the USB memory stick itself. This prevents execution of the engine on any other platform should any unauthorised user gain access to the executable file.

7.5.3 Dynamic Key Exchange

The current Crypstic is based on a set of keys that are 'hardwired' into the final encryption engine, that, along with the algorithms themselves, are

unique to one pair of sticks and only one pair. The overall key becomes the physical USB memory stick itself and is typically held by the user on a key ring along with the rest of his/her keys. If the USB memory stick is lost by either the sender or receiver (or both), then both sender and receiver are required to obtain a new set of sticks. The protocol is, in effect, the same as if a user loses a conventional key for which there is no replica. However, in the case of Crypstic, both a new lock and and a new key must be acquired. Here, the key is a based on issuing a new Crystic and the lock is analogous to the generation of a new and unique encryption engine that, in turn, is based on a unique sequence of pseudo chaotic number generators.

In this thesis, the key exchange method has been researched using conventional ciphers (e.g. the Blum Blum Shub algorithm) modified to incorporate multi-dynamicism using prime number modulation. The key exchange method is independent of the encryption engine that is used and can, thus, be used effective with any symmetric cipher including those based on the application of single- and/or multi-algorithmic ciphers that use deterministic chaos.

7.5.4 Plain Text Image Based Encryption

Watermarking is usually considered to be a method in which the watermark is embedded into a host image in an unobtrusive way. Another approach is to consider the host image to be a data field that, when processed with another data field, generates new information.

Consider two images i_1 and i_2. Suppose we construct the following function

$$n = \hat{F}_2 \left(\frac{I_1}{\mid I_1 \mid^2} I_2 \right)$$

where $I_1 = \hat{F}_2[i_1]$, $I_2 = \hat{F}_2[i_2]$ and $\hat{F}_2$ denotes the two-dimensional Fourier transform operator. If we now correlate n with i_1, then from the correlation theorem

$$i_1 \odot \odot n \Longleftrightarrow I_1^* \frac{I_1}{\mid I_1 \mid^2} I_2 \Longleftrightarrow i_2.$$

In other words, we can recover i_2 from i_1 with a knowledge of n. Because this process is based on convolution and correlation alone, it is compatible and robust to printing and scanning, i.e. incoherent optical imaging [56]. An example of this is given in Figure 8.1. In this scheme, the noise field n is the private key required to reconstruct the watermark and the host image can be considered to be a public key.

Now, one of the principal components associated with the development of methods and algorithms to 'break' cipher text is the analysis of the output generated by an attempted decrypt and its evaluation in terms of an expected type. The output type is normally assumed to be plain text, i.e. the output is assumed to be in the form of characters, words and phrases associated with a natural language such as English or German, for example. If a plain text document is converted into an image file then the method described above can be used to diffuse the plain text image i_2 using any other image i_1 to produce the field n. If both i_1 and n are then encrypted, any attack on these data will not be able to make use of an 'analysis cycle' which is based on the assumption that the decrypted output is plain text. This approach provides the user with a relatively simple method of 'confusing' the cryptanalyst and invalidates attack strategies that have been designed and developed on the assumption that the encrypted data have been derived from plain text alone.

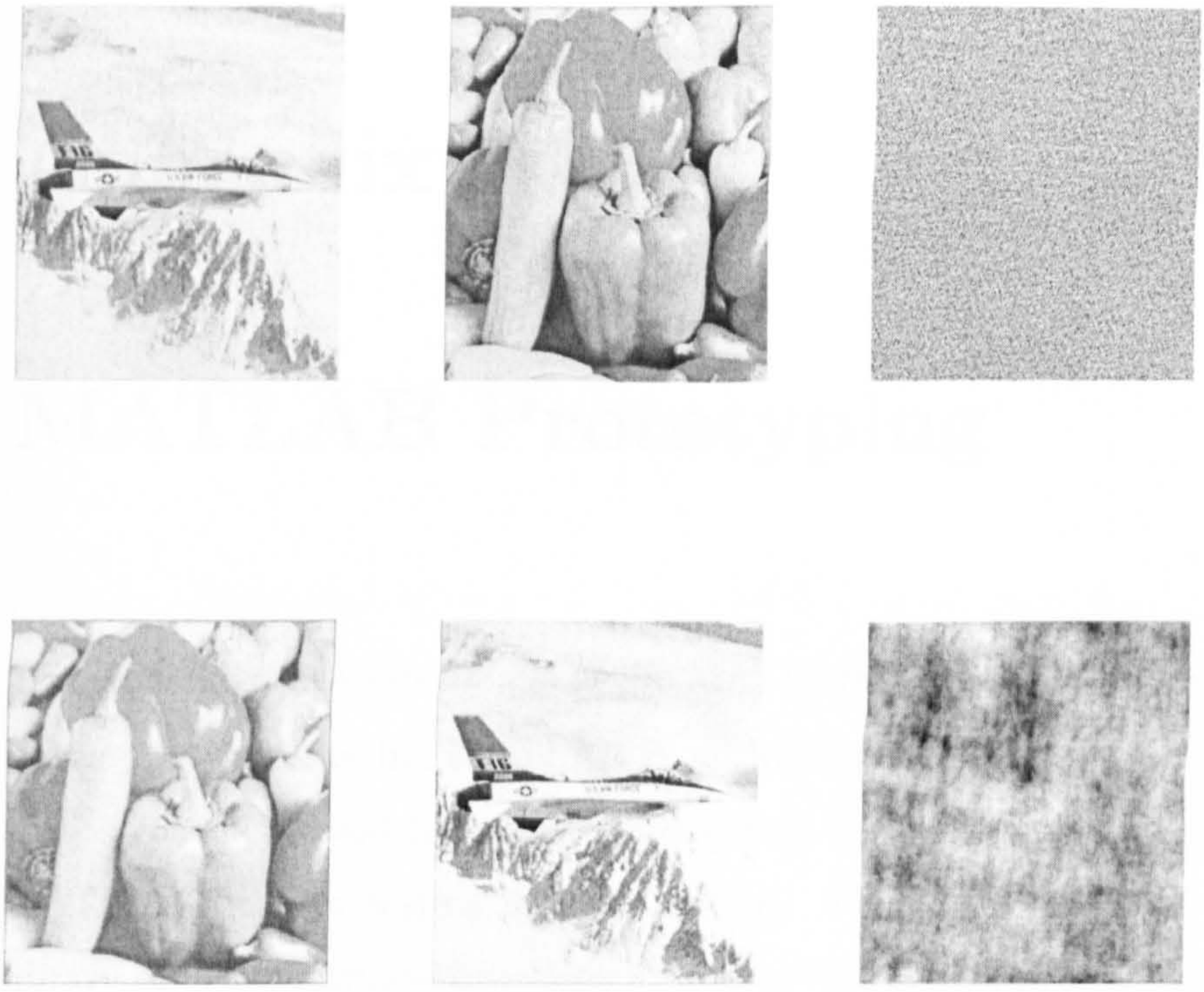


Figure 7.1: Example of a covert image watermarking scheme. i_1 (top-left) is convolved (with pre-processing) with i_2 (top-middle) to produce the noise field (top-right). i_2 is then printed and scanned at 300 dpi and then re-sampled back to its original size (bottom-left). Correlating this image with the noise field generates the reconstruction (bottom-centre). The reconstruction depends on just the host image and noise field. If the noise field and/or the host image are different or corrupted, then a reconstruction is not achieved (bottom-right).

Appendix A

MATLAB Prototyping

This appendix provides details of the MATLAB functions developed for this thesis. MathWorks Inc MATLAB is an ideal platform for numerical work and is routinely used for rapid prototyping, i.e. the rapid development of MATLAB code for testing new algorithms. This includes use of the large library of based intrinsic functions offered by MATLAB and the increasingly wide range of specialist toolboxes offered by the system.

The MATLAB functions (full MATLAB code) are given in the accompanying CD and the back of this thesis.

A.1 Fixed Version Mode

This version is called a fixed version mode. The parameters to be transmitted are one time. The chirp length is recalculated by the recipient modules according to the file length and the key length. The key length is fixed at 80-bits. The length of the noise is also fixed at 99983 bytes. The noise length

and chirp parameters t, f0, and f1 can be varied by the sender at any time.

The sender's program is launched by running 'dbxfix_encrypt.m'. The receiver mode is'dbxfix_decrypt.m'.

The following programs are then executed sequentially or depending on the selection:

dbxfix_encrypt.m

Main program which calls all other functions necessary for encryption.

dbxfix_encrypt.m

Main program used by the recipient which calls all functions needed for decryption.

readparams_ver5.m

This program prompts the sender to either use existing parameters or enter new ones. Four parameters can be changed. Padding length, plus three chirp parameters: f0, t1, and f1. Details of these parameters can be found in the documentation. Once the user enters new parameters, the program displays a list of the old and new parameters.

key_gen_fixedvera.m

Reads in a file, introduces noise, and generates a key. The noise is used as a file header, it is initially fixed at 99983 bytes. This value may be changed by the user before key generation starts. The key is generated by adding up all characters in a file to be encrypted after padding has taken place. The padded file is randomly created.

key_gen_conv_fixedver4.m

This module uses convolution integral to generate a key. It works on the same principle as the previous module, the main difference is that after generating noise and padding to a file, another set of random noise is generated and convoluted to the new file. Part of the resultant output is taken as the key.

key_gen_fixedwavelet6.m

This module uses Deaubechies wavelet to create a key. Takes in a file, generates noise and pads it in front of the file. Wavelet decomposition with Deaubechies wavelets is then applied. Approximation and detailed coefficients are then extracted. Various methods are used to generate an output binary stream used as a key.

key_gen_hash_fixedvera.m

This module uses SHA-256 to generate a key. It reads in a file, creates random noise and pads to the original file. It then gives a hash value of the total file.

encrypt_vera.m

This file will take any type of file, and encrypt it using the key generated from the previous module. It takes the binary key , divides into a number of parts, and uses each part as a seed for BBS generator each time it is initialised. The block length changes for each new cycle, it is currently set to a minimum of 5 and maximum of 50 characters.

wmark_insert_ver4.m

This module takes in the key and the encrypted file. It then converts the binary key into a stream of 1s and -1s. It calculates the chirp length which is

a ratio of the file length to the key length. Each binary stream is multiplied to a chirp, concatenated and finally added to the encrypted data.

wmark_remove_fixedver8.m

This file extracts the key from the encrypted text. It first recreates the chirp. Then applies correlation function, correlates the chirp with the total signal. This extracts the key from the encrypted data. Finally the key is subtracted leaving only encrypted data.

decrypt_fixedver9.m

Since this is a symmetric cipher, this file will decrypt a file into plaintext, by reversing the encryption process. The noise finally gets separated from the actual file.

parameters.txt

Data file for storing chirp and noise parameters. This file is used by *read_paramsver5.m*

The following three files have been successfully tested with the program.

sunset.jpg

test100k.txt

taiwan.txt

A.2 Variable Parameters Mode

In this version, noise length changes with each file to be encrypted, hence the total file length changes. After encryption the encrypted file is sent through

open channel while parameters are sent through secure channels. Parameters need to be transmitted for each encrypted file. This mode is quite similar to the above, the main difference is the parameters to be exchanged.

All of these programs appear on the previous section, so there will be no need for repeat description here.

dbx_encrypt.m

Main encryption program.

dbx_encrypt.m

main decryption program.

key_gen_verc.m

Key generation using summation method.

key_gen_conv_ver3.m

Key generation using convolution integral.

key_gen_wavelet4.m

Key generation using Daubechies wavelets.

key_gen_hash_ver9.m

Key generation using hash function SHA-256.

encrypt_vera.m

Encryption module.

wmark_insert_ver5

Watermark exchange module.

saveparamsa.m

This code is used to save parameters: chirp length and noise length.

loadparams7.m

Loads parameters before decryption process starts.

wmark_remove_ver9.m

Extracts watermark.

decrypt_vera.m

Decrypts.

Appendix B

Crypstic

Crystic is marketed by Crypstic Limited, Mayfair House, 14-18 Heddon Street, Mayfair, London W1B 4DA. It is based on the use of multi-algorithmic deterministic chaos coupled with key exchange using chirp coding to design an encryption engine that unique to a pair of Crystics (i.e. USB memory sticks).

Deatils on the Crypstic system are given on the CD that accompanies this thesis. This includes:

- A full technical report entitled *Digital Cryptography using Deterministic Chaos* that gives a background to thc technical specifications of the system including an information theoretic approach to the use of deterministic chaos for encryption.

- A comprehensive business plan that has been used to generate investment for the development of the product and the establishment of Crystic Limited.

- Documentation concerning the system including a user guide and an example applications program - 'Crypstic'.

- Example publications.

Bibliography

[1] Peter Alfred. Eratosthenes of cyrene. 1998.
http://www.math.utah.edu/ alfeld/Eratosthenes.html.

[2] Ross Anderson. Why Cryptosystems Fail. In *ACM, 1st Conference in Computer and Communication Security*, Virginia, USA, 1993.
http://www.cl.cam.ac.uk/users/rja14/wcf.html.

[3] S. Ares and M. Castro. Hidden structure in the randomness of the prime number sequence? *Physica A: Statistical Mechanics and its Applications*, 2005.
`http://www.sciencedirect.com...sdasrticle.pdf`.

[4] Articsoft Technologies. Introduction to encryption. 2005.
http://www.articsoft.com/wp_explaining_encryption.htm.

[5] David Barton. Encryption. 2004.
http://www.filetopia.org/encryption.htm.

[6] Hal Berghel. Protecting ownership through digital watermarking. 1996.
http://portal.acm.org/citation.cfm?id=620504.

[7] Eli Biham and Adi Shamir. Differential Cryptanalysis of Feal and N-Hash. In *Lecture Notes in Computer Science 547, Advances in Cryptol-*

ogy - EUROCRYPT'91. Springer-Verlag, 1991.
http://members.aol.com/jpeschel/algoritak.htm.

[8] J. Blackledge, A. Evans, and M. Turner, editors. *Mathematical Methods, Algorithms and Applications*, volume 189 - 222. Howard Publishing Series, 2002.

[9] Jonathan M. Blackledge. *Digital Signal Processing.* Horwood, 2003.

[10] L. Blum, M. Blum, and S. Shub. A Simple Unpredictable Random Number Generator. *SIAM Journal of Computing*, 15, 1986.
http://locus.siam.org/SICOMP/volume-15/art0215025.html.

[11] Dan Boneh. Twenty Years of Attacks on the RSA Cryptosystem. *American Mathematical Society*, 46(2):203 – 213, 1999.
http://crypto.stanford.edu/ dabo/papers/RSA-survey.pdf.

[12] Ernest Brickell, Dorothy E. Denning, Stephen T. Kent, David P. Maher, and Walter Tuchman. SKIPjack Review, Interim Report, The SKIPjack Algorithm. 1993.
`http://www.epic.org/crypto/clipper/skipjack_interim_review.html`.

[13] Lawrie Brown. Block Ciphers - Modern Private Key Ciphers (Part 2). 1996.
http://williamstallings.com/Extras/Security-Notes/lectures/blockB.html.

[14] Lawrie Brown. Block Ciphers - Modern Private Key Ciphers (Part I). 1996.
http://williamstallings.com/Extras/Security-Notes/lectures/blockA.html.

[15] James Buchmann. *Introduction to Cryptography.* Springer, 2001.

[16] Chris Caldwell. The Largest Known Prime by Year. 2005.
http://www.utm.edu/research/primes/notes/by_year.html.

[17] Chris Caldwell. The Prime Glossary, Wieferich Prime. 2005.
http://primes.utm.edu/primes/search.php?Comment=SophieNumber=100.

[18] Chris Caldwell. The Prime Glossary, Wieferich Prime. 2005.
http://primes.utm.edu/glossary/page.php?sort=WieferichPrime.

[19] California Intitute of Technology. The Geiger Counter and counting Statistics. 1997.
http://www.kronjaeger.com/hv-old/radio/geiger/caltech/exp2.htm.

[20] Keith Calkins. Biographies of Mathematicians - Goldbach. 1999.
http://www.andrews.edu/ calkins/math/biograph/biogoldb.htm.

[21] C. Charnes, L. O'Connor, J. Pieprzyk, R. Safavi-Naini, and Y. Zheng. Further Comments on the Soviet Encyption Algorithm. 1994.
`http://kremlinencrypt.com/algorithms.htm#GOST`.

[22] Clements Library. Spy Letters of the American Revolution, Secret Methods and Techniques - Invisible Ink. 1999.
http://www.si.umich.edu/spies/methods-ink.html.

[23] Counterpane Internet Security. Security Alert: Microsoft RPC DCOM Worm. 2003.
http://www.counterpane.com/alert-v20030811-001.html.

[24] D I Management Services. Cryptography Code. 2005.
http://www.di-mgt.com.au/crypto.htmlTopOfPage.

[25] Joan Daemen. Cipher and Hash Function Design, Strategies based on lincar and differcntial cryptanalysis, 1995.

[26] H. Davenport. *The Higher Arithmetic.* Cambridge University Press, 2002.

[27] Louis de Branges. Apology for the Proof of the Riemman Hypothesis. 2005.
http://www.math.purdue.edu/ branges/apology.pdf.

[28] Department of the Army. Basic Cryptanalysis. 1990.
`http://www.umich.edu/~umich/fm-34-40-2/#ps>Postscript</`.

[29] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transaqctions on Inofrmation Theory*, 1976.
http://crypto.csail.mit.edu/classes/6.857/papers/diffie-hellman.pdf.

[30] Echelon Watch Organisation. Echelon. 2000.
`http://www.echelonwatch.org`
`http://www.a861.com/info/echelon-watch.html`.

[31] Carl Ellison and Bruce Shneier. Ten Risks of PKI: What Youre not Being Told about Public Key Infrastructure. *Computer Security Journal*, XVI(1), 2000.
http://www.schneier.com/paper-pki.pdf.

[32] Euclid. Euclid's Elements. 300 BC. Compiled (1998) by D. E. Joyce, Department of mathematics and Computer Science, Clark University, USA.
http://aleph0.clarku.edu/ djoyce/java/elements/bookIX/bookIX.html.

[33] European Space Agency. Beagle 2 lander. 2003.
http://www.esa.int/SPECIALS/Mars_Express/SEMPM75V9ED_0.html.

[34] Elizabeth Ferrill and Matthew Moyer. A Survey of Digital Watermarking. 1999.
http://elizabeth.ferrill.com/papers/watermarking.pdf.

[35] Paul Garrett. *Making, Breaking Codes.* Prentice Hall, 2001.

[36] Oded Goldreich. *Foundations of Cryptography.* Cambridge University Press, 2001.

[37] Timothy Gowers. *Mathematics, A Very Short Introduction.* Oxford University Press, 2002.

[38] Niel Hahnfield. Cryptography Tutorial: RSA. 2001.
http://www.antilles.k12.vi.us/math/cryptotut/rsa1.htm.

[39] Feng Hao, Ross Anderson, and John Daugman. Combining cryptography with biometrics effectively. 2005.
http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-640.pdf.

[40] Harvey Heinz. Patterns in Primes. 2000.
http://www.geocities.com/ harveyh/primes.htm.

[41] John Hershey. *Cryptography Demystified.* McGraw-Hill, 2003.

[42] Paul Hoffman. *The Man who Loved only Numbers.* Fourth Estate, 1998.

[43] Institute of Paper Science and Technology at Georgia Tech. Watermarks. 2004.
http://www.ipst.gatech.edu/amp/education/watermark/watermarks.htm.

[44] Kenneth Ireland and Michael Rosen. *A Classical Introduction to Modern Number Theory.* Springer-Verlag, 1993.

[45] Y. Saouter J-M. Deshouillers, H.J.J. te Riele. New Experimental Results Concerning the Goldbach Conjecture. *Centrum voor Wiskunde en Informatica*, MAS-R9804, 1998. ISSN 1386-3703 http://ftp.cwi.nl/CWIreports/MAS/MAS-R9804.pdf.

[46] Neil Johnson. Steganography. 2000. http://www.jjtc.com/stegdoc/bks315.html.

[47] Pascal Junod. Cryptographic Secure Pseudo-Random Bits Generation : The Blum-Blum-Shub Generator. 1999. http://crypto.junod.info/bbs.pdf.

[48] Pascal Junod. Six ways to break DES. 1999. http://lasecwww.epfl.ch/memo_des.shtml.

[49] Mohsen Kavehrad. Conference review: Optics east features nanotechnology and itcom. 2004. http://cictr.ee.psu.edu.

[50] Gary Kessler. Overview of cryptography. 1998. http://www.garykessler.net/library/crypto.htmlhash.

[51] Erica Klarreich. Take a Chance: Scientists put Ramdomness to work. *Science News*, 166(23):362, 2004. http://www.sciencenews.org/articles/20041204/bob9.asp.

[52] Evangelos Kranakis. *Primality and cryptography.* Wiley, 1986.

[53] kremlinencrypt.com. Kremlin Cryptographic Algorithms. 2005. `http://kremlinencrypt.com/algorithms.htm#GOST`.

[54] Jeordan Legon. Teenager arrested in 'Blaster' Internet attack. 2003. http://www.cnn.com/2003/TECH/internet/08/29/worm.arrest/index.html.

[55] Scott Lewis and Todd Steigerwalt. Biometric Encryption. http://www.emory.edu/BUSINESS/et/biometric/Index.htm.

[56] Khaled Mahmoud. *Digital Watermarking for Low Resolution Print Security.* PhD thesis, Loughborough University, 2005.

[57] Walt Mankowski. The Great Internet Mersenne Prime Search. 2005. http://www.mersenne.org/prime.htm.

[58] Itsik Mantin. Analysis of the Stream Cipher RC4. Master's thesis, The Weizmann Institute of Science, 2001.

[59] Media Crypt. Swiss Encryption Technology. 2005. http://www.mediacrypt.com/.

[60] Julie Meloni. Encryption Tutorial. http://webmonkey.wired.com/webmonkey/programming/php/tutorials/tutorial1.html.

[61] Alfred Menezes, Paul Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography.* CRC Press, 2001.

[62] Barbara S. Moses. Dolphins - the ride. 2001. http://www.detroitzoo.org/was/dolphins_resource.pdf.

[63] National Institute of Standards Technology. NIST Brief Comments on Recent Cryptanalytic Attacks on Secure Hashing Functions and the Continued Security Provided by SHA-1. 2004. http://csrc.nist.gov/hash_standards_comments.pdf.

[64] Yves Nievergelt. *Wavelets Made Easy.* Birkhauser, 2001.

[65] J. O'Connor and E. Robertson. Prime Numbers - Some Unsolved Problems. 2005.
http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/Prime_numbers.html.

[66] Ivars Peterson. Prime pursuit: Constructing an effecient prime number detector. *Science News Online*, 162, 2002.
http://www.sciencenews.org/articles/20021026/bob9.asp.

[67] Ivars Peterson. Closing the Gap on Twin Primes. *Science News*, 168(3), 2005.
http://www.sciencenews.org/articles/20050716/mathtrek.asp.

[68] Ivars Peterson. Goldbach's Prime Pairs. 2005.
http://mathforum.org/library/view/17035.html.

[69] Nikolai Ptitsyn. *Encryption using Deterministic Chaos.* PhD thesis, De Montfort University, 2002.

[70] Brian Raiter. Prime number hide-and-seek: How the rsa cipher works. 2003.
http://www.muppetlabs.com/ breadbox/txt/rsa.html.

[71] Arnold G. Reinhold. Diceware Passphrase. 1995.
http://world.std.com/ reinhold/diceware.html.

[72] Paul Reynolds. Breaking codes: An impossible task? 2004.
http://news.bbc.co.uk/1/hi/technology/3804895.stm.

[73] M. J. Robshaw. Stream Ciphers. 1995.
ftp://ftp.rsasecurity.com/pub/pdfs/tr701.pdf.

[74] Phillip Rogaway and Dan Coppersmith. A Software-Optimized Encryption Algorithm. *Journal of Cryptology*, 11(4):273 – 287, 1998.

http://www.springerlink.com/media/G3QMUNWGUTLB85K3JMFK/...
Contributions/2/Y/D/Q/2YDQL7VUJ2E48WP8.pdf.

[75] Kenneth Rosen. *Elementary Number Theory.* Addison Wesley, 2000.

[76] RSA Laboratories. RSA Laboratories' Frequently Asked Questions About Today's Cryptography. 2000.
http://www.rsasecurity.com/rsalabs/node.asp?id=2170.

[77] RSA Laboratories. RSA Laboratories' Frequently Asked Questions About Today's Cryptography. 2000.
http://www.rsasecurity.com/rsalabs/node.asp?id=2171.

[78] RSA Laboratories. RSA Laboratories' Frequently Asked Questions About Today's Cryptography. 2000.
http://www.rsasecurity.com/rsalabs/node.asp?id=2172.

[79] RSA Laboratories. RSA Laboratories' Frequently Asked Questions About Today's Cryptography. 2000.
http://www.rsasecurity.com/rsalabs/node.asp?id=2173.

[80] RSA Security. 200 000 reward for factoring offered by RSA. *Computer Fraud and Security*, 2001(8):4, 2001.

[81] Arto Salomaa. *Public-Key Cryptography.* Springer, 1996.

[82] Security Section, Research and Development Center. Fsango. 2000.
http://www.fsi.co.jp/Cipher-HP_e/Overview.

[83] Bruce Shneier. *Applied Cryptography.* Wiley, 1996.

[84] Bruce Shneier. *Applied Cryptography.* Wiley, 1996.

[85] Bruce Shneier. Self-Study Course in Block Cipher Cryptanalysis. 2000. http://www.schneier.com/paper-self-study.html.

[86] Bruce Shneier. Blowfish. 2005. http://www.schneier.com/blowfish.html.

[87] Simon Singh. *The Code Book.* Anchor Books/Doubleday, 2000.

[88] SSH Security Communications. Random Number Generators. 2005. http://www.ssh.com/support/cryptography/algorithms/random.html.

[89] William Stallings. *Cryptography and Network Security.* Prentice Hall, 2003.

[90] Tripwire Inc. Change Auditing Solutions. 2005. http://www.tripwire.com.

[91] Eric Uner. Generating Random Numbers. 2004. http://www.embedded.com/showArticle.jhtml?articleID=20900500.

[92] Cheryl Vroom and Rossouw Solms. A Practical Approach to Information Security Awareness in the Organisation. In M. Ghonaimy, editor, *Security in the Information Society.* Kluwer Academic Publishers, 2002.

[93] John Walker. HotBits: Genuine random numbers, generated by radioactive decay. 1998. http://www.fourmilab.ch/hotbits/.

[94] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. 2004. http://eprint.iacr.org/2004/199.pdf.

[95] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient Collision Search Attacks on SHA-0. 2004.
http://www.infosec.sdu.edu.cn/paper/sha0-crypto-author-new.pdf.

[96] Peter Wayner. *Disappearing Cryptography.* Morgan Kaufmann, 1996.

[97] Eric W. Weisstein. Hash function. 2005.
http://mathworld.wolfram.com/HashFunction.html.

[98] Eric W. Weisstein. Prime Numbers. 2005.
http://mathworld.wolfram.com/topics/PrimeNumbers.html.

THESIS CONTAINS CD ROM

2 CDs.

BACK of Front of BOOK