

https://dspace.org/

A survey of the mathematics of cryptology

Gebbie, Stewart

2009-03-03T14:28:21Z
http://hdl.handle.net/10539/6608

Downloaded from a DSpace repository. https://dspace.org/

A Survey of the Mathematics of Cryptology

A research report submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in partial fulfilment of the requirements for the degree of Master of Science.

Stewart Gebbie

Johannesburg, February 3, 2002

Declaration

I hereby declare that this research report is my own, unaided work. It is being submitted for the Degree of Master of Science of Mathematics in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

_____

Stewart Gebbie

_____ day of _____ 20_____

Abstract

Herein I cover the basics of cryptology and the mathematical techniques used in the field. Aside from an overview of cryptology the text provides an in-depth look at block cipher algorithms and the techniques of cryptanalysis applied to block ciphers. The text also includes details of knapsack cryptosystems and pseudo-random number generators.

Acknowledgements

I would like to thank my supervisors: Prof. A.Knopfmacher and Prof. H.Prodinger.

Contents

Contents i

List of Figures iv

List of Tables v

1 Introduction 1
- 1.1 History . . . 1
- 1.2 Mathematics and Cryptology . . . 4
- 1.3 Outline . . . 4

2 Cryptographic Basics 6
- 2.1 Terminology . . . 6
- 2.2 Cryptographic Algorithms . . . 7
- 2.3 Simple Ciphers . . . 7
  - 2.3.1 Shift Cipher . . . 8
  - 2.3.2 Affine Cipher . . . 9
  - 2.3.3 Substitution Cipher . . . 11
  - 2.3.4 Vigenere Cipher . . . 11
  - 2.3.5 Hill Cipher . . . 12
- 2.4 Conclusion . . . 13

3 Introduction to Cryptanalysis 15
- 3.1 Modes . . . 15
- 3.2 Cryptanalytic Methods . . . 16
  - 3.2.1 Exhaustive Search . . . 16
  - 3.2.2 Letter Frequency . . . 16
  - 3.2.3 Kasiski Test . . . 17
  - 3.2.4 Index of Coincidence . . . 17
  - 3.2.5 Mutual Index of Coincidence . . . 19
  - 3.2.6 Generalisations . . . 20
- 3.3 Conclusion . . . 21

4 Block Ciphers and Cryptanalysis 22
- 4.1 Overview . . . 22
  - 4.1.1 Modes of Use . . . 22
  - 4.1.2 Block Cipher Algorithms . . . 24
- 4.2 DES Algorithm . . . 25

4.2.1 DES Round Function . . . 27
4.2.2 DES S-Boxes . . . 29
4.2.3 DES Key Schedule . . . 31
4.2.4 DES Properties . . . 33
4.3 Differential Cryptanalysis . . . 35
4.3.1 Overview . . . 35
4.3.2 Application to DES . . . 35
4.4 Linear Cryptanalysis . . . 45

5 Knapsack Cryptosystems 47
5.1 Overview . . . 47
5.2 Subset-Sum Problem . . . 47
5.3 Knapsack Cryptosystem . . . 48
5.4 Cryptanalysis . . . 50
5.4.1 Brute Force Attacks . . . 50
5.4.2 Bit Leaking . . . 51
5.4.3 Breaking the Basic Merkle-Hellman System . . . 51
5.4.4 Solving Low Density Knapsacks . . . 52
5.5 Conclusion . . . 56

6 Pseudo-Random Number Generators 58
6.1 Overview . . . 58
6.2 Definitions of Randomness . . . 58
6.2.1 Sources of Real Random Data . . . 59
6.2.2 Testing Data For Randomness . . . 60
6.2.3 Pseudo-Randomness . . . 61
6.3 Pseudo-Random Number Generating Algorithms . . . 63
6.3.1 Linear Feedback Shift Registers . . . 63
6.3.2 Linear Congruential Generators . . . 64
6.3.3 Number Theoretic Generators . . . 65
6.3.4 Combining PRNGs . . . 66
6.3.5 Blum-Blum-Shub Generator . . . 67
6.4 Conclusion . . . 70

7 Odds and Ends 72
7.1 Hash Functions . . . 72
7.1.1 The Birthday Attack . . . 73
7.1.2 Extending Hash Functions . . . 74
7.1.3 Cryptographic Hashing Algorithms . . . 75
7.2 Digital Signatures . . . 77
7.2.1 Digital Signature Standard . . . 78
7.2.2 Digital Signatures with Time Stamps . . . 80
7.3 Important Number Theoretic Problems . . . 81
7.3.1 Discrete Logarithm Problem . . . 81
7.3.2 Factorisation Problem . . . 81
7.4 Key Distribution and Key Agreement . . . 81
7.4.1 Diffie-Hellman Key Exchange . . . 82
7.5 Steganography . . . 83

7.5.1 Subliminal Channels . . . 84
7.6 Secret Sharing . . . 85
7.6.1 Threshold Schemes . . . 85
7.7 Time–Memory Trade-offs . . . 86
7.7.1 Application to Cellular Phones . . . 88
7.8 Side Channel Attacks . . . 88
7.8.1 Timing Attacks . . . 88
7.8.2 Differential Power Analysis . . . 90
7.8.3 Differential Fault Analysis . . . 91
7.9 Zero-Knowledge Proofs . . . 92
7.9.1 Zero-Knowledge Proof Using Graph Isomorphisms . . . 92

8 Conclusions **94**
8.1 Mathematics and Cryptology . . . 94
8.2 Applications and Implications . . . 95
8.3 Conclusion . . . 95

Bibliography **97**

A DES Implementation **99**
A.1 Results . . . 99
A.2 Source Code . . . 102

B Differential Cryptanalysis Implementation **141**
B.1 Results . . . 141
B.2 Source Code . . . 144

List of Figures

4.1 DES Feistel Network 26
4.2 DES Round Function 28
4.3 DES S-Box Example 29
4.4 DES Key Schedule 32
4.5 DES 3-round Characteristic 39
4.6 Characteristic Probability Calculation 41
4.7 Linear Approximation of S-Box 5 46

6.1 Feedback Shift Register 63

7.1 Precalculation for Time–Memory Trade-Off 87
7.2 Timing and size of SSH packets when an `su` command is initiated 90

List of Tables

3.1 English Letter Frequency 17
3.2 Index of Coincidence 18
3.3 Mutual Index of Coincidence 20

4.1 P-Box Design Criteria 28
4.2 S-Boxes 30
4.3 DES Weak Keys 34
4.4 DES Semi-Weak Key Pairs 34
4.5 XOR Difference Distribution for S-Box 1 37

7.1 Some Digital Signature Algorithms 78

Chapter 1

Introduction

Cryptography has as its root of origin the basic need of different parties to communicate, with only the intended recipients gaining access to the information. However, this can be achieved in many different ways, the simplest being to physically conceal the transmitted data from all but the intended recipients.

Cryptography includes all mechanisms for concealing the information content of a message, even when the message has been intercepted by illegitimate parties. From this one might infer that practises such as using invisible ink, or tiny pin punctures above selected characters, might fall under the title of cryptography. This is not the case, since here the opponent has not found the data comprising the actual message, but merely yet another guise. Such tricks as invisible ink fall under the title of steganography.

Thus cryptography attempts to render the data of a message as useless to any opponent intercepting it, yet enabling the intended recipient to uncover the meaning of the message. That is, the sender encrypts the message and the recipient decrypts the message.

In addition to confidentiality which is achieved by encryption, there are other fields of information security such as data integrity, authentication and non-repudiation [2, page 4]. Together, the disciplines that incorporate various methods of information security are referred to as *cryptography*. The methods and techniques that are developed to attempt to compromise the results of cryptography, are referred to as *cryptanalysis*.

Cryptology refers to the field of study that includes both cryptography and cryptanalysis. In the study of modern cryptology, it is important to understand the techniques that are employed, as well as the mathematics that is used to study and further develop the field.

1.1 History

The development and use of cryptology started slowly and occurred independently in many different cultures. Kahn [20] provides a comprehensive description of the history of cryptology.

It is difficult to pinpoint the exact beginning of cryptography. However, the inscriptions carved into the walls of the main chamber of the tomb of the nobleman Khnumhotep II, provide the first example of deliberate transformation of writing. The tomb is to be found in the town of Menet Khufu bordering on the Nile in Egypt, and the inscriptions date to approximately 1900 BC. The intention of the transformations performed by the scribe was not that of concealment, but most likely to impart dignity and authority. Yet the presence of such intentional transformations demonstrate that the fundamental concepts of cryptography were beginning to develop within the

culture. In tombs built after 1900 BC the occurrence of transformations became more complicated, more contrived and more prolific.

Several forms of secret writing were known and apparently practised in India. The *Artha-śāstra* is a classic work on statecraft that is attributed to Kautilya and was written sometime between 321BC and 300BC. This work recommends that institutes of espionage communicate with their spies via secret writing. Secret writing is even listed in Vātsayāyana's famous textbook of erotics, the Kāma-sūtra, as one of the 64 arts (or yogas) that women should know and practise. The yoga is called "mlecchita-vikalpā" of which two types of secret writing are described:

"kautiliyam" letter substitutions are based upon phonetic relations e.g. vowels become consonants.

"mūladevīya" The cipher alphabet consists of the reciprocal one. This version has been mentioned widely in literature and variations have been known to be used by traders.

Another form of secret communication that developed in India is "nirābhāsa." This is a finger based communications system where the phalanges stand for consonants and the joints stand for vowels. This method of communications is still used by moneylenders and traders. It also forms the basis for sign language which is used by deaf and dumb people.

In contrast to India, China did not develop the same level of cryptographic skills. This can be largely attributed to the difference in the level of literacy between the two countries. Compared to China, India has a very high literacy level. In China writing had been in existence for a long time, but due to low literacy cryptography only developed much later. An astute comment was made by Prof. Owen Lattimore of the University of Leeds, "Although writing is extremely old in the Chinese culture, literacy was always restricted to such a small minority that the mere act of putting something into writing was to a certain extent equivalent to putting it into code." Hence, it is reasonable to assert that one of the driving forces for the development of cryptography is the continued need for people to be able to converse in a private manner.

Another reason that cryptography develops is to protect intellectual property. In Mesopotamia a tablet, dating to 1500BC, was found which contains the earliest know formula for the making of glazes for pottery. To guard this professional secret, the formula was written in an enciphered form of cuneiform. The method of encipherment demonstrates that the author was explicitly attempting to conceal the information contained within the message that was on the tablet. The method relies on performing transformations of the written text of the message, based on correlations between phonetic sounds and the written equivalent. For example: In English, the word `fish` could be enciphered as `GHOTI` using the following correlations:

`f` $\rightarrow$ `GH` : tou*gh*,
`i` $\rightarrow$ `O` : w*o*men,
`sh` $\rightarrow$ `TI` : na*ti*on.

As knowledge of glaze-making spread, the need for secrecy evaporated, and latter formulae and descriptions were written in plain language.

Since cryptography is used to protect a secret, it is to be expected that unintended recipients would attempt to decipher the meaning of the encrypted message. The first record of active cryptanalysis comes from the Arabs during the 700s. Formal techniques, such as letter frequency analysis, only came into being over the next few hundred years. Letter frequency analysis was known by the time the *Subh al-a 'sha*, an enormous 14-volume encyclopedia, was completed in 1412. The information for the cryptography section was mostly attributed to Ibn ad-Duraihim, who lived from 1312 to 1361 and held various teaching and official posts in Syria and Egypt.

Early cryptosystems usually relied on transformations of the plaintext message being performed by the person composing the message. However, as the complexity of the methods increased it became desirable to create tools/machines that would perform the cryptographic tasks. The earliest known device designed specifically for cryptography is the "skytale." The tool was devised in the 5th century BC by the Spartans, the most warlike of the Greeks, so as to augment their military system. The tool consists of a wooden staff around which a strip of papyrus, leather or parchment is wrapped in a close-packed manner. The secret message is written on the strip down the length of the staff. The strip is then unwound and sent to the intended recipient. The letters on the strip make no sense unless the strip is wrapped around a baton of the same thickness.

Another ingenious system was the use of an astragal or disk, with holes in it – one for each letter of the alphabet. A thread is passed from one hole to another, spelling out the message. To recover the message the recipient would need to work backwards as the thread is removed from the holes and then reverse the resultant message.

Current cryptosystems use digital computers to carry out the thousands of calculations needed for modern cryptographic transformations. In some cases general purpose computers are insufficient and dedicated hardware is developed in order to handle the large quantity of data that is to be encrypted and decrypted.

During the middle ages cryptology acquired a taint that lingers even today – the conviction in the minds of many people that cryptology is a black art. Cryptology was linked to magic, since "spells" and recipes for curses were often encoded. Cryptology also resembles divination since, to the layman, extracting an intelligible message from ciphertext seems like exactly the same thing as reading tea leaves or examining the length and intersections of lines in the hand. Even in 1940 the U.S. referred to its Japanese diplomatic cryptanalyses by the codename MAGIC.

Irrespective of the mystical perception that clouded the field of cryptology during the middle ages, the science made steady albeit slow progress. The 1600s saw an important advance with the first use of a word as a mnemonic key for mixing a cipher alphabet. The growing number of diplomatic messages being sent throughout the western world, and the requirements of military groups maintained the need for continued development.

The importance of cryptology to the military has continued to increase. There are many examples of wars for which the outcome has been radically altered based on information gained via the cryptanalysis of intercepted messages. Similarly, there are many outcomes that would not have occurred if the information had not been protected using cryptography.

In 1914, England highlighted the importance of information when their first offensive act against Germany was to sever Germany's transatlantic communication cables. Germany was then forced to communicate via radio or enemy controlled cables. Her only defence was to protect the communications using cryptography. At that time England did not have any department for dealing with enemy cryptograms. This was soon availed when Sir Alfred Ewing began to rally expertise to cryptanalyse the German messages. Initially progress was slow, even with the help of a German codebook that was recovered from a wreck in the Baltic. This department became known as "Room 40." After the war it was estimated that from October 1914 to February 1919, Room 40 solved over 15000 German intercepts.

One of the most famous intercepted messages to be successfully cryptanalysed, became known as the "Zimmermann telegram." This was sent by the German Foreign Minister, Arthur Zimmermann, on the 16th of January, 1917 to a German Ambassador in the United States. When "Room 40" solved the telegram, over a month later, the English military read that Germany was intending to wage unrestricted submarine warfare and that they would not stop even if this provoked the United States into joining the war. Instead they invited Mexico into an alliance wherein Mexico

could reclaim lost territory. When the U.S. was informed this galvanised them into action and they discarded their neutral stance and joined the war. This tipped the balance against Germany.

1.2 Mathematics and Cryptology

As we have seen, the initial cryptographic systems employed very simple methods. These primarily included codes which are used directly as substitutions for words or phrases in the plaintext message. Over time the methods began to use rules that described the transformations of individual letters, e.g. the Caesar cipher describes a simple transformation rule: `a` $\rightarrow$ `D, b` $\rightarrow$ `E, ..., z` $\rightarrow$ `C`.

An important advance was the use of keywords that governed the result of transformation rules. Through the use of keywords the cryptographic function could remain constant and could even be publicly known. The security relies on keeping the keyword secret. It has now been accepted to assume that the opponent knows the full workings of the cryptosystem and that security is based on keeping the key secret. This is usually referred to as Kerckhoff's principle.

In addition to improvements in the cryptosystems, the techniques of analysis also improved. Initially this was helped by the improvement of linguistic analysis which led to the use of letter frequency analysis and the use of techniques that exploit phonetic relationships of the underlying language.

As the science of cryptology progressed it became more mathematical. Many cryptographic functions are directly constructed from mathematical problems that are difficult to solve. These problems come from a diverse range of fields including number theory, geometry and lattice theory. Results from different fields are also used to motivate the security of cryptographic functions when reasonable assumptions are made. This is evident in the design of modern block ciphers, pseudo-random number generators and other cryptographic primitives.

Likewise, the methods for cryptanalysing messages have benefited greatly from mathematical knowledge and techniques. Probability theory has been extensively used and is pivotal to the understanding and development of non-deterministic attacks. Advances in number theory have led to more efficient solutions of systems based on the difficulty of factoring products of large primes, or the discrete logarithm problem.

The study of cryptology is heavily dependent on many fields of mathematics and has also been the inspiration for new mathematical results. In what follows we provide a brief overview of the basics of cryptography and cryptanalysis, referencing and utilising the necessary mathematics.

1.3 Outline

In the following chapters we introduce the basic concepts of cryptography and with this grounding we are able to focus on some of the more important aspects of modern cryptology such as block ciphers, differential and linear cryptanalysis, and pseudo-random number generators.

Chapter 2 introduces the basic nomenclature that is used in the field of cryptology. We then proceed to describe many trivial cryptographic algorithms (the shift cipher, affine cipher, substitution cipher, Vigenere cipher and the Hill cipher) which provide insights into many of the fundamental concepts of cryptosystems.

Chapter 3 introduces the notion of cryptanalysis. We then describe and explain how the simple cryptosystems of Chapter 2 can be compromised.

Chapter 4 introduces block ciphers. The ciphers in this class are the workhorses of modern cryptosystems. A detailed discussion of the Data Encryption Standard (DES) cipher is presented.

Using the context of DES we describe and explain how differential cryptanalysis can be used to compromise the security of DES-like cryptosystems. A brief description of linear cryptanalysis follows.

Chapter 5 introduces the concept of public-key cryptography as we present an overview of knapsack based cryptosystems and the cryptanalysis thereof. Although knapsack systems are not likely to ever be used in practical implementations, the number-theoretic nature of the algorithms and cryptanalysis generates a large amount of interest from researchers.

Chapter 6 discusses Pseudo-Random Number Generators (PRNGs). This includes providing useful definitions of randomness and discussing the attributes that a PRNG should exhibit, for the purposes of security, when being incorporated into a cryptosystem. Many of the commonly used algorithms are described.

Chapter 7 provides a brief insight into the many varied concepts, components and algorithms that might be used in cryptosystems. These include hash functions, digital signatures, key distribution and key agreement, side channel attacks and zero knowledge proofs.

Chapter 2

Cryptographic Basics

This chapter introduces the basic concepts of cryptography. This includes the general functionality of cryptosystems and the terminology used to describe the systems and their use. Also included are descriptions of many of the basic cryptographic algorithms that help provide an additional understanding of the basics.

2.1 Terminology

The primary use for *cryptosystems* is to enable two people, Alice and Bob (as they are commonly referred to in literature), to communicate over an insecure channel in a manner that prevents an opponent, Oscar, from being able to understand the conversation.

To achieve this Alice would convert her original message, the *plaintext*, into a message which is only intelligible by Bob, the *ciphertext*. This process is known as *encryption*. When Bob receives the message he will decrypt the ciphertext to reveal the original message that was sent. Because only Alice and Bob have the key to the encryption algorithm, Oscar will be unable to reconstruct the plaintext, even if he intercepts the ciphertext.

In addition the encryption and decryption algorithms used to convert between the plaintext and ciphertext and vice versa, a clear set of steps is needed to define how the data is to transfered between Alice and Bob. This is called the *protocol*. The protocols used in cryptosystems are implemented to ensure that the participants achieve the desired goal of communication within the constraints of the environment, whilst adhering to the assumptions made during the construction of the components of the system.

Once the logic of the cryptosystem has been designed the system needs to be implemented. In most cases this would mean that computers need to be programmed to carry out the encryption and decryption and that the computers need to be instructed to communicate in strict accordance to the protocols adopted.

Thus a cryptosystem consists of cryptographic algorithms, protocols and an implementation. The security of any cryptosystem depends on all parts from which it is built. If an adversary wishes to break a cryptosystem, that is to affect the security of the communication in an adverse way, then he could attack any combination of the cryptographic algorithms, the protocol or the implementation.

Below we will focus on the cryptographic algorithms, however when discussing cryptanalysis in later chapters some of the methods attack the protocols (e.g. man-in-the-middle attacks) and other methods attack the implementation (e.g. differential power analysis).

2.2 Cryptographic Algorithms

Cryptographic algorithms are the workhorses of a cryptosystem. The most important class of algorithms are the encryption/decryption algorithms. A broad definition of a cipher can be given as follows:

Definition 2.1 (Cipher) *A cipher is a five-tuple* $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ *with:*

1. $\mathcal{P}$ *is a finite set of possible plaintexts*
2. $\mathcal{C}$ *is a finite set of possible ciphertexts*
3. $\mathcal{K}$*, the key-space, is a finite set of possible keys*
4. *For each key* $K \in \mathcal{K}$*, there is an encryption rule* $e_K \in \mathcal{E}$ *and a corresponding decryption rule* $d_K \in \mathcal{D}$ *with the property that the functions* $e_K : \mathcal{P} \to \mathcal{C}$ *and* $d_K : \mathcal{C} \to \mathcal{P}$ *satisfy* $d_K(e_K(x)) = x \ \ \forall\, x \in \mathcal{P}$*.*

Thus for Alice and Bob to communicate they must decide on a key to use, after which Alice would encrypt the message using e_K and pass the ciphertext to Bob over a possibly insecure channel. Once Bob receives the he ciphertext will use d_K to retrieve the original message.

Many different ciphers exist and it is useful to define some broad categories into which the different ciphers fall.

Types of ciphers:

Block: A block cipher operates by transforming a block of plaintext into a block of ciphertext (e.g. DES operates on 64 bit blocks).

Stream: A Stream cipher operates on an input stream of plaintext so as to generate an output stream of ciphertext (e.g. key-stream generator which generates a stream of bits which are then combined the input stream, using the bitwise exclusive-or operator (denoted by $\oplus$ and XOR), to create an output stream).

Symmetric: A symmetric cipher uses the same key for both encryption and decryption.

Asymmetric: An asymmetric cipher uses different keys for encryption and decryption. Often this is combined with the feature that the encryption key can be made public without impacting on the security of the cryptosystem (e.g. RSA).

2.3 Simple Ciphers

We now describe some simple ciphers [31, pages 3–24], many of which have been used in the past, but due to advances in computing power and cryptanalysis they are now considered insecure. These algorithms provide a good starting point for understanding the basics of ciphers and how totally different mechanisms can be used to perform encryption and decryption.

2.3.1 Shift Cipher

In the shift cipher, a ciphertext letter is constructed from a plaintext letter by counting forward in the plaintext alphabet from the plaintext letter by a given amount. The shift cipher is best described using modular arithmetic.

Recall that a is congruent to b modulo m, $a \equiv b \pmod{m}$, means that m divides $b - a$, where m is the modulus. From this we can define modulo arithmetic on the ring $\mathbb{Z}_m$ which is the set $\{0, \ldots, m-1\}$ with the operations $+$ and $\times$. The operations are the same as the operations on real numbers, except that the results are reduced modulo m (i.e. replaced by the remainder when divided by m). With these definitions it can be easily shown that $\mathbb{Z}_m$ exhibits that following properties:

- addition is *closed* i.e. $a, b \in \mathbb{Z}_m \Rightarrow a + b \in \mathbb{Z}_m$
- addition is *commutative* i.e. $a, b \in \mathbb{Z}_m \Rightarrow a + b = b + a$
- addition is *associative* i.e. $a, b, c \in \mathbb{Z}_m \Rightarrow (a + b) + c = a + (b + c)$
- 0 is an *additive identity* i.e. for $a \in \mathbb{Z}_m$ we have $a + 0 = 0 + a = a$
- the *additive inverse* of any $a \in \mathbb{Z}_m$ is $m - a$, i.e. $a + (m - a) = (m - a) + a = 0$ for any $a \in \mathbb{Z}_m$.
- multiplication is *closed* i.e. $a, b \in \mathbb{Z}_m \Rightarrow ab \in \mathbb{Z}_m$
- multiplication is *commutative* i.e. $a, b \in \mathbb{Z}_m \Rightarrow ab = ba$
- multiplication is *associative* i.e. $a, b, c \in \mathbb{Z}_m \Rightarrow (ab)c = a(bc)$
- 1 is a *multiplicative identity* i.e. for $a \in \mathbb{Z}_m$ we have $a \times 1 = 1 \times a = a$
- multiplication *distributes* over addition i.e. for any $a, b, c \in \mathbb{Z}_m$ we have $(a+b)c = (ac)+(bc)$ and $a(b + c) = (ab) + (ac)$

We can now define the shift cipher as applied to the English alphabet. Let $m = 26$ and now map each letter in A,...,Z to the corresponding number in 0,...,25.

Thus A $\leftrightarrow$ 0, B $\leftrightarrow$ 1, ..., Z $\leftrightarrow$ 25. The shift from the plaintext to the ciphertext and vice versa is described in the following definition.

Definition 2.2 (Shift Cipher) *$\mathcal{P} = \mathcal{C} = \mathcal{K} = \mathbb{Z}_{26}$. For $K \in \mathbb{Z}_{26}$ (i.e. $K \in \mathcal{K}$) and $x, y \in \mathbb{Z}_{26}$ we define e_K and d_K as:*

$$e_K(x) = x + K \bmod 26$$

and

$$d_K(y) = y - K \bmod 26.$$

To demonstrate the application of the shift cipher the following short quote regarding disbelief in cryptanalysis will be encrypted, `'Without the key, sir, excuse me if I believe the thing impossible.'` [20, page 153], using the shift cipher and choosing 17 as the key. First the punctuation and spaces are removed:

```
w i t h o u t t h e k e y s i r e x c u
s e m e i f i b e l i e v e t h e t h i
n g i m p o s s i b l e
```

Then numbers corresponding to the letters are written down:

22 8 19 7 14 20 19 19 7 4 10 4 24 18 8 17 4 23 2 20
18 4 12 4 8 5 8 1 4 11 4 8 21 4 19 7 4 19 7 8
13 6 8 12 15 14 18 18 8 1 11 4

Then, to each number, we add 17 and reduce each sum modulo 26. This gives:

13 25 10 24 5 11 10 10 24 21 1 21 15 9 25 8 21 14 19 11
9 21 3 21 25 22 25 18 21 2 21 25 12 21 10 24 21 10 24 25
4 23 25 3 6 5 9 9 25 18 2 21

From this list of numbers we construct the corresponding list of letters and write this down as the final ciphertext: `NZKYFLKKYVBVPJZIVOTLJVDVZWZSVCVZMVKYVKYZEXZDGFJJZSCV`.

To decrypt the message the cipher text would be converted back into numbers, then 17 would be subtracted from each number and the result reduced modulo 26. This list of numbers could then be converted back into letters to recover the plaintext (modulo spaces and punctuation).

2.3.2 Affine Cipher

Affine ciphers construct a mapping from the plaintext alphabet to the ciphertext alphabet using the following type of function:

$$e(x) = ax + b \bmod 26,$$

where a and b together act as the key for the cipher. Such functions are known as affine functions, hence the name of the cipher. Clearly when $a = 1$ the affine cipher is reduced to a shift cipher. For the above function to be useful as a cipher we need to show that there is an inverse function that can be used to decipher the message, and we need to show that this inverse is unique. Thus we need a better understanding of affine functions.

Clearly we need to show that $e(x)$ can be injective, and describe under what conditions this is the case. To answer this we look at the mathematics of congruences, and in this case specifically linear congruences [4, pages 106-114].

The final result that we are going to prove is:
The solution of the linear congruence

$$ax \equiv b \pmod{m}$$

is unique mod m *if and only if* $(a, m) = 1$ *and is given by*

$$x \equiv ba^{\phi(m)-1} \pmod{m}$$

where $\phi(m)$ *is the Euler-Totient function.*

To achieve this we need some definitions and intermediary results. Many of the elementary properties of modulo arithmetic have already been listed in §2.3.1 when describing $\mathbb{Z}_m$.

We proceed to define residue classes and complete residue systems.

Definition 2.3 (Residue Class) *Let* $\hat{a}$ *be defined by* $\hat{a} = \{x \in \mathbb{Z} | x \equiv a \pmod{m}\}$. *Then* $\hat{a}$ *is the residue class of* a *modulo* m.

We provide, without proof, the following simple theorem about residue classes.

Theorem 2.4 *For a given modules* m *we have:*

1. $\hat{a} = \hat{b}$ *if and only if* $a \equiv b \pmod{m}$.
2. $x, y \in \mathbb{Z}$ *belong to the same residue class if and only if* $x \equiv y \pmod{m}$.
3. *The* m *residue classes* $\hat{1}, \ldots, \hat{m}$ *are disjoint and their union is the set of integers.*

Definition 2.5 (Complete Residue System) *A set of* m *representatives, one selected from each of the residue classes* $\hat{1}, \ldots, \hat{m}$*, is called a complete residue system modulo* m.

Definition 2.6 (Greatest Common Divisor) *Given* $a, b \in \mathbb{Z}$ *then the greatest common divisor of* a *and* b *is the largest* $d \in \mathbb{Z}$ *with* $0 < d \leq a$ *and* $0 < d \leq b$ *such that* d *divides* a *and* d *divides* b. *The greatest common divisor is commonly denoted by* $\gcd(a, b)$ *or just* (a, b).

Theorem 2.7 *If* $ac \equiv bc \pmod{m}$ *and if* $d = (m, c)$ *then*

$$a \equiv b \pmod{\frac{m}{d}}.$$

Theorem 2.8 *Assume* $(k, m) = 1$*, where* (k, m) *denotes the greatest common divisor of* k *and* m. *Given* $\{a_1, \ldots, a_m\}$ *is a complete residue system modulo* m*, then so is* $\{ka_1, \ldots, ka_m\}$.

PROOF: If $ka_i \equiv ka_j \pmod{m}$ then $a_i \equiv a_j \pmod{m}$ since $(k, m) = 1$. However, the a_i form a complete residue system, therefore no two elements in $\{ka_1, \ldots, ka_m\}$ are congruent modulo m. Thus, since there are m elements in the set, it too forms a complete residue system. □

We need to introduce one additional concept before we complete our task. If $(a, m) = 1$ then a is said to be relatively prime to m. The function $\phi(m)$ counts the number of positive integers less that m that are relatively prime to m. $\phi(m)$ is called the Euler-Totient function. It can be shown that the value of $\phi(m)$ can be found using the following expression:

$$\phi(m) = m \prod_{p|m} \left(1 - \frac{1}{p}\right),$$

where the product runs over all primes, p, that divide m. Additionally we also have the following theorem.

Theorem 2.9 (Euler-Totient) *Assume* $(a, m) = 1$. *Then we have*

$$a^{\phi(m)} \equiv 1 \pmod{m}.$$

We can now show under what condition an affine function has a unique solution.

Theorem 2.10 *If* $(a, m) = 1$ *the solution of the linear congruence*

$$ax \equiv b \pmod{m}$$

is unique mod m *and is given by*

$$x = ba^{\phi(m)-1} \bmod m$$

where $\phi(m)$ *is the Euler-Totient function.*

PROOF: First we show that the solution is unique. Clearly any solution would be in the set $\{1, \ldots, m\}$ since these numbers form a complete residue system. Now consider the set of products $\{a, 2a, \ldots, ma\}$. By Theorem 2.8 together with the assumption $(a, m) = 1$ we have that this last set also forms a complete residue system. Hence, exactly one of these products is congruent to b modulo m. That is, there is exactly one solution to the congruence.

To calculate the actual value of the solution we can use the Euler-Totient theorem to find $a^{-1} = a^{\phi(m)-1} \bmod m$. This completes the proof. □

We can now give a definition for the affine cipher.

Definition 2.11 (Affine Cipher) *$\mathcal{P} = \mathcal{C} = \mathbb{Z}_{26}$ and $\mathcal{K}$ is defined by:*

$$\mathcal{K} = \{(a, b) \in \mathbb{Z}_{26} \times \mathbb{Z}_{26} | (a, 26) = 1\}.$$

For $K = (a, b) \in \mathcal{K}$ with $x, y \in \mathbb{Z}_{26}$ we define e_K and d_K as:

$$e_K(x) = ax + b \bmod 26$$

and

$$d_K(y) = a^{-1}(y - b) \bmod 26$$

where $a^{-1} = a^{\phi(26)-1}$ and $\phi(26) = 12$ as calculated using the definition of ϕ.

2.3.3 Substitution Cipher

The substitution cipher is another simple cipher that has been employed for hundreds of years. In this cipher letters of plaintext are substituted with letters from a randomly chosen permutation of the plaintext alphabet. This permutation, denoted by π, operates as the key for the cipher thus yielding the following definition for the substitution cipher.

Definition 2.12 (Substitution Cipher) *$\mathcal{P} = \mathcal{C} = \mathbb{Z}_{26}$. $\mathcal{K}$ consists of all possible permutations of the sequence $\{0, \ldots, 25\}$. For each permutation $\pi \in \mathcal{K}$, with π^{-1} the inverse of π we define:*

$$e_\pi(x) = \pi(x)$$

and

$$d_\pi(y) = \pi^{-1}(y).$$

Both the shift cipher and the affine cipher, described above, are special cases of the substitution cipher. The key to either the shift cipher or the affine cipher selects one permutation from a subset of the 26! possible permutations, with the definition of the cipher providing a mechanism for constructing the permutation.

2.3.4 Vigenere Cipher

With the substitution cipher and the special cases thereof, once a key is chosen, each letter of the plaintext alphabet is always mapped to the same letter in the ciphertext alphabet. Thus these ciphers are called *monoalphabetic*. The Vigenere cipher, name after Blaise de Vigenere of the sixteenth century, is however a *polyalphabetic* cipher. That is, the same letter of the plaintext alphabet might be mapped to a different ciphertext letter depending on its position within the plaintext.

With the Vigenere cipher we once again use the correspondence $A \leftrightarrow 0$, $B \leftrightarrow 1, \ldots, Z \leftrightarrow 25$. The key is chosen to be some m letter alphabetic string. To perform the encryption we convert both the key and the plaintext to their corresponding numerical sequences. The encryption function operates on strings of length m at a time, so we break the plaintext sequence up into sequence of length m. Each block is then added, modulo 26, to the key sequence and the result is combined to form the ciphertext sequence which is then converted back into the alphabetical representation.

Below is an example where the plaintext is: `'Meet me in the alley at ten o'clock.'`. The key is chosen to be `'BLAISE'`. The plaintext sequence is thus:

$$\begin{array}{cccccccccccccccccccccc} 12 & 4 & 4 & 19 & 12 & 4 & 19 & 7 & 4 & 0 & 11 & 11 & 4 & 24 & 0 & 19 & 19 & 4 & 13 & 14 \\ 2 & 11 & 14 & 2 & 10 \end{array}$$

and the key sequence is:

$$1 \quad 11 \quad 0 \quad 8 \quad 18 \quad 4$$

We then add each block of 6 together, the first block would be added up as follows:

$$\begin{array}{cccccc} 12 & 4 & 4 & 19 & 12 & 4 \\ 1 & 11 & 0 & 8 & 18 & 4 \\ \hline 13 & 15 & 4 & 1 & 4 & 8 \end{array}$$

Proceeding to add the key to each block of plaintext we would obtain the following ciphertext sequence:

$$\begin{array}{cccccccccccccccccccccc} 13 & 15 & 4 & 1 & 4 & 8 & 20 & 18 & 4 & 8 & 3 & 15 & 5 & 9 & 0 & 1 & 11 & 8 & 14 & 25 \\ 2 & 19 & 6 & 6 & 11 \end{array}$$

with the final ciphertext being: `NPEBEIUSEIDPFJABLIOZCTGGL`.

We now give a definition for the Vigenere cipher.

Definition 2.13 (Vigenere Cipher) *$\mathcal{P} = \mathcal{C} = \mathcal{K} = (\mathbb{Z}_{26})^m$. For a key $K = k_1, \ldots, k_m$ we define*

$$e_K(x_1, \ldots, x_m) = (x_1 + k_1 \pmod{26}, \ldots, x_m + k_m \pmod{26})$$

and

$$d_K(y_1, \ldots, y_m) = (y_1 - k_1 \pmod{26}, \ldots, y_m - k_m \pmod{26}).$$

2.3.5 Hill Cipher

The Hill cipher is another polyalphabetic cipher that was developed far more recently than many of the previous ciphers. The Hill cipher was developed by Lester S. Hill and the details were first published in a paper entitled "Cryptography in an Algebraic Alphabet" in *The American Mathematical Monthly* for June-July 1929 [20, page 404].

The main idea is to use linear combinations to transform a block of plaintext letters into a block of ciphertext characters. If it were given that each block consists of m characters there would be m linear equations producing a ciphertext block of m characters. To represent this we once again transform the plaintext message into a sequence of numbers, that is, if the plaintext was formed from the 26 letters of the English alphabet, we would transform it into the corresponding numbers from $\mathbb{Z}_{26}$. All the linear combinations are calculated in $\mathbb{Z}_{26}$, that is using modulo arithmetic.

Below is an example of a possible linear combination used to encrypt blocks (x_1, x_2, x_3) of length $m = 3$ to create ciphertext blocks:

$$\begin{aligned} y_1 &= 16x_1 + 8x_2 + 9x_3 \\ y_2 &= 6x_1 + 23x_2 + 25x_3 \\ y_3 &= 21x_1 + 5x_2 + 11x_3. \end{aligned}$$

Using this and the plaintext `'Retreat to the South'` the first plaintext block would be $(17, 4, 19)$. Using the plaintext block and the linear combination we can calculate that the corresponding ciphertext block is $(7, 19, 14)$. Thus the first three letters of the ciphertext would be `HTO`.

The most convenient method to describe linear combinations is using matrix representations from linear algebra. The key to the cipher is the matrix of constants used in the linear combinations. Thus the above linear combination could be written as:

$$(y_1, y_2, y_3) = (x_1, x_2, x_3) \begin{bmatrix} 16 & 6 & 21 \\ 8 & 23 & 5 \\ 9 & 25 & 11 \end{bmatrix}$$

and matrix multiplication, with the operations carried out modulo 26, can be used to perform the encryption.

As seen above the decryption process requires one to perform the inverse of the encryption operation. Using the matrix notation it is clear that the decryption process would use the inverse of the key matrix, K. From linear algebra we know that a matrix is invertible if and only if the determinant is non-zero. However, the matrix operations above occur in $\mathbb{Z}_{26}$ and thus the same determinant rule does not apply. The appropriate rule for matrices over $\mathbb{Z}_{26}$ is that a matrix K has an inverse modulo 26 if and only if $\gcd(\det(K), 26) = 1$.

From the above discussion we can give the following definition for the Hill Cipher.

Definition 2.14 (Hill Cipher) *Let $m \in \mathbb{Z}$ be the block size. $\mathcal{P} = \mathcal{C} = (\mathbb{Z}_{26})^m$. And let*

$$\mathcal{K} = \{m \times m \text{ invertible matrices over } \mathbb{Z}_{26}\}.$$

That is, for any $K \in \mathcal{K}$, K is an $m \times m$ matrix with $\gcd(\det(K), 26) = 1$. Then for a given key K we can define the encryption and decryption as:

$$e_K(x) = xK$$

and

$$d_K(x) = xK^{-1}$$

where all the operations are performed over $\mathbb{Z}_{26}$ and K^{-1} is the inverse of K.

2.4 Conclusion

The above cryptographic algorithms introduce us to many of the basic concepts of cryptography. They introduce us to the process of encryption and decryption, to the notion of plaintext and ciphertext and to the notion of an encryption key.

Many of the above algorithms have, in the past, been used in real world scenarios or have been extended and implemented as encryption machines – such as the Alberti disk (invented by Leon

Battista Alberti in 1400s) or the Jefferson wheel (invented in the late 1700s by Thomas Jefferson) which are both polyalphabetic substitution ciphers.

As we will see in the next chapter, all of the above simple cryptographic algorithms have been rendered insecure through advances in mathematics, cryptanalytic techniques and ability of computational machinery. Thus, new cryptographic algorithms rely on new techniques and are often designed around hard mathematical problems.

Chapter 3

Introduction to Cryptanalysis

Cryptanalysis is the study of the techniques used to break information security systems. The attacks include attempting to extract the plaintext from a ciphertext message without having access to the encryption key, or attempting to recover the encryption key when only the ciphertext is known. (Clearly this excludes employing large-men-with-darkglasses-and-big-guns to extract the key from the unwilling victim, sometimes affectionately referred to as rubber-hose cryptanalysis.)

Hence a cryptographic algorithm is said to be broken if there exists a method (probabilistic or deterministic) that can retrieve the key or plaintext with a complexity less than that of a brute force attack (i.e. exhaustive search of the key space).

The techniques in this chapter will focus on the cryptanalysis of simple ciphers such as those described in the previous chapter.

3.1 Modes

It is assumed that the attacker knows the full workings of the cryptosystem in use. To attack a cryptosystem the attacker needs to gather data from the cryptosystem in question. The level of data that can be acquired changes the complexity of the attack and which methods can be used. Below is a list of the common modes of attacks [31, page 25]:

Ciphertext-only

The attacker has acquired a string a cipher text.

Known Plaintext

The attacker has acquired a string of plaintext and the corresponding ciphertext.

Chosen Plaintext

The attacker has acquired access to the encryption machinery and can choose a plaintext string and then construct the corresponding ciphertext.

Chosen Ciphertext

The attacker has acquired access to the decryption machinery and can choose a ciphertext string and then construct the corresponding plaintext.

In each case the attacker attempts to recover the key that was used by working back from the data gathered.

3.2 Cryptanalytic Methods

There are many different methods of cryptanalysis. As new ciphers are created and designed, new methods of breaking them are devised. The previous chapter described some simple ciphers that are good for explaining many of the basic concepts of encryption and decryption but due to advances in cryptanalysis they are no longer secure for real world use. We shall now introduce some cryptanalytic techniques that would be used break the aforementioned simple ciphers, and in so doing we shall introduce some of the basic concepts of cryptanalysis.

3.2.1 Exhaustive Search

The first and most rudimentary method of cryptanalysis is the *exhaustive search*, otherwise know as a *brute force attack*. To implement the attack the opponent simply tries to search through the entire key space. Thus, any cipher that uses a key space that can be feasibly searched will be easily broken. The size of the key space necessary to prevent such an attack is dependent on the computational complexity of the cipher and the type of computing system available to the opponent. When digital computers were first to be developed and applied to cryptanalysis it drastically affected the lower bound necessary to maintain security because the computers could perform calculations much faster than humans could. As computing machinery advances the lower bounds for key space sizes increase. Clearly the shift cipher can be easily broken because there are only 26 different keys. The affine cipher is very insecure with $\phi(26) \times 26 = 312$ keys. If we examine the Vigenere cipher, it is soon evident that modern computers can quickly break it. If the Vigenere cipher has a key word with length in the range $[1, 6]$ the number of key words would be $\sum_{i=1}^{6} 26^i = \frac{26^7-1}{26-1} - 1 = 321272406$, and thus a personal computer could test all the key words in a reasonable amount of time.

Initially DES was considered sufficiently secure, since it uses a key that is 56 bits long, which means that there are in the order of 7.2×10^{16} different keys. However, with the advancement of computers DES is no longer considered secure against a brute force attack.

3.2.2 Letter Frequency

Letter frequency is commonly used to check if a type of permutation cipher was applied and thus if the letter distribution still matches the language of the plaintext. Using this information it is often possible to construct a small number of plausible decryptions from which the correct one could be easily identified.

Table 3.1 shows the expected letter frequencies of the 26 letters of the English alphabet for standard English text.

We now show how letter frequency might be used to recover the plaintext from the message ciphertext created with the affine cipher. First recall that the key to the affine cipher consists of two numbers $a, b \in \mathbb{Z}_{26}$ such that $(a, 26) = 1$, and that the encryption function is $e_K(x) = ax+b \bmod 26$. Thus, if we know the ciphertext letters of any two plaintext letters we can set up a simultaneous equation to solve for both a and b.

By calculating the frequencies of each letter in the ciphertext and comparing these to the expected frequencies for the plaintext language, we can make reasonable guesses at a correspondence between some of the ciphertext letters and their plaintext counter parts. For example, if the plaintext is in English then Table 3.1 tells us that the most commonly occurring ciphertext letter is most probably the encryption of the letter `e` and similarly the second most common letter probably corresponds to `t`.

Table 3.1: English Letter Frequency

Letter	Probability	Letter	Probability
A	.082	N	.067
B	.015	O	.075
C	.028	P	.019
D	.043	Q	.001
E	.127	R	.060
F	.022	S	.063
G	.020	T	.091
H	.061	U	.028
I	.070	V	.010
J	.002	W	.023
K	.008	X	.001
L	.040	Y	.020
M	.024	Z	.001

Letter	Probability	Letter	Probability
E	.127	M	.024
T	.091	W	.023
A	.082	F	.022
O	.075	G	.020
I	.070	Y	.020
N	.067	P	.019
S	.063	B	.015
H	.061	V	.010
R	.060	K	.008
D	.043	J	.002
L	.040	Q	.001
C	.028	X	.001
U	.028	Z	.001

Such a direct application of letter frequencies can only be carried out with monoalphabetic ciphers. The Vigenere cipher can thus not be easily cryptanalysed using this approach. The next two sections describe methods that can be combined to attack the Vigenere cipher.

3.2.3 Kasiski Test

The Kasiski test can be used to identify the length of the keyword used to create a ciphertext message encrypted by using the Vigenere cipher.

The key idea behind this test is the observation that any two identical substrings of the plaintext will encrypt to identical ciphertext substrings when they are both aligned equally relative to the keyword boundaries. That is to say that the distance between the starting positions of each substring is a multiple of the keyword length.

In converse, if we identify multiple substrings of the ciphertext that are identical and of a sufficient length (say three or more letters long) it is highly probable that they are encryptions of identical plaintext. Thus, the length of the keyword must divide the greatest common divisor of the differences in starting positions of each of the ciphertext substrings.

In brief, we implement the Kasiski test by searching for multiple occurrences of identical substrings in the ciphertext, where the substrings are three or more letters long. We then calculate the distance between the starting points of each pair of identical substrings giving a sequence $d_1, \ldots, d_n$. The length of the keyword, m, then divides the greatest common divisor of each of the d_i's.

3.2.4 Index of Coincidence

Like the Kasiski test, the index of coincidence provides a tool that can be used to identify the length of the keyword used when encrypting a message using the Vigenere cipher.

The index of coincidence was developed by Wolfe Friedman in 1920. This technique was a landmark discovery in cryptography as it was the first true statistical method to be used [20, page 376]. The method explicitly led to the construction of PURPLE, a cipher machine developed by the American intelligence to decrypt Japanese messages that were intercepted during World

Table 3.2: Index of Coincidence

Language	Index of Coincidence
English	0.0656
French	0.0778
German	0.0762
Italian	0.0738
Spanish	0.0775
Russian	0.0529

War II [20, page 385]. The method also lay the ground work for many further developments in cryptography.

The index of coincidence measures a statistical property of the text. The property that is measured is the probability that two letters chosen at random will be identical. For English text the value is different from random text. Furthermore, this value is invariant under shifting as would occur in the application of the shift cipher. This last point is the pivotal idea to finding the most probably keyword length used in a Vigenere cipher.

Definition 3.1 (Index of Coincidence) *Let* $\mathbf{x} = x_1x_2\ldots x_n$ *be a string of characters. The index of coincidence of* $\mathbf{x}$, $I_c(\mathbf{x})$, *is defined to be the probability that two distinct randomly selected elements of* $\mathbf{x}$ *are identical. Let the frequencies of the letters A,B,...,Z in* $\mathbf{x}$ *be given by* $f_0, f_1, \ldots, f_{25}$ *respectively. Two elements of* $\mathbf{x}$ *can be chosen in* $\binom{n}{2}$ *ways, and there are* $\binom{f_i}{2}$ *ways of choosing both letters to be the* i_{th} *letter. Hence we have*

$$I_c(\mathbf{x}) = \frac{\sum_{i=0}^{25}\binom{f_i}{2}}{\binom{n}{2}} = \frac{\sum_{i=0}^{25} f_i(f_i-1)}{n(n-1)}. \tag{3.1}$$

We can use the probabilities of occurrence of letters in the English language from Table 3.1 to approximate the index of coincidence for English. If the probabilities of the letters A, B, ..., Z are denoted by $p_0, p_1, \ldots, p_{25}$ respectively, we see that we can use (3.1) to obtain the following approximation:

$$I_c \approx \sum_{i=0}^{25} {p_i}^2 = 0.0656.$$

Similarly, we can calculate the index of coincidence for totally random text. Here the probability will be $\frac{1}{26}$ for each different letter, so clearly we have $I_c \approx 26(\frac{1}{26})^2 = 0.0385$.

Thus, it is seen that different alphabets and different languages would exhibit different values for the index of coincidence, see Table 3.2.

We can now describe how the index of coincidence is used to unveil the length of a keyword used in the Vigenere cipher. Let the ciphertext be represented as the string of characters $\mathbf{z} = z_0\ldots z_{n-1}$ and assume the key length is some integer m. Then construct the strings $\mathbf{y}_i$, $0 \le i \le m-1$ such that the character at position j in $\mathbf{y}_i$ is z_{jm+i}. Conceptually, this is equivalent to writing out the ciphertext into a grid of width m and then constructing each $\mathbf{y}_i$ by reading off the i_{th} column. We now observe that if the keyword length is m then each column, or $\mathbf{y}_i$, has essentially been created by a shift cipher operating on the corresponding column of plaintext and using the i_{th} keyword letter as the key for the shift operation. However, if our assumption regarding the keyword length

is incorrect then the letters of the i_{th} column will not all have been encrypted using the same keyword letter. In the light of the discussion of the index of coincidence and the observation that the index is invariant under shift operations we notice that if the assumption for the keyword length is correct then the index of coincidence of each column will match that of the underlying language, e.g. 0.0656 for English, but if the assumption is incorrect the letters of each column will be more randomly distributed and thus the index of coincidence will be closer to 0.0385. Hence, by iterating over a range of plausible keyword lengths we can identify the most probably keyword length.

3.2.5 Mutual Index of Coincidence

The Mutual Index of Coincidence is an extension of of Index of Coincidence. This tool can be used to identify the relative shift of letters in a Vigenere Cipher keyword, thus reducing the problem of breaking the cryptosystem to a short exhaustive search.

The mutual index of coincidence is like the index of coincidence except that we are calculating the probability of coincidence when the letters are chosen from two different strings rather than from the same string. This then leads to the following definition.

Definition 3.2 (Mutual Index of Coincidence) *Let* $\mathbf{x} = x_1x_2\dots x_n$ *and Let* $\mathbf{y} = y_1y_2\dots y_{n'}$ *be a strings of* n *and* n' *characters respectively. The mutual index of coincidence of* $\mathbf{x}$ *and* $\mathbf{y}$*,* $\mathrm{MI}_c(\mathbf{x},\mathbf{y})$*, is defined to be the probability that a randomly selected element of* $\mathbf{x}$ *is identical to a randomly selected element of* $\mathbf{y}$*. Let the frequencies of the letters A,B,...,Z in* $\mathbf{x}$ *and* $\mathbf{y}$ *be given by* $f_0, f_1, \dots, f_{25}$ *and* $f'_0, f'_1, \dots, f'_{25}$ *respectively. Then by an argument similar to that for the index of coincidence we see that* $\mathrm{MI}_c(\mathbf{x},\mathbf{y})$ *is given by:*

$$\mathrm{MI}_c(\mathbf{x},\mathbf{y}) = \frac{\sum_{i=0}^{25} f_i f'_i}{nn'}.$$

As with the method of determining the length of the keyword, the method for confining the keyword search space relies on iterating over a range of possible parameter values and calculating the mutual index of coincidence at each iteration, so as to look for values that best match the theoretical predictions. In this method, the result of the search will be the relative shift or difference between each of the keyword letters. The relative shift of each letter together with the first letter of the keyword will uniquely define the rest of the keyword. Thus, once we have obtained the relative shift we can consider each of the 26 possible candidates for the first letter of the keyword and obtain a list of 26 possible keywords from which we can choose the one that is most likely.

From the Kasiski test or using the index of coincidence we can determine the keyword length, m. Now let $K = (k_1, \dots, k_m)$ denote the keyword and let $\mathbf{y}_1, \dots, \mathbf{y}_m$ denote the substrings constructed from columns of the ciphertext when the ciphertext is written in a block of width m. Then each column is encrypted by a shift cipher where the relative difference in the shift between $\mathbf{y}_i$ and $\mathbf{y}_j$ is dependent on the difference between k_i and k_j.

Knowing the relative difference between $\mathbf{y}_i$ and $\mathbf{y}_j$, and the frequency characteristic of the underlying plaintext, we can estimate the mutual index of coincidence. To do this we note that the probability of choosing an A from $\mathbf{y}_i$ and from $\mathbf{y}_j$ would be p_{0-k_i} and p_{0-k_j} respectively, where $p_0, \dots, p_{25}$ are the respective probabilities of the $A, \dots, B$ occurring in the plaintext. Hence we can estimate the mutual index of coincidence:

$$\mathrm{MI}_c(\mathbf{y}_i,\mathbf{y}_j) \approx \sum_{h=0}^{25} p_{h-k_i} p_{h-k_j} = \sum_{h=0}^{25} p_h p_{h+k_i-k_j}.$$

Table 3.3: Mutual Index of Coincidence

Relative Shift	**Estimate of** MI_c
0	0.065
1	0.039
2	0.032
3	0.034
4	0.044
5	0.033
6	0.036
7	0.039
8	0.034
9	0.034
10	0.038
11	0.045
12	0.039
13	0.043

This value depends only on the difference $l = k_i - k_j$ modulo 26. That is

$$MI_c(\mathbf{y}_i, \mathbf{y}_j) \approx \sum_{h=0}^{25} p_h p_{h+l} = \sum_{h=0}^{25} p_h p_{h-l}.$$

Using the Table 3.1 we can calculate approximate values for different relative shifts in the range $[0, 13]$ which gives the values in Table 3.3. Clearly a relative shift of zero can be easily distinguished from other relative shifts since the expected MI_c is 0.065 in the former case and should be in the range of 0.031 to 0.045 for all other shifts.

We now calculate MI_c for different relative shifts between pairs of $\mathbf{y}_i$s and tabulate the results. More explicitly, if we choose a particular pair, $\mathbf{y_i}$ and $\mathbf{y}_j$, we can calculate the MI_c for different relative shifts $0 \leq g \leq 25$ using the formula

$$MI_c^g(\mathbf{y_i}, \mathbf{y_j}) = \frac{\sum_{i=0}^{25} f_i f'_{i-g}}{nn'}.$$

When the value of $g = l$, the relative shift present in the key, the estimates in Table 3.3 indicate that the value of MI_c^g will be close to 0.065, otherwise MI_c^g will be closer to the range $[0.031, 0.045]$. By carrying out this procedure for each pair of $\mathbf{y}_i$ and $\mathbf{y}_j$ we can find the most probable relative shifts between each of the keyword letters. An exhaustive search should quickly lead to the correct keyword and hence the plaintext.

3.2.6 Generalisations

Many methods of cryptanalysis attempt to find properties/characteristics of the plaintext that are invariant under the cipher transformation or that are transformed in a known way (independent of the key). These then enable the cryptanalyst to infer from the cipher text, knowledge about the plaintext, or if the characteristic is altered due the key in a known manner, one might be able to infer information about the key from the cipher text.

Using the index of coincidence to analyse the Vigenere Cipher relies on the index being invariant under a monoalphabetic cipher. Using this, the cryptanalyst can recover the keyword length.

3.3 Conclusion

The methods discussed in this chapter provide the ground work for current methods. However, current methods are much more advanced and there are many different techniques.

As time progresses people generalise techniques, which makes it easier to apply the methods to a wider range of cryptosystems. At the same time, cryptanalysis of particular cryptosystems advances all the time. This is a double-edged sword for designers because there are more techniques that need to be taken into account so as to create a secure cryptosystem, but there is also a wealth of general test cases.

Over time cryptographers and the organisations employing them have learnt that security-through-obscurity is a fallacy and most cryptosystems that hold up to cryptanalysis are only deployed once the details of the algorithms are made public and are peer reviewed.

Chapter 4

Block Ciphers and Cryptanalysis

4.1 Overview

In this chapter we discuss symmetric-key block ciphers and the cryptanalysis thereof. Block ciphers are one of the most important components of many cryptosystems. Individually block ciphers are designed to provide confidentiality of data. Block ciphers can also be used as building blocks for creating other cryptographic components such as pseudo-random number generators, stream ciphers, hash functions and MACs.

A block cipher is an algorithm that defines a permutation of the space of fixed size blocks of data, where the permutation is parameterised by a key. We can define a block cipher as follows.

Definition 4.1 (Block Cipher) *Let the block cipher operate on n-bit blocks. Let $\mathcal{K}$ be the key-space. Then the block cipher is a function $\mathsf{E} : \mathbb{Z}_2^n \times \mathcal{K} \to \mathbb{Z}_2^n$, such that for each key, $\mathsf{K} \in \mathcal{K}$, we have that $\mathsf{E}(\mathsf{P}, \mathsf{K})$ is an bijective mapping, which is referred to as the encryption function and usually denoted by $\mathsf{E}_\mathsf{K}(\mathsf{P})$. The inverse mapping is called the decryption function and is denoted by $\mathsf{D}_\mathsf{K}(\mathsf{P})$.*

4.1.1 Modes of Use

As described above, block ciphers operate on fixed size blocks of data. Compared to the quantity of data that is encrypted in practise, it is seen that the block sizes are relatively small. For example, a page of printed text contains about 4000 characters (32000 bits when represented using the ASCII encoding), whereas DES only operates on 64-bit blocks and its successor, AES (Rijndael), operates on 128-bit blocks[1]. To be able to use block ciphers to encrypt messages whose lengths exceed the block size we need to introduce methods for applying the encryption function to the entire message – this is referred to as the mode of operation [2, page 228].

We now describe some of the more fundamental modes of operation. Many of the other modes of operation are derived from these fundamental versions.

Electronic Codebook mode

Electronic Codebook (ECB) mode is the simplest of all modes. Given that the block size is n, we break the message up into n-bit blocks (padding the last block as is necessary). Each block is then encrypted separately using the same encryption function and the same key. We can thus define ECB by the following algorithm.

[1] With Rijndael the block length and the key length can be independently specified to 128, 192 or 256 bits.

ALGORITHM: **ECB mode**:
INPUT: k-bit key K. Plaintext message x split into n-bit blocks $x_1, \ldots, x_t$.
OUTPUT: Ciphertext message c formed by the concatenation of n-bit blocks $c_1, \ldots, c_t$.

Encryption For $1 \leq j \leq t$ let $c_j \leftarrow E_K(x_j)$.

Decryption For $1 \leq j \leq t$ let $x_j \leftarrow D_K(c_j)$.

□

ECB, however, has many disadvantages and is not normally used. With ECB identical plaintext blocks result in identical ciphertext blocks, and the blocks are encrypted independently of each other. This makes ECB susceptible to malicious modification of individual blocks which would result in a modification of the corresponding plaintext blocks. Blocks could be removed, inserted, substituted or interchanged.

Cipher Block Chaining mode

Cipher Block Chaining (CBC) mode operates in a similar fashion to ECB mode but uses the result of previously encrypted blocks to affect the result of the next ciphertext block. To initialise the process an n-bit initialisation vector (IV) is used.

ALGORITHM: **CBC mode**:
INPUT: k-bit key K. n-bit IV. Plaintext message x split into n-bit blocks $x_1, \ldots, x_t$.
OUTPUT: Ciphertext message c formed by the concatenation of n-bit blocks $c_1, \ldots, c_t$.

Encryption $c_0 \leftarrow IV$. For $1 \leq j \leq t$ let $c_j \leftarrow E_K(c_{j-1} \oplus x_j)$.

Decryption $c_0 \leftarrow IV$. For $1 \leq j \leq t$ let $x_j \leftarrow c_{j-1} \oplus D_K(c_j)$.

□

The use of the IV and the chaining mechanism overcomes the main disadvantages of ECB. The ciphertext blocks can not be manipulated so as to affect the ordering of the plaintext because the ciphertext block c_j depends on all the previous blocks. Furthermore, if the IV is changed for each message, block replay is impossible (even though the IV is usually publicly known).

Cipher-Feedback mode

ECB and CBC modes operate on n-bit blocks and can not proceed until n bits of message data are available for encryption. Many applications, however, may want to operate on r-bit blocks, $1 \leq r \leq n$, and transmit the encrypted data as soon as r-bits are available. Often $r = 8$ which is the length of a single byte, and we may even use $r = 1$ and transmit a single bit at a time. Cipher-Feedback (CFB) mode provides a method for working with r-bit message blocks when using an n-bit block cipher. In other respects CFB mode shares many properties with CBC mode.

ALGORITHM: **CFB mode**:
INPUT: k-bit key K. n-bit IV. Plaintext message x split into r-bit blocks $x_1, \ldots, x_u$.
OUTPUT: Ciphertext message c formed by the concatenation of r-bit blocks $c_1, \ldots, c_u$.

Encryption $I_1 \leftarrow IV$. For $1 \leq j \leq u$ we calculate:

1. $O_j \leftarrow E_K(I_j)$.
2. $t_j \leftarrow$ the r least significant bits of O_j.
3. $c_j \leftarrow x_j \oplus t_j$.
4. $I_{j+1} \leftarrow 2^r \cdot I_j \bmod 2^n$.

Decryption $I_1 \leftarrow IV$. For $1 \leq j \leq u$ we calculate: $x_j \leftarrow c_j \oplus t_j$, where t_j, O_j and I_j are computed as for encryption.

□

Output-Feedback mode

Output-Feedback (OFB) is a slight variation on CFB mode. The only difference is in how the I_j values are updated: $I_{j+1} \leftarrow O_j$. As a result the t_j values, which can be viewed as a key-stream, are independent of the plaintext and thus can be precomputed (for a given key and initialisation vector pair). OFB resembles a one-time-pad system.

The advantage of OFB mode over CFB mode is that errors in the transmitted ciphertext are not propagated throughout the reconstructed plaintext. In CFB mode the key-stream is dependent on the preceding ciphertext blocks and if a block, c_j, contains an error then the decipherment of a number of the subsequent blocks will essentially be totally random. With OFB mode an error in block c_j will only affect the decipherment of c_j and in particular only the bits of the recovered plaintext block that correspond to the erroneous bits of c_j will be incorrect.

4.1.2 Block Cipher Algorithms

There are a number of block cipher algorithms that have been developed and used in practise. Below is a list of some of the well-known block ciphers:

Lucifer: Lucifer was developed in the early 1970s and was the result of research at IBM lead by H.Feistel and W.Tuchman. Lucifer is an iterated block cipher that derives its strength by iterating through a number of rounds during which a weak cryptographic function is repeatedly applied. Although Lucifer is the direct predecessor of DES, it operates on 128-bit blocks and uses a 128-bit key (DES uses a 56-bit key and 64-bit blocks). For this reason many believe that Lucifer is stronger than DES and that DES was intentionally weakened by its designers. However, using an extension of differential cryptanalysis, based on so-called conditional characteristics (key-dependent characteristics), Biham and Ben-Aroya show that an attack on Lucifer requires only 2^{36} complexity and chosen plaintexts to find most keys [5].

DES: Despite the initial controversy over the NSA's role in the design, DES is probably the most widely used block cipher. Many of the cryptanalytic technique that have been developed for analysing block ciphers have emerged as a result of studying DES. Furthermore, many of the existing block ciphers have borrowed ideas and design principles from DES.

FEAL: The design of FEAL was published in 1987. FEAL uses a 64-bit key and operates on 64-bit blocks. Its structure is very similar to DES; in fact the designers were hoping to create a DES-like algorithm with a stronger round function. However, the reality it that FEAL has been independently shown to be very weak using a variety of cryptanalytic attacks. The most devastating was the use of differential-linear cryptanalysis which can break 8-round FEAL using only 12 chosen plaintexts.

IDEA: X.Lai and J.Massey developed a cipher called Proposed Encryption Standard (PES) [26]. Further cryptanalysis of PES motivated a minor change to the permutation of the sub-blocks in between rounds and the modified algorithm was called Improved PES (IPES). Ultimately IPES became known as International Data Encryption Standard (IDEA). IDEA is an important algorithm because of the impressive theoretical foundations that were used to derive a sound design that was resistant to differential cryptanalysis and other techniques.

In recent years many other algorithms have been designed: MARS, RC6, Rijndael, Serpent and Twofish. These algorithms incorporate many designs that make them resistant to cryptanalytic attacks such as differential cryptanalysis, linear cryptanalysis, related-key attacks, truncated differentials and finding weak keys.

Since DES is well known and extensively studied, the next section will provide a detailed description of DES (see Appendix A for the details of the implementation of DES in C++). We will then introduce differential cryptanalysis and linear cryptanalysis and show how they can be applied to breaking DES (see Appendix B for the details of the C++ implementation of differential cryptanalysis of DES).

4.2 DES Algorithm

The Data Encryption Standard[2] algorithm was developed at IBM. IBM had first developed Lucifer and then developed an improved version. Before submitting this improved version as a candidate for the proposed Data Encryption Standard, IBM conferred with the U.S. National Security Agency (NSA) to strengthen it. This resulted in improvements to the S-boxes (substitution boxes) as well as a reduction of the key length from 64 bits to 56 bits plus 8 parity bits. This final version was then passed onto the U.S. National Bureau of Standards (NBS). The U.S. NBS received other candidates but IBM's proposal was the only viable one. The U.S. NBS published this proposal in 1975, but it was only adopted as the standard in 1977 [1]. In the interim the U.S. NBS had to dispel may accusations, made by non-governmental cryptologists, that the U.S. NSA had inserted a "trap door" or deliberately weakened the algorithm [20, page 980].

The DES algorithm is a block cipher operating on 64-bit blocks. The key is also 64 bits, but 8 bits are parity bits so the effective length is only 56 bits.

The DES algorithm falls into a class of algorithms known as Feistel networks [29, page 347]. In Feistel networks the plaintext block is divided into two halves of equal length: L and R. L consists of bits $1 \ldots 28$ and R consists of bits $29 \ldots 56$. An iterated block cipher is then defined where the input to the $(i+1)$th round is determined by the output of the previous round:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i), \end{aligned}$$

where K_i is the subkey for the ith round, f is an arbitrary round function and $1 \leq i \leq n$ where n is the number of rounds present in the Feistel network. Figure 4.1 shows the DES Feistel network. Iterated block ciphers are special cases of product ciphers where multiple functions are composed to form a product of functions – with an iterated cipher the same function is composed with itself multiple times.

The Feistel network provides a mechanism by which a relatively simple function can be used to construct a complex resultant function. The simple round function does not, by itself, provide

[2] As of 2000 the U.S. has supplanted DES by adopting the Rijndael algorithm as the Advanced Encryption Standard.

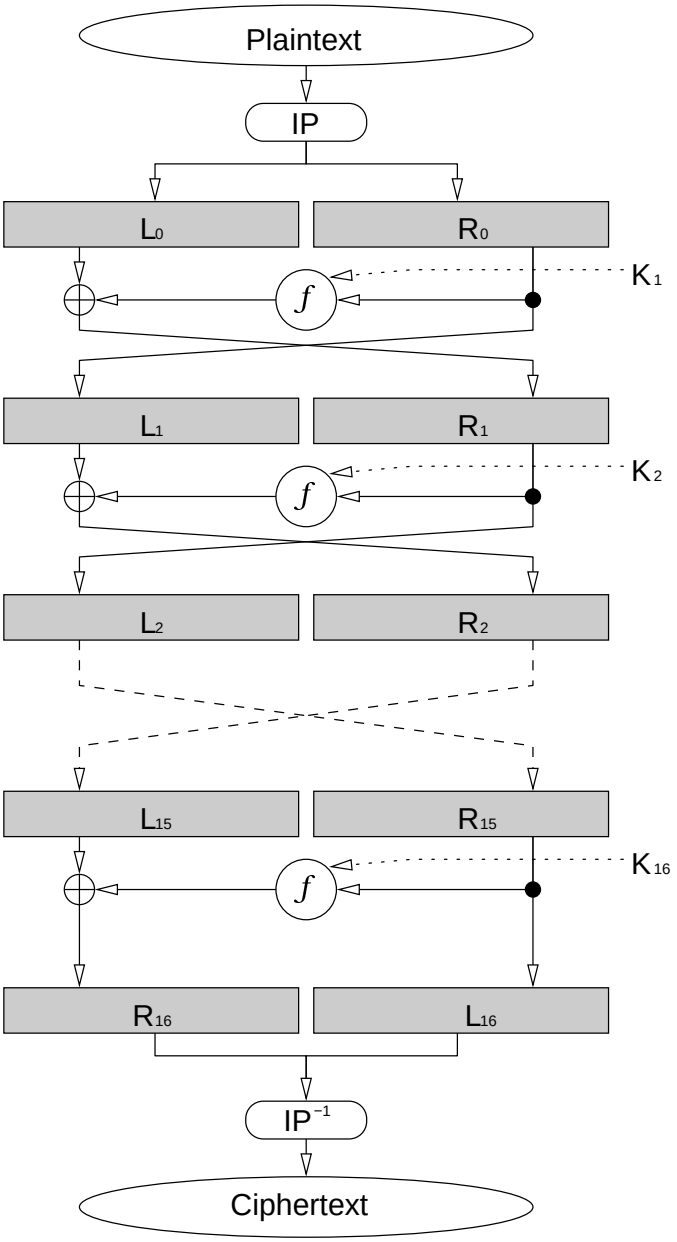


Figure 4.1: DES Feistel Network

sufficient cryptographic strength and could easily be broken. However, the repeated iteration and the use of a different subkey at each round works together to provide a complete encryption routine that is difficult to break.

An important property of Feistel networks is that the algorithm is invertible and thus can be used as a cipher (since both encryption and decryption can be defined). This is independent of whether or not f is invertible.

DES uses 16 rounds. Prior to the 16 iterations of the round function, f, an initial permutation, IP, is applied to the 64-bit plaintext block. After the 16 rounds have been applied the inverse, IP^{-1}, of the initial permutation is applied to the 64-bit block produced by the Feistel network.

Each entry of the permutation table refers to the position of the bit in the input data that will be copied into the output data at the position corresponding to that of the table entry. Thus the 58th bit of the plaintext will, as a result of the IP permutation, be placed into the 1st bit of the output, while the 50th bit will be placed in 2nd position. This format is used for all the other permutations that DES uses, i.e. P, E, PC1 and PC2.

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

Below is a description of the round function and how the S-boxes are used. This is followed by a description of the key schedule algorithm that produces the subkey used during each round.

4.2.1 DES Round Function

The DES round takes as input a 32-bit data block, R, and a 48-bit subkey block, K. The round function produces a 32-bit resultant block $f(R, K)$. The main components are an expansion permutation, E, the S-boxes and a final permutation P. The input data is expanded using E and the result is XORed with the subkey block. The result of the XOR is passed through the S-boxes and then operated on by P. Figure 4.2 shows how these components fit together to form the round function.

Experience has shown that to improve the security of a cipher, the algorithm should incorporate mechanisms for confusion and diffusion. These are techniques for obscuring the redundancies in a plaintext message. Confusion obscures the relationship between the plaintext and the ciphertext, thus making it more difficult to find redundancies and statistical patterns within the ciphertext. The easiest way to achieve this is via substitution. Diffusion dissipates the redundancy of the plaintext by spreading it out over the ciphertext, thus making it more difficult for a cryptanalyst to uncover the redundancies. The simplest way to introduce diffusion is via permutation.

The DES round function embodies both of these techniques. The permutation, P, provides diffusion (this is clear from the design criteria, see Table 4.1) whereas the S-boxes provide confusion (see §4.2.2).

The subkeys are generated by a key schedule algorithm that returns subsets of the input key (see §4.2.3).

- The 4 output bits from each S-box in a given round are distributed so that 2 of them affect the middle-bits of S-boxes in the next round and the other 2 affect end bits.
- The 4 output bits from each S-box affect 6 different S-boxes, yet no 2 affect the same S-box.
- If the output bit from one S-box affects the middle bit of another S-box, then an output bit from that S-box cannot affect the middle bit of the first S-box.

Table 4.1: P-Box Design Criteria

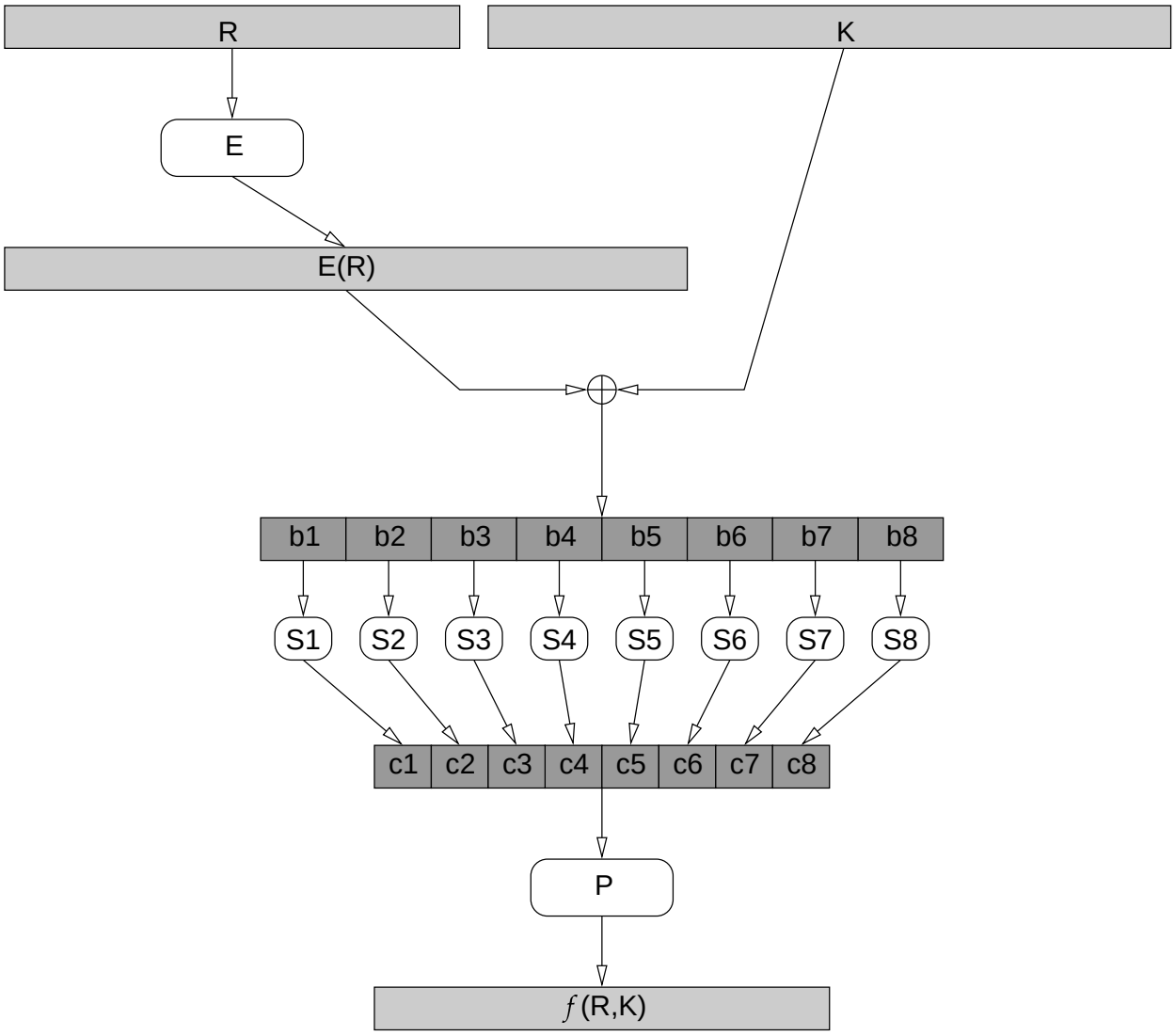


Figure 4.2: DES Round Function

$$P = \begin{bmatrix} 9 & 17 & 23 & 31 \\ 13 & 28 & 2 & 18 \\ 24 & 16 & 30 & 6 \\ 26 & 20 & 10 & 1 \\ 8 & 14 & 25 & 3 \\ 4 & 29 & 11 & 19 \\ 32 & 12 & 22 & 7 \\ 5 & 27 & 15 & 21 \end{bmatrix} \quad E = \begin{bmatrix} 32 & 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 6 & 7 & 8 & 9 \\ 8 & 9 & 10 & 11 & 12 & 13 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 16 & 17 & 18 & 19 & 20 & 21 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 24 & 25 & 26 & 27 & 28 & 29 \\ 28 & 29 & 30 & 31 & 32 & 1 \end{bmatrix}$$

4.2.2 DES S-Boxes

With respect to the strength of DES against cryptanalysis, the S-boxes are by far the most critical component. An S-box is a substitution that maps m-bit inputs to n-bit outputs.

DES S-boxes map 6-bit inputs to 4-bit outputs. The S-boxes are arranged in a table with 4 rows and 16 columns. If the input is $b_1b_2b_3b_4b_5b_6$ then $b1$ and b_6 are combined to from a 2-bit number in the range $0 \ldots 3$, and $b2 \ldots b_5$ are combined to form a 4-bit number in the range $0 \ldots 15$. These two numbers then act as an index to the S-box table by indicating the row and column respectively. The output (substitution) produced by the S-box is the value of entry indicated. The S-box entries are in the range $0 \ldots 15$ and thus represent all possible 4-bit output values. See Table 4.2 for a list of the DES S-Boxes and see Figure 4.3 for an example of using the S-Box tables.

S-Box Example
If the input to S-Box 1 was:

$$b_1b_2b_3b_4b_5b_6 = 1\ 1\ 1\ 0\ 1\ 0$$

the row would be:

$$b_1b_6 = 1\ 0 = 2$$

and the column would be:

$$b_2b_3b_4b_5 = 1\ 1\ 0\ 1 = 13.$$

Therefore, as seen from the S-Box definitions in Table 4.2, the output would be:

$$10 = 1\ 0\ 1\ 0.$$

Figure 4.3: DES S-Box Example

The S-boxes provide the only non-linear component in the DES algorithm, whereas all the other steps are linear and thus easy to analyse.

Since the S-boxes are absolutely critical to the security of the algorithm, a lot of effort has been applied to the theory of S-box design. The result of this research is split into two basic camps:

- S-boxes should be designed according to certain criteria.
- S-boxes should be chosen at random.

Table 4.2: S-Boxes

S-Box S1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S-Box S2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S-Box S3															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S-Box S4															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S-Box S5															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S-Box S6															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S-Box S7															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S-Box S8															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Naïvely one might think that a criteria based design would be better but both methods have compelling arguments.

The criteria oriented design can be either based on strict mathematical principles or it can be performed "by hand" in a more intuitive manner. When based on mathematical principles one usually implements a search algorithm that tests multiple candidates to see if they match all the criteria.

When using the criteria based approach one can ensure that the S-boxes are immune to all know cryptanalytic attacks. Thus the S-boxes can be chosen to reduce potential linearities which results in an algorithm that will be resistant to linear cryptanalysis. If, simultaneously, the S-boxes can be chosen such that the probability of distinct differences are similar and small, then one can also instill an immunity to differential cryptanalysis. One way to make S-boxes immune to differential cryptanalysis is to increase the size of the S-box. However, if only the number of output bits is increased then the likelihood of a linearity occurring increases and thus linear cryptanalysis would be more effective [6]. Similarly, linear cryptanalysis can be thwarted by increasing the randomness of the S-boxes, but random S-boxes are in general not as effective against differential cryptanalysis as carefully chosen S-boxes. Thus immunity to different attacks can often create conflicting requirements.

It can not be ensured that carefully constructed S-boxes will be safe against as yet unknown attacks. Furthermore, because of the computational difficulty of creating criteria based S-boxes, it is not feasible to repeatedly change the S-boxes. Thus criteria based S-boxes remain static and can be analysed at leisure by cryptanalysts.

Random S-boxes may not be as resistant to known attacks as selected S-boxes, but are more likely to be resistant to unknown attacks. Also, since random S-boxes are computationally easy to construct it is possible to change the S-box contents based on the encryption key. The S-boxes become a moving target for the cryptanalyst and new techniques will need to be developed to analyse S-box schedules.

4.2.3 DES Key Schedule

The key schedule takes the 56-bit key (64-bit key with the parity bits removed by discarding every 8th bit) and produces a sequence of 16 48-bit subkeys, $K_1 \dots K_{16}$, which are created by selecting particular key bits as described below. Each subkey is used in the corresponding round of the Feistel network.

The key schedule is performed by applying the following steps (see Figure 4.4):

1. Apply the fixed permutation, PC1, which produces a 56-bit permuted key.

2. Split the 56-bit permuted key into two 28-bit blocks, C_0 and D_0.

3. For each $1 \leq i \leq 16$ apply the shift scheme LS_i to compute

$$\begin{aligned} C_i &= LS_i(C_{i-1}) \\ D_i &= LS_i(D_{i-1}), \end{aligned}$$

   where LS_i represents a cyclic shift to the left (shift from the least significant bit towards the most significant bit, bit 1 shifts to bit 2) of either one or two positions, depending on the value of i: shift by one position if $i = 1, 2, 9$ or 16, otherwise shift by two positions.

4. For each $1 \leq i \leq 16$ apply PC2 to the 56-bit data block C_iD_i to compute K_i.

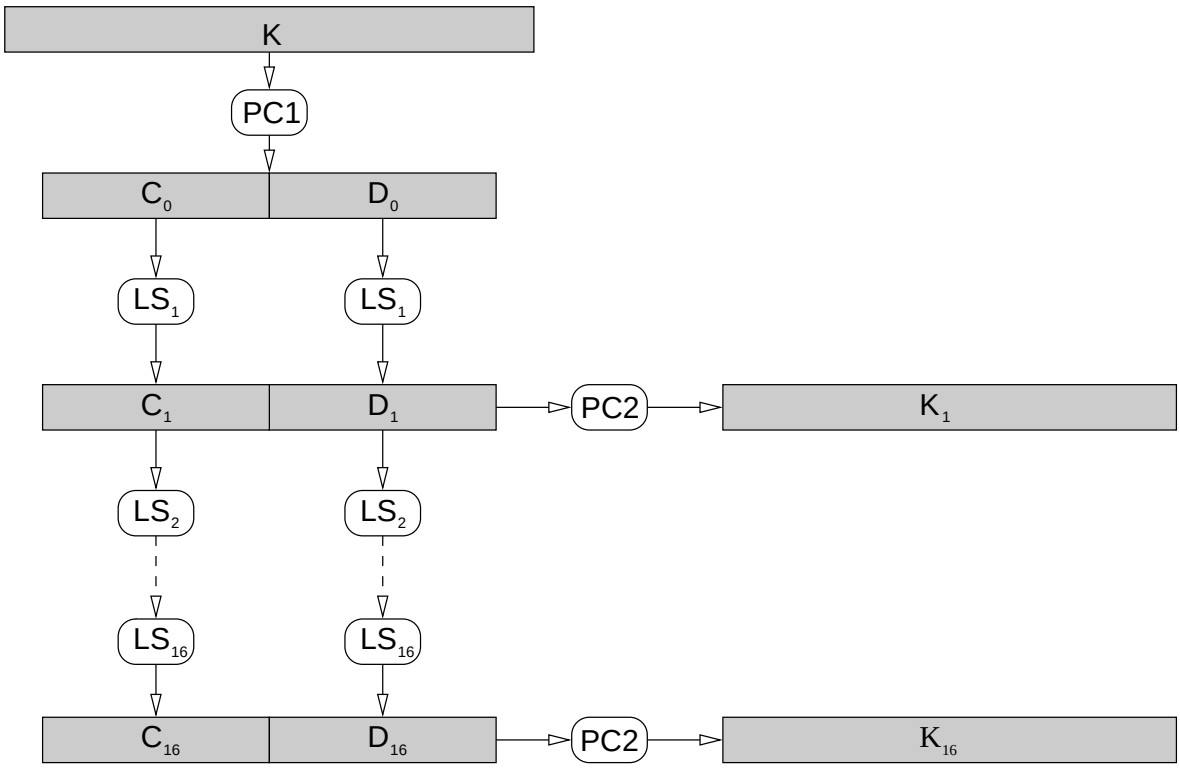


Figure 4.4: DES Key Schedule

$$\text{PC1} = \begin{bmatrix} 57 & 49 & 41 & 33 & 25 & 17 & 9 \\ 1 & 58 & 50 & 42 & 34 & 26 & 18 \\ 10 & 2 & 59 & 51 & 43 & 35 & 27 \\ 19 & 11 & 3 & 60 & 52 & 44 & 36 \\ 63 & 55 & 47 & 39 & 31 & 23 & 15 \\ 7 & 62 & 54 & 46 & 38 & 30 & 22 \\ 14 & 6 & 61 & 53 & 45 & 37 & 29 \\ 21 & 13 & 5 & 28 & 20 & 12 & 4 \end{bmatrix} \quad \text{PC2} = \begin{bmatrix} 14 & 17 & 11 & 24 & 1 & 5 \\ 3 & 28 & 15 & 6 & 21 & 10 \\ 23 & 19 & 12 & 4 & 26 & 8 \\ 16 & 7 & 27 & 20 & 13 & 2 \\ 41 & 52 & 31 & 37 & 47 & 55 \\ 30 & 40 & 51 & 45 & 33 & 48 \\ 44 & 49 & 39 & 56 & 34 & 53 \\ 46 & 42 & 50 & 36 & 29 & 32 \end{bmatrix}$$

As a result of the key schedule a different subset of key bits is used in each subkey. Each bit is used in approximately 14 of the 16 subkeys (not all bits are used an equal number of times).

4.2.4 DES Properties

Group Structure of DES

To obtain greater cryptographic strength from a block cipher implementors may decide to apply a given encryption algorithm multiple times. This is the case with Triple-DES which applies DES iteratively three times, using a different key at each stage.

However, this scheme does not necessarily improve security. A block cipher defines permutation on the plaintext block, each key selecting a different permutation. DES can have 2^{56} distinct keys and thus defines at most 2^{56} permutations out the set of $2^{64}!$ possible permutations. By using multiple encryption we are hoping to be able to tap a larger set of permutation, but this is only true if the DES operation does not have certain algebraic structures.

If DES were closed, then for any K_1 and K_2, there would always exist a K_3 such that

$$E_{K_2}(E_{K_1}(P)) = E_{K_3}(P).$$

That is, the DES encryption operation would form a group, and multiple encryption would still define a permutation from the set defined using the 2^{56} keys.

Thus, it was important to prove whether or not DES encryption was a closed operation, and if it is not closed then one needs to show that the subgroup generated is large enough to provide enhanced security.

The first publicly know proof that DES is not a group came from Campbell and Wiener in 1992 [12]. They extended many previous efforts so as to provide probabilistic evidence that DES is not a group as well as explicitly calculating a lower bound for the number of distinct permutations that can be reached via multiple encryption. The lower bound shows that the subgroup generated by the set of DES permutations contains more than 10^{2499} elements, which is greater than the number of DES permutations and thus DES is not closed.

To find the lower bound Campbell and Wiener consider the permutation resulting from the composition of encryption using a key of all 0s and then using a key of all 1s. This permutation produced by DES is known to have short cycles (the output repeats after about 2^{32} iterations). Such short cycle lengths make it feasible to measure exact cycle lengths when the permutation is applied to different starting messages. The cycle length for each message must, however, divide the order of the subgroup generated by the particular permutation, which must in turn divide the order of the subgroup generated by all DES permutations. Thus, by finding the cycle lengths for a number of distinct starting messages and then calculating the least common multiple, one is able to obtain a lower bound for the size of subgroup of permutations generated by DES.

Weak Keys (With parity bits)				Key Value	
0101	0101	0101	0101	0000000	0000000
0101	0101	0101	0101	0000000	FFFFFFF
0101	0101	0101	0101	FFFFFFF	0000000
0101	0101	0101	0101	FFFFFFF	FFFFFFF

Table 4.3: DES Weak Keys

01FE	01FE	01FE	01FE	and	FE01	FE01	FE01	FE01
1FE0	1FE0	0EF1	0EF1	and	E01F	E01F	F10E	F10E
01E0	01E0	01F1	01F1	and	E001	E001	F101	F101
1FFE	1FFE	0EFE	0EFE	and	FE1F	FE1F	FE0E	FE0E
011F	011F	010E	010E	and	1F01	1F01	0E01	0E01
E0FE	E0FE	F1FE	F1FE	and	FEE0	FEE0	FEF1	FEF1

Table 4.4: DES Semi-Weak Key Pairs

Weak Keys

When cryptanalysing a cipher such as DES it is important not only to consider the round function, but also to consider the key schedule. After the initial permutation the key schedule for DES splits the key into two halves and then independently shifts the two halves from which the subkeys are created. By looking at how the key schedule works it is possible to construct keys that produce the same subkey for each round. These are known as weak keys. DES has four weak keys, see Table 4.3. Below is a more general definition of weak keys.

Definition 4.2 (Weak Keys) *Let* $E_K(x)$ *denote encryption of the message* x *by the cipher* E *using the key* K. K *is said to be a weak key if* $E_K(E_K(x)) = x$.

Due to the nature of the DES key schedule it is also possible to find pairs of keys such that a plaintext message can be encrypted with the first key and then decrypted with the second key – these are known as semi-weak keys. DES has six pairs of semi-weak keys, see Table 4.4. Semi-weak keys are defined as follows:

Definition 4.3 *Let* $E_K(x)$ *denote encryption of the message* x *by the cipher* E *using the key* K. (K_1, K_2) *is said to be a semi-weak key pair if* $E_{K_1}(E_{K_2}(x)) = x$.

Complement Property

Let $\overline{x}$ denote the bitwise complement of x. Then, for DES encryption, $E_K(P) = C$ implies that $E_{\overline{K}}(\overline{P}) = \overline{C}$. This is easy to understand by first noting that for any binary strings x and y:

- $\overline{x} \oplus \overline{y} = x \oplus y$ and
- $\overline{x} \oplus y = \overline{x \oplus y}$.

Now, the input to the S-boxes is $E(R_i) \oplus K_i = \overline{E(R_i)} \oplus \overline{K_i} = E(\overline{R_i}) \oplus \overline{K_i}$, and thus we have, for the output of the round function f: $f(R_i, K_i) = f(\overline{R_i}, \overline{K_i})$. Similarly, for the right hand side for the next round we have: $\overline{R_{i+1}} = \overline{L_i \oplus f(R_i, K_i)} = \overline{L_i} \oplus f(\overline{R_i}, \overline{K_i})$.

The complement property implies an attack on DES that required only 2^{55} operations, rather than the full 2^{56} operations of an exhaustive search (although on average a brute force search will only need to consider half of the keys).

4.3 Differential Cryptanalysis

4.3.1 Overview

Differential cryptanalysis was the most successful attack on reduced round DES when it first introduced to the public in 1990 by Eli Biham and Adi Shamir [7]. At that point the basic method did not scale to full 16-round DES. In 1992 Eli Biham and Adi Shamir improved upon their first method and presented the first publicly known attack which is capable of breaking full 16-round DES in less than the 2^{55} complexity of an exhaustive search [8]. This improved version can break full 16-round DES in 2^{37} time and negligible space by analysing 2^{36} ciphertexts obtained from a larger pool of 2^{47} chosen plaintexts.

Differential cryptanalysis works by finding anomalies in the distribution of DES outputs. Some properties of the outputs have a probability that is higher than if the outputs occurred randomly. These anomalies can then be used to suggest which keys were used during the encryption.

More specifically, if one were to analyse the distribution of ciphertext outputs as produced by chosen plaintext inputs, one would not be able to find any useful correlation to the keys used for the encryption. However, if, for a fixed key, one considers pairs of inputs that share certain properties then the probability that the corresponding pairs of output share related properties is higher than one would expect from a random data source. Due to the correlation produced one can obtain information about the key that was used during the encryption.

By searching for pairs of plaintext inputs that exhibit the desired characteristics for the plaintext and ciphertext pairs, and then analysing the results, differential cryptanalysis is able to recover the encryption key using less operations than an exhaustive search.

4.3.2 Application to DES

Differential cryptanalysis is a chosen plaintext attack that is capable of recovering the encryption key. The attack relies on two phases:

- the first phase is an information gathering phase which collects data that will be useful for the attack,
- the second phase uses the collected data to assign probabilities to a collection of potential keys, and then subsequently to test the most likely keys for validity.

The actual implementation of differential cryptanalysis usually blurs the distinction between the various phases. This occurs because a large amount of information is shared between the phases and time/space optimisations can be put in place by combining the calculations.

Subkey Recovery

To understand how differential cryptanalysis recovers keys, we first define what is meant by a *difference* of two bit strings. We then show how the knowledge of differences of bit strings, before and after passing through the S-boxes, can be used to tag certain keys as possible candidates for the actual encryption key used.

We adopt the following notation: if we denote by B a bit string that occurs at some place in the encryption algorithm, then we use B^* to denote a second instance that occurs at the same place but during a separate calculation; B' is used to denote the difference between the two bit strings and is defined by $B' = B \oplus B^*$ (that is the prime markings ($'$) indicate the XOR of two bit strings)[3].

Definition 4.4 (Input/Output Difference) *Let S_j, $1 \leq j \leq 8$, be a particular S-box. Let B_j and B_j^* be two 6-bit inputs to the S-box. Then $B_j' = B_j \oplus B_j^*$ is referred to as the input difference of S_j. $S_j(B_j) \oplus S_j(B_j^*)$ is referred to as the output difference of S_j.*

Definition 4.5 ($\Delta(B_j')$) *Given $B_j' \in (\mathbb{Z}_2)^6$, we define $\Delta(B_j')$ to be the set of ordered pairs (B_j, B_j^*) with a difference equal to B_j':*

$$\Delta(B_j') = \left\{ (B_j, B_j^*) \in (\mathbb{Z}_2)^6 \times (\mathbb{Z}_2)^6 \,\middle|\, B_j \oplus B_j^* = B_j' \right\}.$$

$\Delta(B_j')$ is a set consisting of $2^6 = 64$ ordered pairs. The contents of the set can be calculated as follows:

$$\Delta(B_j') = \left\{ (B_j, B_j \oplus B_j') \,\middle|\, B_j \in (\mathbb{Z}_2)^6 \right\}.$$

For each pair in $\Delta(B_j')$ we can calculate the corresponding output pair of the S-box S_j, from which we can find the output difference. If we perform this calculation for each possible input difference we can then tabulate the resultant distribution, see Table 4.5 for a partial listing of the distribution for S-box 1. The non-uniformity of this distribution for each of the S-boxes enables one to create methods by which to detect more probable keys and thus form the basis for the differential cryptanalytic attack.

Clearly, if we know the input to the S-boxes, B, and the expansion of the input to the round, E, then we can find the corresponding subkey: $K = B \oplus E$. This is, however, not usually possible. But using input differences and back-propagation of the output differences it is possible to calculate a set of possible subkeys.

Given a pair of inputs, R and R^*, to the round function for a particular round we can use the expansion permutation to find, E and E^*. These, in turn, can be used to determine the input difference, B' (see Figure 4.2):

$$B' = B \oplus B^* = (E \oplus K) \oplus (E^* \oplus K) = E \oplus E^* = E'.$$

K is the subkey used in the given round.

Furthermore, given the outputs, C and C^*, of the S-boxes for a particular round we can find the output difference, C'. C and C^* can be calculated from the round function outputs by applying the inverse of the permutation, P.

Let $B = B_1B_2 \ldots B_8$ where each B_j is a 6-bit string corresponding to the input to the jth S-box, S_j. Let $C = C_1C_2 \ldots C_8$ where each C_j is a 4-bit string corresponding to the output of the jth S-box. B_j' and C_j' respectively denote the input and output difference of S_j.

Based on the information used to construct the tabulated distribution we see that for a given input difference/output difference pair (B', C') there is a subset of all possible pairs (B, B^*) that would produce the correct values for the differences. Let this set be denoted by $I_j(B_j', C_j')$:

$$I_j(B_j', C_j') = \left\{ (B_j, B_j^*) \in \Delta(B_j') \,\middle|\, S_j(B_j) \oplus S_j(B_j^*) = C_j' \right\}.$$

[3]The first attacks using differential cryptanalysis used XOR as the difference. It is, however, possible to define different types of differences, and this may be necessary when attacking algorithms that are very different to DES

Input XOR	Output XOR															
	0_{16}	1_{16}	2_{16}	3_{16}	4_{16}	5_{16}	6_{16}	7_{16}	8_{16}	9_{16}	A_{16}	B_{16}	C_{16}	D_{16}	E_{16}	F_{16}
0_{16}	64	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1_{16}	0	0	0	6	0	2	4	4	0	10	12	4	10	6	2	4
2_{16}	0	0	0	8	0	4	4	4	0	6	8	6	12	6	4	2
3_{16}	14	4	2	2	10	6	4	2	6	4	4	0	2	2	2	0
4_{16}	0	0	0	6	0	10	10	6	0	4	6	4	2	8	6	2
5_{16}	4	8	6	2	2	4	4	2	0	4	4	0	12	2	4	6
6_{16}	0	4	2	4	8	2	6	2	8	4	4	2	4	2	0	12
7_{16}	2	4	10	4	0	4	8	4	2	4	8	2	2	2	4	4
⋮									⋮							
C_{16}	0	0	0	8	0	6	6	0	0	6	6	4	6	6	14	2
⋮									⋮							
38_{16}	0	6	2	2	2	0	2	2	4	6	4	4	4	6	10	10
39_{16}	6	2	2	4	12	6	4	8	4	0	2	4	2	4	4	0
$3A_{16}$	6	4	6	4	6	8	0	6	2	2	6	2	2	6	4	0
$3B_{16}$	2	6	4	0	0	2	4	6	4	6	8	6	4	4	6	2
$3C_{16}$	0	10	4	0	12	0	4	2	6	0	4	12	4	4	2	0
$3D_{16}$	0	8	6	2	2	6	0	8	4	4	0	4	0	12	4	4
$3E_{16}$	4	8	2	2	2	4	4	14	4	2	0	2	0	8	4	4
$3F_{16}$	4	8	4	2	4	0	2	4	4	2	4	8	8	6	2	2

Table 4.5: XOR Difference Distribution for S-Box 1

By combining each entry of $I_j(B'_j, C'_j)$ with the known E_j we can then find a set of possible subkey bits, $T_j(E_j, E^*_j, C'_j)$:

$$T_j(E_j, E^*_j, C'_j) = \left\{ K_j \,\middle|\, K_j = B_j \oplus E_j,\ (B_j, B^*_j) \in I_j(E'_j, C'_j) \right\}.$$

Key Recovery

The result of all possible combinations of the subkey bits produces a set of suggested subkeys, $T(E, E^*, C')$:

$$T(E, E^*, C') = \left\{ K \,\middle|\, K = K_1 K_2 \dots K_8,\ K_j \in T_j(E_j, E^*_j, C'_j), 1 \le j \le 8 \right\}.$$

In some attacks it is not possible to know E_j, E^*_j and C'_j for each $j \in [1, 8]$, but rather only for each j in some subset of $\{1, \dots, 8\}$. In this case we could combine the known T_j to form a set of possible subkeys with less than 48 bits.

For each subkey in the set of possible subkeys we could perform an exhaustive search to attempt to find the remaining unknown key bits. However, even this search would usually be too large to be practical.

To constrain the search we could employ a strategy whereby we generate test sets for multiple triples (E_j, E^*_j, C'_j) and look at the intersection of all the suggested subkeys.

In implementations of differential cryptanalysis these triples are usually selected via a probabilistic algorithm that utilises characteristics and right-pairs (page 38). Thus, the set may contain triples that only suggest incorrect subkeys and hence the intersection may be empty. However, each

valid triple results from a right-pair and right-pairs will occur with the characteristic's probability. Furthermore, each valid triple will suggest the correct subkey and all other suggested subkeys (from right-pairs or wrong-pairs) will be approximately uniformly distributed. Consequently, the right subkey appears with the characteristic's probability (from right-pairs) as well as some other random occurrences (from wrong-pairs). If the number of triples tested is large enough then the right subkey will be the subkey that is suggested most often. We can then use this single subkey together with an exhaustive search to find the full encryption key.

For further details see the descriptions of the 3-round attack (page 40) and the 6-round attack (page 42).

Characteristics and Right Pairs

To obtain the input pair and output differences that can be used to recover the key we introduce the notion of characteristics and right-pairs. A characteristic is selected for its high probability, and once we have found a pair that matches the characteristic we can infer sufficient information about the internal state of the cipher to be able to produce a triple (E, E^*, C') that can be used to produce a set of suggested keys.

Definition 4.6 (Characteristic) *A characteristic is a couple* (α, β)*, where* α *and* β *are two bit strings equal in length to the block size of the cipher. Associated with a characteristic is a round length,* i*, and a probability,* ω*.* α *is the difference of a pair input plaintexts,* X *and* X^**, and* β *is a possible difference of the output,* Y_i *and* Y_i^**, resulting after* i *rounds in the Feistel network.* ω *is the conditional probability that* β *is the difference* Y_i' *of the ciphertext pair after* i *rounds given that the pair* (X, X^*) *has a difference* $X' = \alpha$ *(the plaintext inputs and the round subkeys are assumed to be chosen randomly from a uniform distribution).*

Characteristics are sometimes referred to as i-round differentials. The probability of a characteristic (α, β) could be written as $P(Y_i' = \beta | X' = \alpha)$ [26]. Figure 4.5 depicts a 3-round DES characteristic.

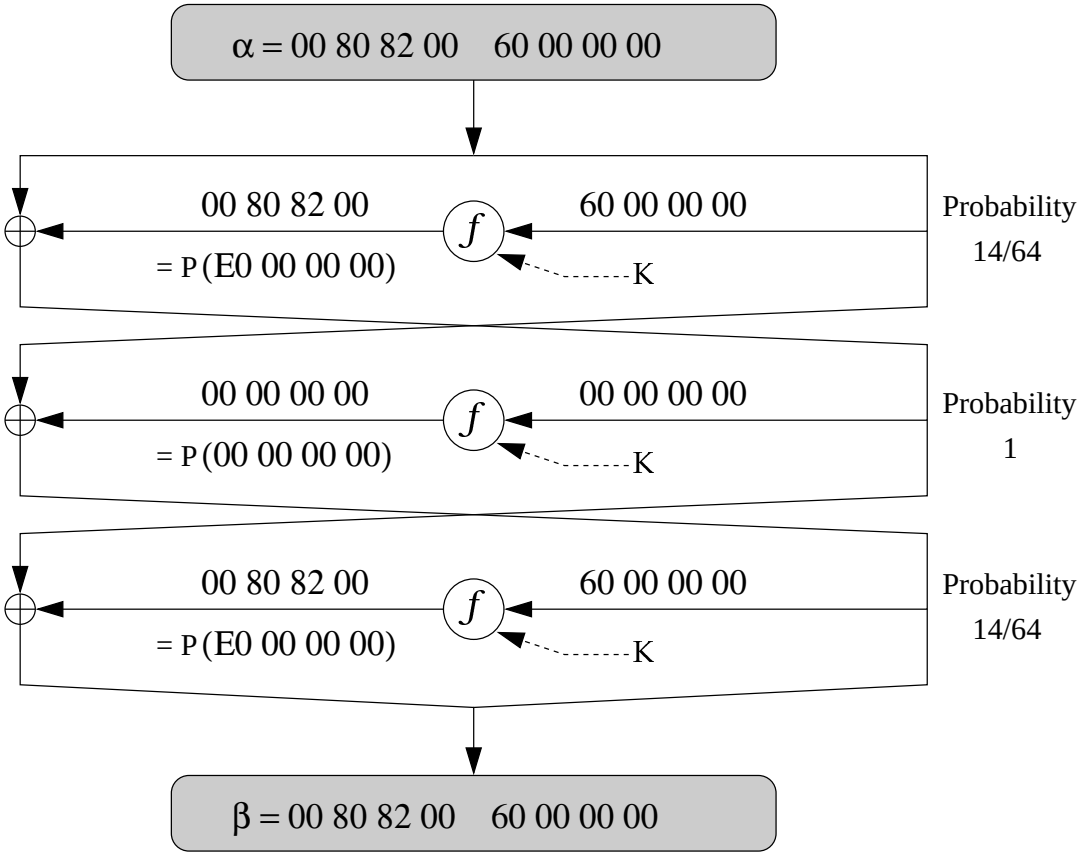
In what follows we will present a method for calculating the probability of a characteristic from the distribution for the S-box input and output differences. From this method it is easy to see that the non-uniformity of the distribution for input and output differences of S-boxes, will lead to large variations in the distribution of probabilities exhibited by various characteristics. Furthermore, the method for calculating the probability of a given characteristic can be easily adapted to find characteristics that will have a high probability.

To calculate the probability of a characteristic we consider each round individually. For the calculation we need to know the values of the S-box output differences. Although a characteristic is referenced by giving the input and output differences, each characteristic is usually constructed with the knowledge of the S-box input and output differences for each round. If (α, β) is an i-round characteristic then let $\delta^1, \ldots, \delta^i$ denote the output differences of the S-boxes after each round.

For each round we carry out the following steps:

1. Let r be the number of the round we are considering.
2. Let the left hand side of the input difference be L' and let the right hand side of the input difference be R'.
3. Calculate the result of applying the expansion permutation, E, to L'. As we have seen above this would be equal to B' since

$$B' = B \oplus B^* = E(R) \oplus E(R^*) = E(R \oplus R^*) = E(R').$$



A three round characteristic

$$(\alpha, \beta) = (00808200\ 60000000_{16}, 00808200\ 60000000_{16})$$

with probability $\frac{14}{64} \cdot 1 \cdot \frac{14}{64} \approx 0.048$.
The expansion of the right hand side, 60000000_{16}, before entering the S-boxes is

$$E(60000000_{16}) = 300000000000_{16}.$$

Therefore, the input difference to S-box 1 is 001100_2 which occurs 14 out 64 times when the output difference is 1110_2. This can be seen by looking at row $0C_{16}$ column E_{16} of Table 4.5. The input difference for each of the remaining S-boxes is 000000_2 which always produces an output difference of 0000_2.

Figure 4.5: DES 3-round Characteristic

4. For each S-box the corresponding input difference would be B'_j where $B' = B'_1 B'_2 \dots B'_8$. Similarly for each S-box the corresponding output difference would be δ^r_j where $\delta^r = \delta^r_1 \delta^r_2 \dots \delta^r_8$.

   By considering the pair (B'_j, δ^r_j) and consulting the distribution table for the jth S-box we can determine the likelihood of an input pair with input difference B'_j producing an output difference of δ^r_j. Let this be denoted by ω^r_j.

   EXAMPLE: considering the distribution for S-box 1 (Table 4.5 is a partial listing of the difference distribution for S-box 1) we see that 12 out of the 64 possible 6-bit input pairs with a difference of $3D_{16} = 111101_2$, would produce an output difference of $D_{16} = 1101_2$. □

   Thus, the probability of an input pair having an input difference of B' and producing an output difference of δ^r, would be given by the product of the probabilities calculated for each S-box. We denote this by ω^r:

$$\omega^r = \prod_{j=1}^{8} \omega^r_j.$$

5. To find the output difference of the round function we simply apply the permutation P to δ^r.

6. The input differences for the subsequent round can now be calculated as follows:

$$\begin{aligned} R' &:= L' \oplus P(\delta^r) \\ L' &:= R'. \end{aligned}$$

To find the probability of the characteristic we consider the effect of each round. Thus the probability of the complete characteristic is given by:

$$\omega = \prod_{r=1}^{i} \omega^r.$$

For the purposes of implementing differential cryptanalysis it is useful to introduce the concepts of a right-pair.

Definition 4.7 (Right-Pair) *An input pair (X, X^*) is said to be a right-pair for the characteristic (α, β) if $X' = \alpha$ and the $Y'_i = \beta$, where the characteristic is defined across i rounds and Y_i and Y^*_i are the outputs of the Feistel network after the corresponding rounds.*

In terms of the definition of right-pairs, the probability of a characteristic (α, β) is the probability that a randomly selected pair (X, X^*), such that $X \oplus X^* = \alpha$, is in-fact a right-pair for the given characteristic. Thus, strictly speaking the method for calculating the probability of a characteristic is an approximation, but the error is negligible (see Note 4.6).

Breaking 3-round DES

3-round DES is particularly simple and the data gathering does not require the full complexity of characteristics. However, the attack on 3-round DES is useful for providing a concrete demonstration of the methods for recovering the key from plaintext/ciphertext pairs.

Using the structure of the Feistel network and by choosing the input pairs to have a particular input difference we are able to construct test triples (E, E^*, C') for each of three chosen plaintexts.

NOTE: **Characteristic Probability Calculation**
The probability calculated for a single round is exact. However, the rounds are not necessarily independent since there may be additional pairs that meet the input and output criteria, even though the internal state does not match. Thus, the probability calculated using the product may be lower than the actual value. For this reason it is possible that an actual implementation of differential cryptanalysis may be more effective than predicted.

Note 4.6: Characteristic Probability Calculation

Firstly, by looking at Figure 4.1 we see that

$$\begin{aligned} R_3 &= L_2 \oplus f(R_2, K_3) \\ &= R_1 \oplus f(R_2, K_3) \\ &= L_0 \oplus f(R_0, K_1) \oplus f(R_2, K_3). \end{aligned}$$

Note that for ease of illustration we ignore the initial and final permutations, IP and IP^{-1}. Furthermore, we include the twist after the final round of encryption. Since the corresponding operations are invertible, they have no impact on the security of the algorithm.

We can then express the output difference of the right hand side as follows:

$$R'_3 = L'_0 \oplus f(R_0, K_1) \oplus f(R^*_0, K_1) \oplus f(R_2, K_3) \oplus f(R^*_2, K_3).$$

If we choose the plaintexts, L_0R_0 and $L^*_0R^*_0$, such that $R'_0 = R_0 \oplus R^*_0 = 00\ldots0_{16}$ then

$$f(R_2, K_3) = f(R^*_2, K_3).$$

Therefore,

$$R'_3 = L'_0 \oplus f(R_2, K_3) \oplus f(R^*_2, K_3).$$

If we use C and C' to denote the S-box outputs and recall that the round function output is found by applying the permutation, P, then we see that for the third round we have

$$\begin{aligned} P(C) &= f(R_2, K_3) \\ P(C^*) &= f(R^*_2, K_3). \end{aligned}$$

Hence, using the inverse of P we can find C':

$$\begin{aligned} C' &= P^{-1}(f(R_2, K_3) \oplus f(R^*_2, K_3)) \\ &= P^{-1}(R'_3 \oplus L'_0). \end{aligned}$$

Furthermore, $R_2 = L_3$ and $R^*_2 = L^*_3$. Thus, by applying the expansion permutation we can form the test triple $(E, E^*, C') = (E(L_3), E(L^*_3), P^{-1}(R'_3 \oplus L'_0))$.

Using the test triple we can form the test sets $T_j(E_j, E^*_j, C'_j)$ for each S-box, $1 \leq j \leq 8$. This is repeated for each of three chosen plaintext pairs.

We now count which subkey bits occur most often for each S-box. This is achieved by maintaining an array of 64 counters for each S-box. There is one counter for each string of 6 subkey bits. When finished we expect, for each S-box, a single counter to have a value of 3. We then combine the eight 6-bit strings to form a 48-bit subkey.

The remaining 8 bits of the 56-bit encryption key can be recovered via an exhaustive search.

In this way we are able to recover a 3-round DES encryption key by analysing only 3 pairs of chosen plaintext.

Depending on the particular choice of plaintexts it may occur that more than one of the 6-bit subkey strings is suggested three times. In this case the search has failed to find a unique 48-bit subkey. The easiest solution is to consider a new set of input plaintext pairs.

See Appendix B for a description of the C++ implementation of differential cryptanalysis of 3-round DES. Also listed are the results of one such run. This shows the test triples that are constructed, as well as the corresponding test sets containing the suggested 6-bit subkey strings.

Breaking 6-round DES

The attack on 6-round DES requires a more in-depth application of characteristics so as to be able to gather sufficiently useful data which can be used during the key recovery phase. The key recovery phase for the attack on 6-round DES is conceptually similar to the attack on 3-round DES, but due to storage constraints the method is modified.

We use a 3-round characteristic to find right-pairs that can be used to construct test triples (E, E^*, C). However, as we will see, only the parts of E and E^* that correspond to S-boxes S_2, S_5, S_6, S_7, S_8 will be known and thus we can only construct the set of suggested keys: $T_j(E_j, E_j^*, C_j')$ for $j = 2, 5, \ldots, 8$. Hence, we will only recover $5 \times 6 = 30$ bits of the subkey in round 6. To recover more key bits a second 3-round characteristic could be employed. The bits that are still not recovered can then be found using an exhaustive search.

We will describe how the test triples are found using the 3-round characteristic:

$$(\alpha, \beta) = (40080000\ 04000000_{16}, 04000000\ 40080000_{16}).$$

This characteristic has a probability of $\frac{1}{16}$.

From the structure of the Feistel network we have:

$$\begin{aligned} R_6 &= L_5 \oplus f(R_5, K_6) \\ &= R_4 \oplus f(R_5, K_6). \end{aligned}$$

Therefore

$$R_6' = R_4' \oplus f(R_5, K_6) \oplus f(R_5^*, K_6). \tag{4.1}$$

Similarly

$$R_4' = L_3' \oplus f(R_3, K_4) \oplus f(R_3^*, K_4). \tag{4.2}$$

If C' denotes the output difference of the S-boxes after round 6, then using (4.1) and applying the inverse of the permutation P we have

$$\begin{aligned} C' &= P^{-1}(f(R_5, K_6) \oplus f(R_5^*, K_6)) \\ &= P^{-1}(R_6' \oplus R_4') \\ &= P^{-1}(R_6') \oplus P^{-1}(R_4'). \end{aligned} \tag{4.3}$$

Let the input pair $(L_0R_0, L_0^*R_0^*)$ be a right-pair. Then $L_3' = 04000000_{16}$ and $R_3' = 40080000$ with a probability of $\frac{1}{16}$. Thus the right-hand side input difference for round 4 is 40080000_{16}, the expansion of which is

$$200050000000_{16} = 001000\ 000000\ 000001\ 010000\ 000000\ 000000\ 000000\ 000000_2.$$

Therefore the input difference for S-boxes S_2, S_5, S_6, S_7, S_8 is zero and thus the output differences of these S-boxes is also zero. If we let D' denote the output differences of the S-boxes in the 4th round then we have

$$D' = D'_1 D'_2 \dots D'_8 = P^{-1}(f(R_3, K_4) \oplus f(R^*_3, K_4)), \tag{4.4}$$

with $D'_j = 0000_2$ for $j = 2, 5, \dots, 8$. From the characteristic we know that $L'_3 = 04000000_{16}$. By combining this with (4.2), (4.3) and (4.4) we see that:

$$\begin{aligned} C' &= P^{-1}(R'_6) \oplus P^{-1}(L'_3 \oplus f(R_3, K_4) \oplus f(R^*_3, K_4)) \\ C' &= P^{-1}(R'_6) \oplus P^{-1}(L'_3) \oplus D' \\ C' &= P^{-1}(R'_6 \oplus 04000000_{16}) \oplus D'. \end{aligned}$$

Let $A' = A'_1 \dots A'_8 = P^{-1}(R'_6 \oplus 04000000_{16})$. Then, since $D'_2 = D'_5 = \dots = D'_8 = 0000_2$ we know that

$$C'_j = A'_j \text{ for } j = 2, 5, \dots, 8.$$

Hence we now know how to find the output differences of 5 of the S-boxes for round 6.

The inputs for the S-boxes in round 6 can be found from

$$E = E_1 E_2 \dots E_8 = E(R_5) = E(L_6)$$

and

$$E^* = E^*_1 E^*_2 \dots E^*_8 = E(R^*_5) = E(L^*_6).$$

Thus for each right-pair we can form the test triples (E_j, E^*_j, C'_j) for $j = 2, 5, \dots, 8$. From these test triples we can construct the corresponding sets of suggested subkey bits, $T_j(E_j, E^*_j, C'_j)$ for $j = 2, 5, \dots, 8$. We can then form the cross product of the 5 sets of suggested subkey bits to create a set of suggested 30-bit subkeys. This set would typically have approximately 15000 subkeys, but could contain in excess of 100000 subkeys.

We now need to gather right-pairs and use the suggested keys to find the actual encryption key. However, to be sure that a chosen pair is a right-pair for our 3-round characteristic, we would need to be able to check the output difference after the third round. Since we are attempting to break 6-round DES and we do not yet know the encryption key, this is clearly not possible.

Instead of explicitly gathering right-pairs, we recall that for our chosen 3-round characteristic a chosen plaintext pair with the correct input difference will have a 1 in 16 chance of being a right-pair. We then count how many times each subkey is suggested. Each right-pair will suggest the correct subkey once, as well as suggesting a collection of random subkeys. Each wrong-pair will suggest a random spread of subkeys. There are 2^{30} possible subkeys, relative to which only a small number of wrong subkeys is produced[4]. Thus, each wrong subkey is likely to be only counted 0 or 1 times. In contrast, the correct subkey will be counted approximately $\frac{1}{16}n$ times, where n is the number of input pairs tested. Thus by looking for a subkey that is counted approximately $\frac{1}{16}$ times we can find the correct subkey.

To perform the counting one could maintain a list of all suggested subkeys along with their respective counters. However, this would require a lot a storage space and would become infeasible with respect to the resources available to most computers, when one attempts to break DES with more rounds. To solve this we use the following algorithm that requires less space and time.

For each input plaintext pair we use the corresponding $T_j(E_j, E^*_j, C'_j)$ to construct the 64-bit vector, t_j, where the ith coordinate of t_j is set to 1 if the bit-string of length 6 that corresponds

[4] Typically about 200 plaintext pairs are considered when breaking 6-round DES. Therefore, the number of wrong-pairs would be approximately $200 \times 15000 = 3 \times 10^6$. In comparison $2^{30} \approx 10^9$.

to the binary representation of i, is contained in the set $T_j(E_j, E_j^*, C_j')$. All other coordinates of t_j are set to 0. We label the vectors produced by each plaintext pair as t_j^r for $j = 2, 5, \ldots, 8$ and $1 \leq r \leq n$.

To find the correct key bits we construct subsets $I \subseteq \{1, \ldots, n\}$ where I is said to be allowable if for each $j \in \{2, 5, 6, 7, 8\}$ there is at least one coordinate equal to $|I|$ in the vector

$$\sum_{r \in I} t_j^r.$$

If every pair indexed by I is a right-pair then I is allowable. Hence, we expect there to be a single allowable set of size approximately $\frac{1}{16}n$ which should suggest the correct subkey and no other. To find the allowable sets we can employ a recursive algorithm that generates all possible subsets of $\{1, \ldots, n\}$ and check which sets are allowable.

To further enhance the attack outlined above we can filter the chosen plaintext pairs by identifying and discarding a portion of the wrong-pairs. To perform the filtering we note that if an input pair generates a test set, $T_j(E_j, E_j^*, C_j')$, that is empty for any S-box then the pair must be a wrong-pair. The test sets are essentially random for wrong-pairs so by studying the distribution of input/output differences for the S-boxes we would expect about $\frac{1}{5}$ of the sets to be empty. Thus, the probability of at least one of the 5 test sets being empty is about $1 - 0.8^5 \approx 0.67$. That is, by looking for empty test sets we can eliminate about $\frac{2}{3}$ of the wrong pairs. The number of pairs that would be left after the filtering operation would be about $N = \frac{6}{16}n$. Of this about $\frac{1}{6}$ would be right-pairs. Thus the process of finding the correct subkey bits is simplified, since we only have to search through the subsets $I \subseteq \{1, \ldots, N\}$ to find allowable subsets. We would then expect to find a single allowable subset of size approximately $\frac{N}{6}$ which should suggest the correct 30-bit subkey.

Putting it all together

This section highlights some of the difficulties in implementing a differential analytic attack on the full 16-round version of DES, as well as some of the advances which have been developed to overcome these difficulties. The 1992 paper by Biham and Shamir [7], explains how to launch a successful attack against full 16-round DES.

The method described for 6-round DES does not extend to 16-round DES since it is difficult to find a characteristic that spans sufficient rounds and simultaneously has a sufficiently high probability to enable a useful attack. By way of example, prior to the 1992 paper a 13-round iterated characteristic had been created from a 2-round characteristic with a probability $\frac{1}{234}$. The 13-round characteristic then had a probability of about $2^{-47.2}$ and could be used, in theory, to break 15-round DES. However, if the 2-round characteristic is iterated 7 time, the resultant 14-round characteristic would have a probability of about $2^{-55.1}$ and thus an attack using this would be slower than an exhaustive search.

A further problem with the suggested 15-round attack was that it required up to 2^{42} counters. This imposes a vast storage requirement as well as incurring a time penalty for search through the counters for the correct suggested subkey.

To overcome the difficulty with the longer characteristic having a probability that is too low, Biham and Shamir discovered a way of choosing the input pairs such that they could still use the 13-round characteristic to attack 16-round DES. Furthermore, by analysing the S-box output differences at multiple rounds they were able to effectively suggest a 52-bit subkey. This leaves only $2^4 = 16$ choices for the remaining bits of the suggested key. In this way each input pair could be used to suggest a very small set of possible keys which could be immediately tested for validity.

Thus, there is no longer any need for large arrays of counters (or even for the algorithm suggested for counting suggested subkeys when breaking 6-round DES). Biham and Shamir also incorporated other filtering procedures and optimisations to improve the attack.

Since the input pairs independently suggest keys that can be immediately tested for validity, the attack is totally parallelisable. Up to 2^{33} disconnected processors could be used with a linear speedup. Furthermore, the independence of the testing implies that the attack will even work if the ciphertexts are derived from up to 2^{33} different keys due to frequent key changes.

The attack by Biham and Shamir recovers the key by analysing 2^{36} ciphertexts in 2^{37} time. The 2^{36} ciphertexts are obtained by filtering a larger pool of 2^{47} chosen plaintexts. The filtering is done when the plaintexts are generated and if the plaintext passes the filter then it is immediately input into the differential cryptanalytic attack. Thus the storage requirements of the entire attack are negligible.

4.4 Linear Cryptanalysis

Another method of attack that is able to break full 16-round DES is Linear Cryptanalysis. This method can break DES with an average of 2^{43} known plaintexts and is more efficient than differential cryptanalysis. Linear cryptanalysis was introduced in 1993 by Mitsuru Matsui [27]. Eli Biham helped to formalise the method and demonstrated that although the low-level details are vastly different, there are structural similarities of linear cryptanalysis to differential cryptanalysis [6].

Linear cryptanalysis works by constructing linear approximations of the cryptosystem. The linear approximations are described in terms of subsets of input, output and key bits that are XORed together to obtain a particular value (usually zero). Each linear approximation has a corresponding probability of being valid. If this probability if different from $\frac{1}{2}$ then the bias can be exploited to attack the cryptosystem.

As with differential cryptanalysis the process of finding good linear approximations starts by analysing individual S-boxes. Thus, we consider all possible pairs

$$(\alpha, \beta) \in \left\{ (x, y) \,\middle|\, x \in (\mathbb{Z}_2)^6, y \in (\mathbb{Z}_2)^4 \right\}$$

where α and β are used as bit masks to select which input and output bits of a given S-box are to be XORed together in a linear equation. For a fixed mask (α, β) we count the number of inputs that produce a parity of 0. If this mask creates an unbiased linear approximation then 32 out of the 64 possible inputs would produce a parity of 0 and the other 32 inputs would produce a parity of 1. If, however, there is a linear bias then the count will be different from 32. If we subtract 32 from the count and tabulate the result then the entries with the greatest absolute values will be the best linear approximations for the S-box. By incorporating the expansion permutation, E, and post permutation, P, as well the input subkey we can find a linear approximation for a single round – see Figure 4.7 for an example.

In a process similar to differential cryptanalysis, we can use multiple 1-round linear characteristics to create a linear approximation that spans multiple rounds. With linear cryptanalysis we need to be slightly more careful when composing characteristics since it is possible that one approximation could cancel the next. By refining this approach and using multiple distinct linear characteristics, one can utilise the bias to recover encryption key bits.

A fascinating spin-off of this method is that under certain circumstances it is possible to find linear approximations that are independent of the input plaintext. In particular, Matsui showed that it is possible to break DES with only 2^{29} known ciphertexts if the ciphertexts are produced

EXAMPLE: **Linear Bias in S-Box 5**
Choose $(\alpha, \beta) = (16 = 01000_2, 15 = 1111_2)$ and consider S-box 5, where the 6-bit S-box inputs are $B_1, B_2, \ldots, B_8$ and the corresponding 4-bit outputs are $C_1, C_2, \ldots, C_8$. If we use B[i] to denote the ith bit of B then the parity that we consider is

$$B_5[26] \oplus C_5[1] \oplus C_5[2] \oplus C_5[3] \oplus C_5[4].$$

By calculating the parity for each of the 64 possible inputs we find that it is equal to zero 12 times. If the 32-bit round input is X and the corresponding 32-bit round output is $f(X, K_i)$ then the definition of E, P and the DES key schedule implies that the following linear approximation can be constructed

$$X[26] \oplus f(X, K_i)[3, 8, 14, 25] = K_i[26].$$

The corresponding probability that this approximation is valid is $\frac{12}{64} \approx 0.19$.

Figure 4.7: Linear Approximation of S-Box 5

from natural English plaintexts that are represented by ASCII codes. Further work in this area includes combining the strength of both linear and differential cryptanalysis to create improved attacks on DES and other cryptosystems.

Chapter 5

Knapsack Cryptosystems

5.1 Overview

Knapsack cryptosystems are based on the knapsack or subset-sum problem. The first knapsack based cryptosystem was a public key cryptosystem proposed by Merkle and Hellman in 1978 and was among the first public key cryptosystems to be invented.

The knapsack based cryptosystems were originally considered very promising candidates when compared to other public key encryption systems and were later further adapted by Adi Shamir for digital signatures [29, page 462]. The attraction to knapsack based cryptosystems is their conceptual simplicity, as well as their efficiency when compared to other cryptosystems of comparable key sizes. For example, the RSA algorithm is approximately 100 times slower than the singly-iterated Merkle-Hellman system [3, page 79].

However, the original Merkle-Hellman system was broken by Adi Shamir as shown in his 1984 publication *A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem.* Many other knapsack based systems have been proposed, such as multiple-iterated knapsacks, Graham-Shamir knapsacks and others, but all of these variants have been broken. One version, the Chor-Rivest, has best resisted cryptanalysis but even it has been shown to be weak in some situations [13].

Knapsack based cryptosystem do not appear to be secure and past cryptanalytic work has undermined the trust in such systems. However, these systems are still of theoretical interest when studying the mathematics of both the implementations and the cryptanalysis.

5.2 Subset-Sum Problem

The subset-sum problem is a decision problem. The question that is asked is, given integer weights $a_1, \ldots, a_n$ and an integer s, are there $x_1, \ldots, x_n$ such that

$$\sum_{j=1}^{n} x_j a_j = s, \quad x_j \in \{0, 1\} \quad \forall\, j.$$

For the knapsack problem the weights a_j are interpreted as the sizes of various objects that are to be packed into a knapsack that can be filled to exactly size s. Thus, to solve the knapsack problem we need to not only decide on the existence of the x_j but actually to calculate their values. Notice that if we can efficiently answer the subset-sum decision problem then we can efficiently

solve the knapsack problem. To see this assume that $x_1 = 1$ then test if there exists a solution to

$$\sum_{j=2}^{n} x_j a_j = s - a_1, \quad x_j \in \{0,1\} \quad \forall\, j.$$

If the assumption was correct then there will be a solution to the above equation, otherwise we see that we must have $x_1 = 0$, in any solution to the original problem. We can now iteratively determine values for $x_2, \ldots, x_n$ and find a complete solution [3, page 75].

5.3 Knapsack Cryptosystem

The basic idea of the knapsack cryptosystem is for Alice to choose a set of weights, $a_1, \ldots, a_n$, and publish the set of weights as her public key. When Bob wishes to send Alice a message $(x_1, \ldots, x_n)$, $x_j \in \{0,1\}$ for $1 \leq j \leq n$, he calculates

$$s = \sum_{j=1}^{n} x_j a_j. \tag{5.1}$$

He then transmits the encrypted message s to Alice. If an eavesdropper, Eve, were to intercept the message she would have to solve the general knapsack problem. However, the subset-sum problem is $\mathcal{NP}$-complete and hence, through polynomial reduction, the knapsack problem is also $\mathcal{NP}$-complete [33, page 477]. As we will see in §5.4 the best algorithm for solving the general knapsack problem runs in $O(n2^{n/2})$ time. Thus the problem becomes intractable for Eve to solve. Obviously in the above formalism it would also be intractable for Alice to decrypt the message.

To make the system practical we need to be able to create a system that contains some secret information, a private key, that Alice can use to make the decryption feasible. To achieve this, we first note that there are some types of knapsack problems that are easy to solve. One such knapsack is formed when the set of weights $b_1, \ldots, b_n$ can be ordered to form a super-increasing sequence, that is the b_j satisfy

$$b_j > \sum_{i=1}^{j-1} b_i \quad 2 \leq j \leq n.$$

As a trivial example, if the $b_j = 2^{j-1}$ then the solution $(x_1, \ldots, x_n)$ is merely the binary representation of s. To solve the knapsack problem when the weights form a super-increasing sequence, we note that $x_1 = 1$ if and only if

$$s > \sum_{j=1}^{n-1} b_j.$$

Hence we can determine that $x_n = y$, say, and the knapsack problem can be reduced to the smaller knapsack problem of determining $x_1, \ldots, x_{n-1}$ such that

$$s - y b_n = \sum_{j=1}^{n-1} x_j b_j, \quad x_j \in \{0,1\}, \quad 1 \leq j \leq n-1.$$

The complete solution to the original knapsack problem, $(x_1, \ldots, x_n)$ can thus be retrieved recursively. In this manner the ciphertext message could be decrypted to reveal the original plaintext.

We now describe how a tractable form of the knapsack problem can be transformed into an (apparently) intractable form. That is, we wish to transform a given set of super-increasing weights,

$b_1, \ldots, b_n$, into a set, $a_1, \ldots, a_n$, that conceals the underlying structure and thus renders solving the knapsack problem infeasible. It is this set, $a_1, \ldots, a_n$, that is published as the public key. The creator of the public key, however, retains the transformation parameters and the underlying super-increasing sequence as the private key which enables her to decrypt the ciphertext messages.

In the basic knapsack cryptosystem of Merkle and Hellman we see that modular multiplication is used to transform an easily solvable knapsack problem into an apparently more complicated knapsack. For Alice to use the knapsack cryptosystem, she first needs to construct the secret super-increasing knapsack weights. She does this by choosing the $b_1, \ldots, b_n$ to satisfy

$$b_1 \approx 2^n, \quad b_j > \sum_{i=1}^{j-1} b_i, \quad 2 \leq j \leq n, \quad b_n \approx 2^{2n}. \tag{5.2}$$

Alice then chooses positive integers M and W such that

$$M > \sum_{j=1}^{n} b_j, \quad (M, W) = 1.$$

The choice of the b_j, M and W, is a compromise between efficiency and security. If the b_j, and thus M, are large then the knapsack will be inefficient since n bits of plaintext will be encoded in about $\log_2 M$ bits of ciphertext. However, if the b_j are too small then the system would be insecure, since it would then be possible to reveal either M or W and it can be shown that the knapsack would be breakable.

The ratio of the information bits to the transmitted bits is referred as the *density* of the knapsack and is calculated by

$$\frac{n}{\log_2 \max\{b_1, \ldots, b_n\}}. \tag{5.3}$$

We can see that for the Merkle-Hellman system, the choice of parameters, described above, leads to a density in the order of $1/n$. Thus the knapsack created is a low-density knapsack.

For a practical application, n would be about 250 and thus items of the super-increasing knapsack would range between about 250 and 500 bits in length. Furthermore, W should be somewhere between 100 and 200 bits long [29, page 465].

Alice can now begin to calculate the public key. She first computes

$$a'_j \equiv b_j W (\text{mod } M), \quad 0 < a'_j < M. \tag{5.4}$$

Note that since $(M, W) = 1$ and $M > b_j$ we can not have $a'_j = 0$. Alice then selects the last of her private parameters, a permutation π of $\{1, \ldots, n\}$. Using π she defines

$$a_j = a'_{\pi(j)}, \quad 1 \leq j \leq n. \tag{5.5}$$

Alice now proceeds to publish the a_j as her public key.

It is now possible for Bob to encrypt a plaintext message, $(x_1, \ldots, x_n)$, by computing

$$s = \sum_{j=1}^{n} x_j a_j.$$

Bob then transmits the ciphertext, s, to Alice.

Alice is the only person in possession of the private key (secret parameters) and can now easily decrypt the message. To proceed with the decryption she first computes

$$c = sW^{-1} (\text{mod } M), \quad 0 \leq c < M,$$

where W^{-1} is the multiplicative inverse of W modulo M. From the above definitions of s, a_j and a'_j we see that

$$\begin{aligned} c &\equiv \sum_{j=1}^{n} x_j a_j W^{-1} \ (\text{mod } M) \\ &\equiv \sum_{j=1}^{n} x_j a'_{\pi(j)} W^{-1} \ (\text{mod } M) \\ &\equiv \sum_{j=1}^{n} x_j b_{\pi(j)} \ (\text{mod } M). \end{aligned}$$

Since $M > \sum_{j=1}^{n} b_j$, the choice of $0 \leq c < M$ implies

$$c = \sum_{j=1}^{n} x_j b_{\pi(j)}.$$

This last equation is a knapsack problem set with super-increasing weights, b_j. This type of knapsack can be easily solved, as described above, and thus Alice can now easily decrypt the message sent to her.

5.4 Cryptanalysis

The cryptanalysis of knapsack-type cryptosystems started off slowly and the system was considered to be secure, but still needed to prove itself. However, over time various attacks on the system undermined any future trust in its use. Below is a description of some of the main results pertaining to attacks on knapsack cryptosystems.

5.4.1 Brute Force Attacks

It is impractical to attack the knapsack problem using an exhaustive search since it is an $\mathcal{NP}$-complete problem. However, there is an improvement on searching through all 2^n possible solutions, but this improvement is still impractical for a real attack.

The improved method is to compute

$$\begin{aligned} S_1 &= \left\{ \sum_{j=1}^{\lfloor n/2 \rfloor} x_j a_j \;\middle|\; x_j = 0 \text{ or } 1 \text{ for all } j \right\}, \\ S_2 &= \left\{ s - \sum_{j>\lfloor n/2 \rfloor} x_j a_j \;\middle|\; x_j = 0 \text{ or } 1 \text{ for all } j \right\}. \end{aligned}$$

Now sort each set and scan through them for common elements. If there is an element common to both S_1 and S_2 then there must be a solution to the knapsack problem. We can see this as follows:

if

$$\begin{aligned} y &= \sum_{j=1}^{\lfloor n/2 \rfloor} x_j a_j \\ &= s - \sum_{j>\lfloor n/2 \rfloor} x_j a_j, \end{aligned}$$

then

$$s = \sum_{j=1}^{n} x_j a_j.$$

Creating the sets takes $O(2^{n/2})$ operations, the sorting takes about $O(n2^{n/2})$ operations and finally the scanning for common elements takes $O(2^{n/2})$ operations. Hence the whole procedure takes $O(n2^{n/2})$ operations. However, this method also requires $O(2^{n/2})$ storage space, which may exceed the computation resources available [3, page 76].

5.4.2 Bit Leaking

If the basic knapsack equation,

$$\sum_{j=1}^{n} x_j a_j = s, \quad x_j \in \{0, 1\} \quad \forall\, j,$$

is viewed modulo 2 then one obtains an equation for the x_j modulo 2 and thus a single bit of plaintext information can be obtained if s is intercepted. This is a theoretical break of the knapsack cryptosystem but no way has been discovered to exploit it in practise [3, page 80].

5.4.3 Breaking the Basic Merkle-Hellman System

Adi Shamir published the first serious attack on knapsack cryptosystems. This attack leverages the special structure of the Merkle-Hellman public knapsack to create a super-increasing sequence that can be used to break the cryptosystem. The structure arises due to the construction of the public weights from the private super-increasing weights.

We now show that there is a hidden structure in the public weights, $a_1, \ldots, a_n$, and show how this can be exploited to crack a message encrypted using those weights. Let $b_1, \ldots, b_n$ be the private super-increasing weights that are transformed into the public weights, $a_1, \ldots, a_n$. Let $u \equiv w^{-1} \pmod{m}$ with $0 < u < m$. Then, by (5.4) and (5.5) we have

$$a_j \equiv b_{\pi(j)} w \pmod{m},$$

and thus

$$b_{\pi(j)} \equiv a_j u \pmod{m},$$

which, by definition of congruences, implies that there exists some integer k_j such that

$$a_j u - k_j m = b_{\pi(j)}.$$

Hence,

$$\frac{u}{m} - \frac{k_j}{a_j} = \frac{b_{\pi(j)}}{a_j m}.$$

Since $M > \sum b_j$ and $0 < a_j < M$, we have that $b_{\pi(j)}/a_j M$ is small and thus k_j/a_j is close to U/M. Now, only the a_j and n are publicly known and thus we need to extract some structure so as to obtain information about U, M, π, the b_j, or the k_j. Our remark of closeness provides a lead and we begin to create an estimate of the closeness that is dependent on the publicly known values alone.

From (5.2) we see that

$$b_1, b_2, \ldots, b_5 \lesssim 2^n,$$

and by letting $j_i = \pi^{-1}(i)$ we obtain

$$\left| \frac{U}{M} - \frac{k_{j_i}}{a_{j_i}} \right| \lesssim \frac{2^n}{2^{4n}} = 2^{-3n}, \ 1 \leq i \leq 5. \tag{5.6}$$

Thus we see that

$$|k_{j_i} a_{j_1} - k_{j_1} a_{j_i}| \lesssim 2^n, \ 2 \leq i \leq 5. \tag{5.7}$$

Equation (5.7) shows that the Merkle-Hellman knapsack is not a general knapsack but instead exhibits very particular structure. In particular we see that even though the a_j and k_j are in the order of 2^{2n}, and thus $k_{j_i} a_{j_1}$ is in order of 2^{4n}, the difference of such terms is in the order of 2^n. That is, the a_{j_i} and the k_{j_i} have a very particular dependency.

Shamir noticed that we could determine the k_{j_i} by viewing (5.6) as an integer programming problem. In particular, Lenstra proves in [19] that the integer linear programming problem with a fixed number of variables can be solved in polynomial time. Thus, Shamir has effectively shown that the k_{j_i} with $1 \leq i \leq 5$ can be recovered in polynomial time. Using the k_{j_i} and (5.6) we can construct a pair (U', M') with U'/M' close to U/M such that the weights c_j defined by

$$c_j \equiv a_j U' \pmod{M'}, \quad 0 < c_j < M, \quad 1 \leq j \leq n,$$

form a super-increasing sequence. We note that the original U and M are not recovered but that the super-increasing sequence formed by the c_j can still be used to solve the knapsack problem in the same way that the private weights, b_j, could be used.

For a practical implementation of this attack we note that the $j_1, \ldots, j_5$ can not be explicitly selected since π is private. However, we can try all n^5 possible choices for the j_i and the total running time of the attack would remain polynomial in n [3, pages 82–83].

5.4.4 Solving Low Density Knapsacks

The cryptanalysis presented above constructs a direct link between the public weights and the private weights so as to attack the knapsack used in the cryptosystem. This section discusses a more general approach to solving low-density (5.3) knapsack problems which can then be used to attack many knapsacks used in cryptosystems. There are two known approaches to solving low-density knapsacks, of which one is due to Brickell [11], and the other is due to Lagarias and Odlyzko. We shall present the approach described by Lagarias and Odlyzko and provide the necessary mathematical background [33, pages 447–461 and 477–479].

The solution to low-density knapsacks rests on results from lattice theory and integer programming, in particular on results regarding short vectors in lattices. We begin by giving a definition of a lattice.

Definition 5.1 (Lattice) *Given* $n \in \mathbb{N}$ *and* $f_1, \ldots, f_n \in \mathbb{R}^n$ *with* $f_i = (f_{i1}, \ldots, f_{in})$ *and the* f_i *linearly independent over* $\mathbb{R}^n$*. Then*

$$L = \mathbb{Z}f_1 \oplus \ldots \oplus \mathbb{Z}f_n = \sum_{1 \leq i \leq n} \mathbb{Z}f_i = \left\{ \sum_{1 \leq i \leq n} r_i f_i \,\middle|\, r_1, \ldots, r_n \in \mathbb{Z} \right\}$$

is the lattice, or $\mathbb{Z}$*-module, generated by* $f_1, \ldots, f_n$*. The vectors* $f_1, \ldots, f_n$ *are called the basis of* L*.*

Definition 5.2 (Vector Norm, Inner Product and Orthogonality) *Let* $f = (f_1, \ldots, f_n) \in \mathbb{R}^n$ *be a vector. Then we can define a norm of* f *by*

$$\|f\| = \|f\|_2 = \left(\sum_{i=1}^{n} f_i^2 \right)^{1/2} = (f \cdot f)^{1/2} \in \mathbb{R}.$$

This is the definition of the Euclidean norm, or 2*-norm, and* $f \cdot g = \sum_{1 \leq i \leq n} f_i g_i$ *is the usual inner product. If we have* $f \cdot g = 0$ *then* f *and* g *are said to be orthogonal.*

Definition 5.3 (Lattice Norm) *Given* $f_1, \ldots, f_n$*, a basis of lattice* L*, we can define a norm of* L *by*

$$|L| = |\det(f_{ij})_{1 \leq i,j \leq n}| \in \mathbb{R}.$$

Before we can use this norm we need to show that it is well defined, that is we need to show that it is independent of the basis used. To this end we present the following theorem.

Theorem 5.4 *Let* $N \subseteq M \subseteq \mathbb{R}^n$ *be lattices generated by* $g_1, \ldots, g_n$ *and* $f_1, \ldots, f_n$ *respectively. Then* $|M|$ *divides* $|N|$*.*

PROOF: Since $N \subseteq M$ we have that for each g_i there exist $a_{ij} \in \mathbb{Z}$ with $1 \leq j \leq n$ such that $g_i = \sum_{1 \leq j \leq n} a_{ij} f_j$. Hence, by properties of determinants, we have $|N| = |\det(a_{ij})| \cdot |M|$. In other words, $|M|$ divides $|N|$ as required. □

Now if we have $N = M$ then clearly $N \subseteq M$ and $M \subseteq N$. Then by the above theorem we have that $|M|$ divides $|N|$ and $|N|$ divides $|M|$ and thus $|M| = |N|$, from which it follows that the norm is well defined. Geometrically, $|L|$ is the volume of the parallelepiped spanned by the basis vectors of L, $f_1, \ldots, f_n$. Clearly, $\prod_{1 \leq i \leq n} \|f_i\|$ is the volume of the n-dimensional rectangle with edges of length $\|f_i\|$. This provides an intuitive reason for Hadamard's inequality which states that

$$|L| \leq \|f_1\| \ldots \|f_n\|. \tag{5.8}$$

In our quest for finding short vectors in lattices we now move on to review the Gram-Schmidt orthogonalisation process, after which we will show how this relates to finding short vectors.

Given a basis $f_1, \ldots, f_n$ of $\mathbb{R}^n$ we can use the Gram-Schmidt process to calculate a new basis $f_1^*, \ldots, f_n^*$, such that the f_i^* are mutually orthogonal. Basically the f_i^* are calculated inductively by considering, for each $2 \leq i \leq n$, the space spanned by the $f_1, \ldots, f_{i-1}$ and then letting f_i^* be the projection of f_i orthogonal to that space, with $f_1^* = f_1$. Thus we see that the f_i^* are defined inductively by

$$f_i^* = f_i - \sum_{1 \leq j < i} \mu_{ij} f_j^*, \text{ where } \mu_{ij} = \frac{f_i \cdot f_j^*}{f_j^* \cdot f_j^*} = \frac{f_i \cdot f_j^*}{\|f_j^*\|^2} = \text{ for } 1 \leq j < i. \tag{5.9}$$

For clarity we now place this into matrix notation. Consider the f_i and f_i^* as row vectors in $\mathbb{R}^n$, and define the $n \times n$ matrices F,F* and M in $\mathbb{R}^{n \times n}$

$$F = \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix}, F^* = \begin{pmatrix} f_1^* \\ \vdots \\ f_n^* \end{pmatrix}, M = (\mu_{ij})_{1 \leq i,j \leq n},$$

where $\mu_{ii} = 1$ for $i \leq n$, and $\mu_{ij} = 0$ for $1 \leq i < j \leq n$. We then see that we can rewrite (5.9) as

$$F = \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix} = \begin{pmatrix} 1 & & 0 \\ \vdots & \ddots & \\ \mu_{n1} & \dots & 1 \end{pmatrix} \begin{pmatrix} f_1^* \\ \vdots \\ f_n^* \end{pmatrix} = MF^*.$$

We now show how the Gram-Schmidt process relates to finding short vectors.

Theorem 5.5 *Given a basis, $f_1, \dots, f_n$, for a lattice, $L \subseteq \mathbb{R}^n$, with $f_1^*, \dots, f_n^*$ the Gram-Schmidt orthogonal basis. Then for any $f \in L \backslash \{0\}$ we have*

$$\|f\| \geq \min\{\|f_1^*\|, \dots, \|f_n^*\|\}.$$

PROOF: Choose any $f = \sum_{1 \leq i \leq n} \lambda_i f_i \in L \backslash \{0\}$ with $\lambda_i \in \mathbb{Z}$. Let k be the highest index such that $\lambda_k \neq 0$. From the Gram-Schmidt process we can substitute $\sum_{1 \leq j \leq i} \mu_{ij} f_j^*$ for each f_i, which together with the definition of k gives

$$f = \sum_{1 \leq i \leq k} \lambda_i \sum_{1 \leq j \leq i} \mu_{ij} f_j^* = \lambda_k f_k^* + \sum_{1 \leq i < k} \nu_i f_i^*$$

for appropriate $\nu_i \in \mathbb{R}$. We can then calculate $\|f\|^2$ and simplify using the pairwise orthogonality of the f_i^* and that $\lambda_k \in L \backslash \{0\}$ as follows:

$$\begin{aligned} \|f\|2 &= f \cdot f = \left(\lambda_k f_k^* + \sum_{1 \leq i < k} \nu_i f_i^*\right) \cdot \left(\lambda_k f_k^* + \sum_{1 \leq i < k} \nu_i f_i^*\right) \\ &= \lambda_k^2 (f_k^* \cdot f_k^*) + \textstyle\sum_{1 \leq i < k} (f_i^* \cdot f_i^*) \\ &\geq \lambda_k^2 \|f_k^*\|^2 \\ &\geq \|f_k^*\|^2 \\ &\geq \min\{\|f_1^*\|^2, \dots, \|f_n^*\|^2\}. \end{aligned}$$

□

Thus, from the above theorem, if the Gram-Schmidt orthogonal basis is also a basis for the lattice, then one of the f_i^* would be the shortest element of the lattice. However, generally the f_i^* fall outside of L. Even though this approach does not find the shortest vectors in a general lattice, it is still promising and will lead to an algorithm for finding short vectors, though not necessarily the shortest vector.

The Gram-Schmidt orthogonalisation process motivates the notion of an "almost orthogonal" basis. This is a basis that satisfies the constraints of the lattice and is close to being orthogonal. An "almost orthogonal" basis for a lattice is called a reduced basis. More precisely, we give the following definition [19, page 540]:

Definition 5.6 (Reduced Basis) *A basis,* $f_1, \ldots, f_n$*, for the lattice* L *is called a reduced basis if there exists a constant* c*, that depends only on the dimension* n*, such that*

$$\prod_{1 \le i \le n} \|f_i\| \le c|L|.$$

When related to Hadamard's inequality (5.8) and the fact that $|L|$ is the volume of the parallelepiped spanned by the basis vectors, we see that this is an intuitive definition. The above definition states that the basis is a reduced basis if it can, in a sense, be bounded by the norm of the lattice.

Definition 5.6 can be directly related to the following more technical result which is sometimes used as a definition of a reduced basis.

Theorem 5.7 *Let* $f_1, \ldots, f_n \in \mathbb{R}^n$ *be linearly independent with* $f_1^*, \ldots, f_n^*$ *being the corresponding Gram-Schmidt orthogonal basis. Then* $f_1, \ldots, f_n$ *is a reduced basis if* $\|f_i\|^2 \le 2\|f_{i+1}^*\|^2$ *for* $1 \le i < n$*.*

Theorem 5.5 provides a lower bound for the short vectors of a lattice in the Gram-Schmidt orthogonal basis. Using a reduced basis the following theorem shows a weaker bound for short vectors.

Theorem 5.8 *Given* $L \subseteq \mathbb{R}^n$ *a lattice with a reduced basis* $f_1, \ldots, f_n$*, then for any* $f \in L \backslash \{0\}$ *we have*

$$\|f_1\| \le 2^{(n-1)/2}\|f\|.$$

PROOF: From the Gram-Schmidt process we have $\|f_1^*\| = \|f_1\|$. Then from Theorem 5.7 we get $\|f_1\|^2 = \|f_1^*\|^2 \le 2\|f_2^*\|^2 \le 2^2\|f_3^*\|^2 \le \ldots \le 2^{n-1}\|f_n^*\|^2$. In other words, $\frac{1}{2}x_1 \le x_2$, $\frac{1}{4}x_1 \le x_3$,..., $\frac{1}{2^{n-1}}x_1 \le x_n$. Hence, using Theorem 5.5 we get $\|f\| \ge \min\{f_1^*, \ldots, f_n^*\} \ge 2^{-(n-1)/2}\|f_1\|$. □

We now present an algorithm that computes a reduced basis of a lattice $L \subseteq \mathbb{R}^n$ from an arbitrary basis. The algorithm we present is that from [33, page 452] and is given without proof. This algorithm is also presented in [19]. Let $\lceil \mu \rfloor = \lfloor \mu + \frac{1}{2} \rfloor$ denote the integer closest to μ.

ALGORITHM: **Basis Reduction**:

Input: Basis $f_1, \ldots, f_n \in \mathbb{Z}^n$ of lattice $L = \sum_{1 \le i \le n} \mathbb{Z} f_i \subseteq \mathbb{Z}^n$.
Output: A reduced basis $g_1, \ldots, g_n$ of L.

1. **for** $i = 1, \ldots, n$ **{**
2. $\quad g_i \leftarrow f_i$.
   }
3. compute the Gram-Schmidt Orthogonalisation $G^*, M \in \mathbb{Q}^{n \times n}$, $i \leftarrow 2$.
4. **while** $i \le n$ **{**
5. $\quad$ **for** $j = i-1, i-2, \ldots, 1$ **{**
6. $\quad\quad g_i \leftarrow g_i - \lceil \mu_{ij} \rfloor g_j$,
   $\quad\quad$ update the Gram-Schmidt Orthogonal basis replacing G^* and M.
   $\quad$ **}**

7. **if** $i > 1$ and $\|g_{i-1}^*\|^2 > 2\|g_i^*\|^2$ **then**
8. exchange g_{i-1} and g_i and update the Gram-Schmidt Orthogonal basis.
9. **else**
10. $i \leftarrow i + 1$.

}

11. **return** $g_1, \ldots, g_n$.

□

It can be shown that the above algorithm completes in polynomial time and correctly computes a reduced basis of L. It performs $O(n^4 \log A)$ arithmetic operations on integers of length $O(n \log A)$, where $A = \max_{1 \le i \le n} \|f_i\|$.

Having now shown how reduced bases relate to short vectors (Theorem 5.8) and how to find a reduced basis, we return to the original problem of breaking the knapsack cryptosystem. To see how short lattice vectors and knapsacks are related, consider an $(n+1)$-dimensional lattice, L, with the following basis

$$V = \begin{pmatrix} 1 & 0 & \ldots & 0 & -a_1 \\ 0 & 1 & \ldots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & -a_n \\ 0 & 0 & \ldots & 0 & s \end{pmatrix}, \tag{5.10}$$

where $a_1, \ldots, a_n$ are the weights and s is the size of the knapsack. Now if $(x_1, \ldots, x_n) \in \{0,1\}^n$ is a solution to the original knapsack problem (5.1) and the rows of V are denoted by the vectors $v_1, \ldots, v_{n+1}$ then we have

$$w = \sum_{1 \le i \le n} x_i v_i + v_{n+1} = (x_1, \ldots, x_n, 0) \in L.$$

Since the x_i are 0 or 1 we have $\|w\| \le \sqrt{n}$. Thus w is a very short vector compared to most vectors in L since the a_i are generally large. Thus to break a knapsack cryptosystem we construct the lattice basis (5.10) and then apply the basis reduction algorithm to find a reduced basis. This reduced basis then presents us with a set of short vectors that can be checked to see if it contains a solution to the knapsack problem.

This approach does not work well for the general knapsack problem. However, for a low-density knapsack (5.3), as is the case when the a_i are large, one could expect that the above procedure would often be successful. More rigorous results have been proven to uphold this view and furthermore, empirical evidence shows that the procedure generally performs better in practise than the guaranteed theoretical bounds.

5.5 Conclusion

Knapsack cryptosystems will probably never be used in practical cryptographic applications. However, the study of knapsack systems has lead to further investigation into integer programming, basis reduction, and the study of lattices. These fields still show promise for cryptography, in

particular the complexity of the problem of finding the shortest non-zero vector in lattices has been shown to be $\mathcal{NP}$-hard by Miklos Ajtai [33, pages 554–555]. Furthermore, Ajtai showed that the problem is $\mathcal{NP}$-hard on average and not just in the worst case. Through the work of Ajtai & Dwork this discovery has been used to create a cryptosystem. The work of Ajtai has also inspired an improved public key cryptosystem [15] which uses the shortest non-zero vector problem to create a trapdoor function and in turn an algorithm that is more efficient than the Ajtai-Dwork system ($O(\mathfrak{n}^2)$ vs $O(\mathfrak{n}^4)$).

Chapter 6

Pseudo-Random Number Generators

6.1 Overview

Random values are useful in many computational disciplines, such as numerical analysis, Monte Carlo simulations and statistical sampling. In cryptography, random numbers are used abundantly, both directly (such as in one time pads) and indirectly (to generate the keys used in cryptosystems, or challenges used in challenge/response protocols).

To gain a better understanding of random numbers, we need to look at both the properties and the sources of random numbers. At an intuitive level we note that random numbers are non-deterministic, yet computers are completely deterministic and thus algorithms that are implemented on a computer can not produce true random numbers. True random numbers can be obtained by measuring physical processes such as flipping a coin, the keyboard latency of a user typing at a computer, or even the frequency instability of a free-running oscillator (AT&T built a commercial chip that uses this last phenomenon to generate random numbers [29, page 423]). Cryptosystems, however, often require a large quantity of random numbers and most cryptographic implementations do not have access to a plentiful supply of true random numbers. This disparity leads to the introduction of the notion of pseudo-random numbers.

Pseudo-random numbers share many of the properties of random numbers, but are generated by deterministic algorithms. For cryptographic applications it is useful to distinguish between two broad classes of properties of random numbers: statistical properties and unpredictability. It is not difficult to construct algorithms for generating pseudo-random numbers that satisfy sufficiently many statistical tests to be appropriate for use in most computational disciplines. Cryptography, however, requires that the pseudo-random numbers also be sufficiently unpredictable and this is more difficult to achieve.

In the following sections we will present a more rigorous definition of random numbers and of pseudo-random numbers. We will then present a number of the more common algorithms for producing pseudo-random numbers, following which we will discuss the cryptanalysis of pseudo-random number generators.

6.2 Definitions of Randomness

We will first provide a definition of a random number sequence.

Definition 6.1 (Random Number Sequence) *A sequence of random numbers,* $x_1, \ldots, x_n$*, is a sequence obtained via independent samples of a fixed probability distribution on some sample space.*

Generally the field of random number generators is discussed in terms of generating random sequences of bits. That is, the samples are drawn independently from the set $\{0, 1\}$. Random bit generators can be used to create sequences of random numbers (if we need a random number from the interval $[0, n]$ we can use a sequence of $\lfloor \log_2 n \rfloor + 1$ bits to create a random number and then test if the number does in fact fall within the required range and discard it, and try again, if it does not).

6.2.1 Sources of Real Random Data

The above definition motivates using various naturally occurring physical processes as a source of random bits, since the measurements can be seen to be independent samples [29, page 422]. In applications this is often referred to as *entropy gathering*. Entropy gathering is used in applications such as PGP (used to encrypt e-mail messages), which can use keyboard timings to obtain a random seed which is used when creating encryption keys.

Below is a list of some sources of true random data.

- **RAND Tables:** In 1955 the Rand Corporation published a book that contained one million random digits. The numbers were generated by an electronic roulette wheel that could produce one number a second.
- **Geiger Counter:** A Geiger counter could count the number of emissions over a fixed period of time and then be used to generate a 1 if the count is odd and 0 if it is even (i.e. choose the least significant bit, of the counter).
- **Keyboard Latency:** When a person types at a keyboard, the resulting timing pattern contains both a random and a non-random component. The non-random component can be used to identify the user that is typing. However, if one only concentrates on the fluctuation in the time between successive keystrokes and then looks at the least significant bit the result is random.
- **Lava Lite lamp:** The motion of the "lava" in a lava lamp is chaotic and thus a good potential source of random data. This inspired a team at Silicon Graphics, Inc. to use a digital camera to capture images of the lamps at regular time intervals (see http://lavarand.sgi.com/). The resultant image data was passed through a SHA-1 (Secure Hashing Algorithm) hash function to generate a 140-bit seed. This seed was in-turn used to generate a random sequence using the Blum-Blum-Shub generator.

Normally we would need to distill the random data from the real world measurements, and discard any biased or correlated bits. One possible result of the measurement is a stream of uncorrelated but biased bits, where the probability of a 1 is not 1/2. In this case a very simple procedure can be employed to remove the bias. Group the measured output into pairs and if the pair is 10 output a 0, if the pair is 01 output a 1, while the pairs 00 and 11 are discarded [2, page 173]. A more general procedure (though not provably secure) is to use the measured bit stream as input for a cryptographic hash function such as SHA or MD5 [29, page 426]. In a practical application the result of the hash function would then be used as a seed for a pseudo-random bit generator such as the Blum-Blum-Shub generator, see §6.3.5.

6.2.2 Testing Data For Randomness

Once we have acquired bit sequences we wish to be able to test that the sequences are random in nature. There are a number of statistical tests that can be used to verify that the data is random. First, a statistic is any deterministic function $\sigma(x)$ of a sample x drawn from a distribution. A statistical test is created by calculating a statistic over a sample drawn from the probability distribution generated by the bit-generator and comparing it to a tolerance level [18, page 126]. Below are some of the more common statistical tests which operate on a sample $s = s_0 s_1 \dots s_{n-1}$ of length n [2, page 183]:

- **Frequency Test (mono-bit):** It is expected that on average the number of 0s and the number of 1s occurring in a random bit sequence would be the same. Let n_0 and n_1 denote the number of 0s and 1s respectively. The statistic calculated is

$$X_1 = \frac{(n_0 - n_1)^2}{n}.$$

  This statistic approximately follows the χ^2 distribution with 1 degree of freedom.

- **Serial Test (bi-bit):** This test checks if the number of occurrences of 00, 01, 10 and 11 is the same as would be expected from a true random sequence. Let n_0 and n_1 denote the number of 0s and 1s respectively and let n_{00},n_{01},n_{10} and n_{11} denote the number of 00, 01, 10 and 11 respectively. Then the statistic to be calculated is

$$X_2 = \frac{4}{n-1}(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2) - \frac{2}{n}(n_0 + n_1) - 1.$$

  This statistic approximately follows the χ^2 distribution with 2 degrees of freedom.

- **Poker Test:** The poker test is a generalisation of the frequency test. The poker test determines whether or not varying sequences of length m appear in the sample as often as they would in a true random sequence. Let m be a positive integer such that $\lfloor \frac{n}{m} \rfloor \geq 5 \cdot 2^m$ and let $k = \lfloor \frac{n}{m} \rfloor$. Divide the sample s into k non-overlapping parts of length m and let n_i, $0 \leq i \leq m$, be the number of occurrences of the m-bit sequence that would represent 2^i in binary. The statistic to calculate is

$$X_3 = \frac{2^m}{k}\left(\sum_{i=1}^{2^m} n_i^2\right) - k.$$

  This statistic approximately follows the χ^2 distribution with $2^m - 1$ degrees of freedom.

- **Runs Test:** This test checks if the number of runs of 0s or 1s, of various lengths, in the sample sequence corresponds to what would be expected from a true random sequence. A run is a consecutive sequence consisting of only 1s or 0s such that the sequence is not preceded by or succeeded by the same symbol. Runs of 0s are referred to as gaps and runs of 1s are referred to as blocks. The expected number of gaps (and thus also blocks) of length i in a random sequence of length n is $e_i = (n - i + 3)/2^{i+2}$. Let k be the largest integer i such that $e_i \geq 5$. Let B_i and G_i denote the number of blocks and gaps respectively of length i in the sample, $1 \leq i \leq k$. The statistic is defined by

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}.$$

  This statistic approximately follows the χ^2 distribution with $2^k - 2$ degrees of freedom.

- **Autocorrelation Test:** This test checks for the amount of correlation of the sample s with a shifted version of it. Let d be an integer with $1 \leq d \leq \lfloor n/2 \rfloor$. The number of bits in s not equal to their d-shifts is given by $A(d) = \sum_{i=1}^{n-d-1} s_i \oplus s_{i+d}$, where $\oplus$ denotes the XOR operator. The statistic that is tested is given by

$$X_5 = \frac{2(A(d) - \frac{n-d}{2})}{\sqrt{n-d}}.$$

  This statistic approximately follows the normal distribution with a mean of 0 and variance of 1.

- **Maurer's Universal Statistical Test:** This test is based on the concept that any truly random sequence should not be significantly compressible (using a non-lossy compression algorithm). The test is referred to as universal because it can detect a very general class of possible defects. In particular, it can be shown that if any of the above tests fail then the Maurer test will also fail.

If a bit-generator fails any one of the tests then it is rejected and is not considered to produce a sufficiently good approximation of random data. If a bit-generator passes a given set of statistical tests then it is accepted to be sufficiently random, though strictly speaking this indicates that it is "not rejected" rather than being "accepted." We note that the conclusion of a statistical test is probabilistic and thus the result may be different next time the statistical test is invoked.

6.2.3 Pseudo-Randomness

As noted earlier, for practical implementations, it is not enough to identify a random source and distill random bits from the source. Many sources of random bits are not readily available to many implementations of cryptosystems (such as "varactor diodes" which are not commonly available to most desktop computers). Furthermore, many random sources do not generate enough random data for use in cryptosystems. Also, a true random source can never reproduce the exact same sequence as was acquired from a previous sample and this is sometimes a useful property. To this end we introduce pseudo-random number generators (PRNGs), or more particularly pseudo-random bit generators (PRBGs).

A PRBG is a deterministic algorithm for generating bit sequences that "look random." Most PRBGs take, as input, a short random bit-string (seed value), which is drawn from a given probability distribution, and then generate a longer sequence of apparently random bits, which simulates a sample drawn from another target probability distribution on a range space. This leads to the following definition of a PRBG.

Definition 6.2 (Pseudo-Random Bit Generator) *Let k,l be positive integers with $l \geq k+1$. A (k,l)-PRBG is a function $f : (\mathbb{Z}_2)^k \to (\mathbb{Z}_2)^l$ that can be computed in a time period polynomial in k. The input $s_0 \in (\mathbb{Z}_2)^k$ is called the seed, and the output $f(s_0) \in (\mathbb{Z}_2)^l$ is called a pseudo-random bit-string.*

To test if the PRBG is performing well we wish to show that it has the property of being computationally randomness-increasing. That is, we wish to show that the entropy of the distribution generated by the PRBG appears to be higher than the entropy of the input distribution. Strictly speaking this can never be the case since a deterministic mapping can never increase the entropy in a system. However, when computing power is limited, the generated distribution may approximate

a distribution having a much greater entropy, and within the limits of the available computing power one cannot tell the distributions apart [18, page 124].

Intuitively we want it to be impossible, in an amount of time that is polynomial in k, to distinguish a string of l pseudo-random bits produced by a (k, l)-PRBG from a string of l truly random bits. To express this more formally, we introduce the concept of indistinguishable probability distributions [31, page 363].

Definition 6.3 (ϵ-distinguisher) *Let p_0 and p_1 be two probability distributions on the domain $(\mathbb{Z}_2)^l$ of bit-strings of length l. Let $\mathbf{A} : (\mathbb{Z}_2)^l \to \{0, 1\}$ be a probabilistic algorithm that runs in polynomial time, as a function of l. Let $\epsilon > 0$. For $j = 0, 1$ define*

$$E_{\mathbf{A}}(p_j) = \sum_{(z_1,\ldots,z_l)\in(\mathbb{Z}_2)^l} p_j(z_1, \ldots, z_l) \times p(\mathbf{A}(z_1, \ldots, z_l) = 1 | (z_1, \ldots, z_l)).$$

If

$$|E_{\mathbf{A}}(p_0) - E_{\mathbf{A}}(p_1)| \geq \epsilon$$

then $\mathbf{A}$ is called an ϵ-distinguisher of p_0 and p_1.

p_0 and p_1 are called ϵ-distinguishable if there exists an ϵ-distinguisher of p_0 and p_1.

The algorithm $\mathbf{A}$ attempts to predict if a bit-string $(z_1, \ldots, z_l)$ is more likely to have arisen from probability distributions p_0 or p_1. We then calculate the expectation $E_{\mathbf{A}}(p_j)$ of the output of $\mathbf{A}$ over the entire probability distribution, for each of the two probability distributions. If the expectations differ by more than ϵ then we have found a process for distinguishing instances of one probability distribution from another. The algorithm $\mathbf{A}$ or more particularly $E_{\mathbf{A}}$ relates directly to statistical tests mentioned above.

Clearly if sequences of l bits were generated by a truly random process then each of the 2^l sequences would occur with an equal probability, $1/2^l$. Let this probability distribution be called p_0. We now use p_0 as a reference to which we compare the probability distribution on $(\mathbb{Z}_2)^l$ induced by the PRBG p_1, defined by the likelihood of the PRBG in question outputting various bit-strings of length l. If we can then find an ϵ-distinguisher for some ϵ then the PRBG is not as random as we would like because it can be distinguished from the uniform distribution.

For cryptographic systems it is not enough that the distribution is good, but the bit sequences must also be unpredictable. For this, the concept of next-bit predictors is useful. Here we use the previous bits from a sequence generated by a PRBG to attempt to guess the next bit that would be generated.

Definition 6.4 (ϵ-next bit predictor) *Let f be a (k, l)-PRBG. Let $\mathbf{B}_i : (\mathbb{Z}_2)^{i-1} \to \{0, 1\}$ be a probabilistic algorithm that accepts the first $i-1$, with $0 < i \leq l-1$, bits generated by f and attempts to predict the next bit, z_i. $\mathbf{B}_i$ is called an ϵ-next bit predictor if it can correctly predict the ith bit generated by f with a probability of at least $1/2 + \epsilon$, where $\epsilon > 0$.*

Interestingly, Yao has shown that an ϵ-next bit predictor is a universal test for PRBG, that is a PRBG is "secure" if and only if there does not exist an ϵ-next bit predictor except for very small values of ϵ. The actual result is a general theorem due to Yao that shows the equivalence between ϵ-next bit predictors and ϵ-distinguishers [18, page 126].

6.3 Pseudo-Random Number Generating Algorithms

The previous section lays down the theory for random numbers and provides definitions of pseudo-random bit generators. We now describe some of the commonly used algorithms for generating sequences of pseudo-random bits.

6.3.1 Linear Feedback Shift Registers

Shift registers are simple devices that can be used to produce random bit sequences. These devices can be very efficiently implemented in hardware, and can also be implemented well in software. Shift registers consist of a fixed length chain of single bit registers. Each time a new bit is needed the bit is extracted from one end of the chain of registers, all the bits are shifted along by one so as to overwrite the bit that was just read and then a new bit is computed (using an update function taking the current bits as input) and placed into the far end, see Figure 6.1.

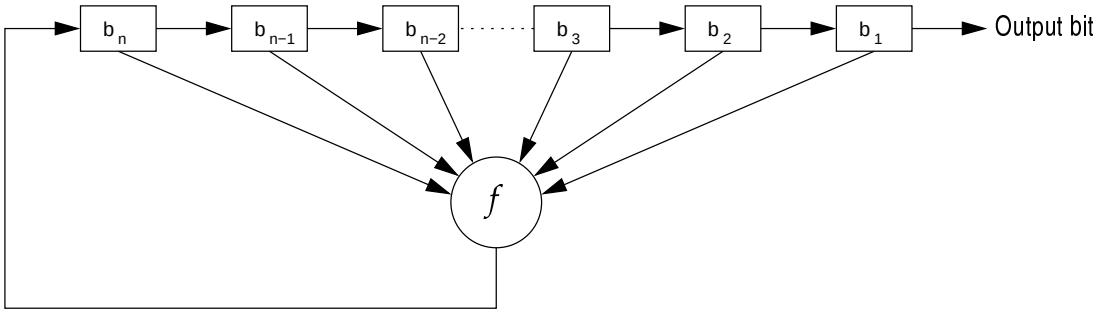


Figure 6.1: Feedback Shift Register

With linear feedback shift registers (LFSR) the update function consists of XORing a fixed subset of the bit-register values together, to calculate the new bit. This subset is referred to as the tap sequence. Any LFSR will ultimately produce periodic output (that is except for a fixed length, possibly zero, initial part of the sequence, the rest of the sequence will be periodic). If the length of the LFSR is n then the LFSR can only be in $2^n - 1$ internal states. Thus the maximal period is $2^n - 1$. The actual period that results is dependent on the tap sequence.

Definition 6.5 (Linearly Recurrent and Characteristic Polynomials) *Given a sequence* $s = (s_i)_{i \in \mathbb{N}}$ *with each* $s_i \in \mathbb{Z}_2$ *then if there exists* $n \in \mathbb{N}$ *and* $f_0, \ldots, f_n \in \mathbb{Z}_2$ *with* $f_n \neq 0$ *such that*

$$\sum_{j=0}^{n} f_j s_{i+j} = f_n s_{i+n} + \ldots + f_1 s_{i+1} + f_0 s_i = 0$$

for all $i \in \mathbb{N}$ *then the sequence is said to be linearly recurrent. Furthermore, the polynomial* $f = \sum_{j=0}^{n} f_j x^j \in \mathbb{Z}_2[x]$ *is called a characteristic polynomial of* s*, also known as an annihilating or generating polynomial.*

We can view the tap sequence as defining a characteristic polynomial $f(x)$ over $\mathbb{Z}_2[x]$ for the sequence output by the LFSR. It can be shown that the period is maximal if and only if the polynomial corresponding to the tap sequence is a primitive polynomial, $f(x)$ over $\mathbb{Z}_2[x]$. That is, $f(x)$ is an irreducible polynomial of degree n and x is a generator of $\mathbb{F}_{2^n}^*$, the multiplicative group

formed by the non-zero elements of $\mathbb{Z}_2[x]/\langle f(x)\rangle$. In more colloquial terms $f(x)$ is primitive if x is a multiplicative generator of $\mathbb{Z}_2[x]$ modulo $f(x)$. It can be shown that the irreducible polynomial $f(x)$ is primitive if and only if $f(x)$ divides $x^k - 1$ for $k = 2^n - 1$ and for no smaller positive integer k (this extends to any prime, p, not just $p = 2$).

Cryptanalysis

We now introduce results showing that LFSR based PRNGs can be broken. Clearly if we can recover the tap sequence and the register states at a given time then we can generate the continuation of the sequence and thus the PRBG has been broken.

We now note that the characteristic polynomial created from the tap sequence is not necessarily unique. There may be many other characteristic polynomials for the same sequence and if we recover any one of the characteristic polynomials for the LFSR then we can generate the same sequence that the LFSR generates. The set of all characteristic polynomials for s forms an ideal in $\mathbb{Z}_2[x]$. This ideal is called the annihilator of s and is denoted by $\mathrm{Ann}(s)$. It can be shown that any ideal in $\mathbb{Z}_2[x]$ can be generated by a single polynomial, in particular either $\mathrm{Ann}(s) = \{0\}$ or there exists a unique monic polynomial $m \in \mathrm{Ann}(s)$ of least degree such that $\langle m\rangle = \{rm | r \in \mathbb{Z}_2[x]\} = \mathrm{Ann}(s)$. This polynomial is called the minimal polynomial of s [33, pages 318–323].

The Berlekamp-Massey algorithm can be used to determine the coefficients of the linear recurrence of shortest length that generates the sequence $s = s_0, s_1, \ldots, s_{n-1}$ [25, pages 148–149]. That is, the Berlekamp-Massey algorithm can be used to find the minimal polynomial of s. In this way any sequence which is generated by a linear recurrence, such as the sequence generated by LFSRs, can by broken.

6.3.2 Linear Congruential Generators

A linear congruential generator is a simple generator given by the equation

$$x_n = ax_{n-1} + b \bmod m,$$

where $a, b, m \in \mathbb{Z}$ are secret parameters of the generator and x_0 is a secret seed. This generator exhibits good performance and also passes all the necessary statistical tests when the parameters are chosen appropriately. Work has been carried out which provides details for how to choose the parameters so that the generator will have a maximal period – that is a period of m.

Cryptanalysis

However, this generator was rendered useless for cryptographic purposes by Boyar in 1989. Boyar's algorithm breaks the linear congruential generator by recovering the secret parameters a,b and m. An important aspect of Boyar's work is that she proved that her algorithm will always arrive at a correct solution in polynomial time and thus breaking linear congruential generators is computationally feasible.

She then extended her work to break quadratic generators:

$$x_n = (ax_{n-1}^2 + bx_{n-1} + c) \bmod m,$$

and cubic generators:

$$x_n = (ax_{n-1}^3 + bx_{n-1}^2 + cx_{n-1} + d) \bmod m.$$

Other people have extended her work to general polynomial congruential generators.

One of the main observations that Boyar made was that with the linear congruential generator defined as above we have that

$$x_{n+1} \equiv a(x_n - x_{n-1}) \pmod{m}$$

and

$$x_{n+2} - x_{n+1} \equiv a(x_{n+1} - x_n) \pmod{m}.$$

This in turn implies that m divides

$$(x_{n+1} - x_n)^2 - (x_n - x_{n-1})(x_{n+2} - x_{n+1}), \tag{6.1}$$

provided that a and m are co-prime. From this it is apparent that if one has captured sufficient output from the generator one can start generating potential values of m by noting that m must be larger than any output of the generator and must satisfy (6.1).

We present a simple example which recovers the secret parameters for the following short sequence generated by a linear congruential generator: $1, 3, 10, 12, 4, \ldots$. Firstly we note that m must be larger than 12. Then substituting into (6.1) with $n = 1$ we get:

$$(10-3)^2 - (3-1)(12-10) = 45 = 3 \cdot 3 \cdot 5,$$

and with $n = 2$

$$(12-10)^2 - (10-3)(4-12) = 60 = 2 \cdot 2 \cdot 3 \cdot 5.$$

Thus m must be a common divisor of 45 and 60, and must be greater that 12. From the factorisations it is easy to see that $m = 15$. Now

$$\begin{aligned} x_1 &= ax_0 + b \bmod m \\ x_2 &= ax_1 + b \bmod m \end{aligned}$$

implies that

$$(x_2 - x_1) = a(x_1 - x_0) \bmod m.$$

Substituting in we get $(10-3) = a(3-1) \bmod 15$ and thus $a = 11$. From a single iteration we can now find that $b = 7$. We can use these parameters and the seed of 1 to produce the full sequence: $1, 3, 10, 12, 4, 6, 13, 0, 7, 9, 1$. We see that this generator has a period of 10.

6.3.3 Number Theoretic Generators

We now provide brief descriptions of many pseudo-random number generators that rely on various number-theoretical functions [18, pages 119–122].

1/P Generator

This generator is described by the parameters P and d. We expand the rational

$$\frac{1}{P} = .d_0 d_1 d_2 \ldots d_j d_{j+1} \ldots$$

in base d. As a seed we take a position j as the initial digit. Thus the pseudo-random sequence $x_0, x_1, \ldots$ is given by $x_n = d_{j+n}$.

Blum, Blum and Shub cryptanalysed this generator and showed that given d and any sequence of length at least $\lceil \log_d(2p^2) \rceil$ generated by such a generator the seed and p could be recovered in polynomial time in p.

Power Generator

Given parameters d and N and a seed x_0 we can create a pseudo-random generator as follows:

$$x_{n+1} \equiv (x_n)^d \pmod{N}.$$

Two special cases have been studied in more depth.

The first case occurs when $N = p_1 p_2$ is a product of two distinct primes and $(d, \phi(N)) = 1$, where $\phi(N)$ denotes Euler's totient function. In this case the map $x \to x^d \pmod{N}$ is one-to-one on $\mathbb{Z}_N^*$, the group of multiplicatively invertible elements (mod N). This operation is the encryption operation of the RSA public-key cryptosystem, where d and N are publicly known. This special case of the power generator is often called the RSA generator.

The second special case occurs when $N = p_1 p_2$ is a product of two distinct primes with $p_1 \equiv p_2 \equiv 3 \pmod{4}$ and $d = 2$. This is called the Square generator. The constructed mapping,

$$x_{n+1} \equiv (x_n)^2 \pmod{N},$$

is four-to-one on $\mathbb{Z}_n^*$. If we restrict the generator to the domain of quadratic residues this becomes a one-to-one mapping. With the restricted domain this is in fact the Blum-Blum-Shub generator, which is discussed in further detail in §6.3.5.

Discrete Exponential Generator

Given parameters g and N and a seed x_0 we can create a pseudo-random generator as follows:

$$x_{n+1} \equiv g^{x_n} \pmod{N}.$$

When N is chosen to be an odd prime and g is chosen to be a primitive root (mod p) then the problem of recovering the seed is essentially the discrete logarithm problem, for which there is no known efficient solution.

6.3.4 Combining PRNGs

Pseudo-random number generators can be constructed from simpler ones using different "mixing" operations.

Hashing

Given $\{x_n\}$ binary strings of k bits, and $H : \{0,1\}^k \to \{0,1\}^l$ with $l < k$ a fixed function, then H is referred to as a hash function. We can then define $z_n = H(x_n)$. The choice of H can make the sequences z_n more secure.

A simple hash function, known as the truncation operator, is given by:

$$T(x) = \lfloor 2^{-j} x \rfloor \pmod{2^l}.$$

This hash function, parameterised by j and l, extracts a string of l bits starting j bits from the least significant bit of a string of k bits. Knuth suggested applying this hash function to the bits generated by a linear congruential generator so as to keep only the high order half of the bits and thus increase apparent randomness. Unfortunately, in this case the security was not increased and Reeds cryptanalysed this system in 1979.

Composition

More than one source of pseudo-random bits can also be combined to create a single resultant bit stream with an increased apparent randomness. In general, if $\{x_n\}$ and $\{y_n\}$ are sequences of k-bit strings, let $* : \{0,1\}^k \times \{0,1\}^k \to \{0,1\}^k$ be a binary operation and set $z_n = x_n * y_n$. Depending on the choice of $*$ this can increase the security of the random bits produced.

6.3.5 Blum-Blum-Shub Generator

The Blum-Blum-Shub (BBS) Generator is one of the simplest and most efficient PRBGs that has been shown to be secure. Strictly speaking the BBS generator is only secure if there is no efficient way to factor large composite numbers.

Simply put, the BBS generator relies on the repeated squaring of a seed number (modulo a parameter, n) and then extracting the least significant bit of the result at each iteration. For a more precise definition of the generator and a discussion of it's security we will need to briefly recall theories pertaining to quadratic residues [4, pages 178–190].

Quadratic residues arise from the study of congruences of the form:

$$x^2 \equiv a \pmod{p} \tag{6.2}$$

where p is an odd prime and $a \not\equiv 0 \pmod{p}$.

Definition 6.6 (Quadratic Residue) *If, for a given odd prime* p *and an integer* a*, there is a solution to (6.2) then* a *is said to be a quadratic residue* mod p*. If there is no solution to (6.2) then* a *is said to be a quadratic non-residue* mod p*.*

As an example, consider finding all the quadratic residues modulo 7. To do this we square all the numbers $1, 2, \ldots, 6$ and reduce mod 11:

$$1^2 \equiv 1, \quad 2^2 \equiv 4, \quad 3^2 \equiv 2, \quad 4^2 \equiv 2, \quad 5^2 \equiv 4, \quad 6^2 \equiv 1 \pmod{7}.$$

Consequently the quadratic residues mod 7 are $1, 2, 4$, and the quadratic non-residues are $3, 5, 6$.

It can be shown that there are always exactly $(p-1)/2$ quadratic residues modulo p. Similarly, there are exactly $(p-1)/2$ quadratic non-residues.

For the study of quadratic residues it is useful to introduce the following symbol.

Definition 6.7 (Legendre's symbol) *Given an odd prime* p *and* $a \not\equiv 0 \pmod{p}$*, the Legendre's symbol,* $(a|p)$ *is defined by:*

$$(a|p) = \begin{cases} +1 & \text{if } a \text{ is a quadratic residue mod } p, \\ -1 & \text{if } a \text{ is a quadratic non-residue mod } p. \end{cases}$$

If $a \equiv 0 \pmod{p}$ *then define* $(a|p) = 0$*.*

Recall that Fermat's Little theorem states that $a^p \equiv a \pmod{p}$ for any prime p and any integer a. From this it follows that $a^{p-1} \equiv 1 \pmod{p}$ if $p \not| \ a$. And since

$$a^{p-1} - 1 = (a^{(p-1)/2} - 1)(a^{(p-1)/2} + 1)$$

we see that $a^{(p-1)/2} \equiv \pm 1 \pmod{p}$. A result known as Euler's criterion shows that the sign depends on whether or not a is a quadratic residue modulo p.

Theorem 6.8 (Euler's Criterion) *Given* p*, an odd prime then for all* a *we have*

$$(a|p) \equiv a^{(p-1)/2} \pmod{p}.$$

We now give two simple results for evaluating $(-1|p)$ and $(2|p)$ based on Euler's criterion.

Theorem 6.9 ($(-1|p)$) *For every odd prime* p *we have*

$$(-1|p) = (-1)^{(p-1)/2} = \begin{cases} +1 & \text{if } p \equiv 1 \pmod{4}, \\ -1 & \text{if } p \equiv 3 \pmod{4}. \end{cases}$$

Theorem 6.10 ($(2|p)$) *For every odd prime* p *we have*

$$(2|p) = (-1)^{(p^2-1)/8} = \begin{cases} +1 & \text{if } p \equiv \pm 1 \pmod{8}, \\ -1 & \text{if } p \equiv \pm 3 \pmod{8}. \end{cases}$$

We now describe another result that is useful for the study of quadratic residues. The quadratic reciprocity law states that if p and q are distinct odd primes, then $(p|q) = (q|p)$ unless $p \equiv q \equiv 3 \pmod{4}$, in which case $(p|q) = -(q|p)$. This result is commonly presented by the following theorem attributed to Legendre.

Theorem 6.11 (Quadratic Reciprocity Law) *Given distinct odd primes* p *and* q *we have*

$$(p|q)(q|p) = (-1)^{(p-1)(q-1)/4}.$$

Finally, we introduce the generalisation of the Legendre symbol, the Jacobi symbol.

Definition 6.12 (Jacobi symbol) *Let* n *be an odd positive integer with the prime factorisation,* $n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$*. Let* a *be a positive integer. Then the Jacobi symbol* $(a|n)$ *is defined by*

$$(a|n) = \prod_{i=1}^{k} (a|p_i)^{e_i}.$$

Note that $(a|n)$ can take on the values 1,-1 or 0. $(a|n) = 0$ if and only if $(a, n) > 1$. If a is a quadratic residue modulo n then $(a|n) = 1$, however the converse is not necessarily true. If $(a|n) = 1$ but a is not a quadratic residue then a is called a pseudo-square.

In the study of the BBS generator, we will consider values of n of the form $n = pq$ with p and q distinct odd primes. Because we are only considering this special case, it is useful to present the following meaning of the Jacobi symbol when n is the product of exactly two distinct odd primes.

Definition 6.13 (Jacobi symbol for $n = pq$) *Let* $n = pq$ *where* p *and* q *are distinct odd primes. Let* a *be a positive integer. Then*

$$(a|n) = \begin{cases} 0 & \text{if } (a, n) > 1, \\ 1 & \text{if } (a|p) = (a|q) = 1 \text{ or } (a|p) = (a|q) = -1, \\ -1 & \text{if } (a|p) = -(a|q). \end{cases}$$

With this brief overview of the quadratic residues, Legendre's symbol and the Jacobi symbol it is now possible to give a formal definition of the BBS generator.

ALGORITHM: **(Blum-Blum-Shub Generator)**:
Let p and q be two $(k/2)$-bit primes such that $p \equiv q \equiv 3 \pmod 4$. Define $n = pq$. Let $QR(n)$ denote the set of quadratic residues modulo n.

Choose a seed s_0 to be any element in $QR(n)$. Then for $i \geq 0$, define

$$s_{i+1} = s_i^2 \bmod n,$$

and define

$$f(s_0) = (z_1, z_2, \ldots, z_l),$$

where

$$z_i = s_i \bmod 2$$

and $1 \leq i \leq l$. Then f is a (k, l)-PRBG known as the Blum-Blum-Shub generator. □

Based on our discussion of quadratic residues we can now make some observations about the choice of the parameters for the BBS generator. The constraint, $p \equiv q \equiv 3 \pmod 4$, ensures that $(-1|p) = -1$ and $(-1|q) = -1$. That is, the constraint ensures that -1 is a quadratic non-residue modulo both p and q. This in turn implies that for any given quadratic residue y, exactly one of its four square roots is itself a quadratic residue [18, page 120]. In other words, this constraint ensures that the mapping $x \rightarrowtail x^2 \bmod n$ used in the BBS-generator is a one-to-one mapping and thus defines a permutation on $QR(n)$, the set of quadratic residues modulo n.

Cryptanalysis

We now discuss the security of the BBS-generator. We show that, in the setting of ϵ-distinguishers, the BBS generator is most likely secure because if it is not secure then it is possible to construct a polynomial-time Monte Carlo algorithm for deciding whether or not a number is a quadratic residue and it is commonly believed that such algorithms do not exist.

To start we assume that there is an ϵ-distinguisher for the (k, l)-BBS generator. Then by a parallel to the notion of ϵ-next bit predictors, there exists an ϵ-previous bit predictor. Utilising this previous-bit predictor we can construct an algorithm that distinguishes quadratic residues from pseudo-squares.

ALGORITHM: **(quadratic residue/pseudo-square distinguisher)**:
Input: $x \in \mathbb{Z}_n^*$ with $(x|n) = 1$

1. **let** $s_0 = x^2 \bmod n$.
2. compute $z_0 = s_0 \bmod 2$.
3. use the BBS Generator to compute $z_1, \ldots, z_{l-1}$ from the seed s_0.
4. use the ϵ-previous bit predictor to predict z, the bit that should come before $z_0, \ldots, z_{l-1}$.
5. **if** $(x \bmod 2) = z$ **then**
6. **return** "x is a quadratic residue."
7. **else**

8. **return** "x is a pseudo-square."

□

Now the ϵ-previous bit predictor guesses correctly with a probability of at least $1/2+\epsilon$ and as a result the above algorithm correctly classifies x with a probability of at least $1/2+\epsilon$. To see this, we note that the observations mentioned previously about the constraint $p \equiv q \equiv 3 \bmod 4$ imply that if $(x|n) = 1$ then the unique quadratic residue that is a square root of $s_0 = x^2$ is x if x is a quadratic residue, and is $-x$ if x is a pseudo-square. Furthermore, $(-x \bmod n) \bmod 2 \neq (x \bmod n) \bmod 2$, so the output of the algorithm is correct if and only if the previous-bit predictor guessed correctly.

Without going into the details, we note that the above algorithm can be used to construct a Monte Carlo algorithm with an error of at most $1/2 - \epsilon$. This Monte Carlo algorithm can in turn be iterated a number of times to produce a prediction, based on the major result, that has an error probability of at most δ for any $\delta > 0$.

Hence, we have shown that the following set of implications exist:

The (k, l)-BBS generator can be ϵ-distinguished from l random bits
$\Downarrow$
There exists a (ϵ/l)-previous bit predictor for the (k, l)-BBS generator
$\Downarrow$
There exists a distinguishing algorithm for quadratic residues that is correct with probability at least $1/2 + \epsilon$
$\Downarrow$
There exists a Monte Carlo algorithm for quadratic residues having error probability at most $1/2 - \epsilon$
$\Downarrow$
There exists a Monte Carlo algorithm for quadratic residues having error probability at most δ, for any $\delta > 0$

But as noted in the beginning of the section, it is considered highly unlikely that there exists a polynomial-time Monte Carlo algorithm for identifying quadratic residues with a small error probability. Thus, it is highly unlikely that we will be able to construct an ϵ-distinguisher for the BBS generator. Hence the BBS generator is considered to be secure.

6.4 Conclusion

Pseudo-Random Number Generators form an important part of almost all cryptosystems. One has to choose a PRNG which meets the needs of the system it is to be used in. Among others this includes factors such as efficiency and security. If the PRNG is inadequate in efficiency, the cryptosystem may take to long to perform its task. In many cases, if the PRNG can be broken then the entire cryptosystem will also be compromised.

As an example, the SSH computer networking protocols provide mechanisms for secure communication between networked computers. The data to be transfered is encrypted using a symmetric algorithm. To set up the keys for the symmetric algorithm, either RSA public-key encryption is used, or DSA is used. DSA is the favoured solution but needs to be used with care because it requires a good source of random numbers. Because such sources are not necessarily available to all computing platforms, some developers of SSH software implementations have chosen not to make

the DSA key exchange available for fear that the poor random source could be used to compromise communications.

As with any component of a cryptosystem, the PRNGs need to be constantly analysed for potential flaws. The PRNG implemented in the OpenSSL software package (version 0.9.6a), which provides a number of cryptographic components, was discovered to be susceptible to attack. The adversary could reconstruct the internal state of the PRNG from a couple of hundred 1-byte requests. In this case it is not likely that the attack could be made practical, but since a large number of software packages rely on OpenSSL for their cryptographic primitives it is always possible that one of them would be susceptible to attack.

Chapter 7

Odds and Ends

The previous sections focus primarily on encryption functions. However, encryption functions form only one part of a complete cryptosystem. A complete cryptosystem may require communication protocols, message digests, authentication and identification schemes, key distribution and other components.

This chapter provides a brief look at some of these components and related issues. Included are also some general techniques that can be used to attack cryptosystems.

7.1 Hash Functions

"Somehow the verb 'to hash' magically became standard terminology for key transformation during the mid-1960s, yet nobody was rash enough to use such an undignified word publicly until 1967."– D.E. Knuth [22].

Hashing is a general term given to key transformations that map a larger domain (possibly with infinite cardinality) into a smaller range. The general problem is 'to hash' a block of data to create a small key that can be used to identify the data [28]. Depending on the application, the hash function will need to exhibit different properties (e.g. database applications require a fixed length resultant hash that minimises collisions and is very efficient to calculate).

In cryptography, hashes are used to create message digests (MD) and message authentication codes (MAC). A MD is a fixed length hash calculated from an arbitrary length message. In many situations, the result can be used as a representation of the message. A MAC is similar to a MD with the addition that its creation includes the use of a key. Thus, to recalculate and verify a MAC, the same key is required. Message Digests are used with digital signature algorithms (see §7.2) to improve efficiency and to enable the use of signature algorithms that only operate on fixed length input. For example, DSA operates on a 160-bit input to create a 320-bit signature and the DSS recommends using SHA to create a 160-bit MD from the original message before signing using DSA.

For digital signatures to be secure, the hashes used must exhibit certain properties. If two messages hash to the same value then they will both have the same signature and a valid signature for one of the documents will automatically be valid for the other document. Now, if an opponent intercepts a message and its signature, and is then able to construct a second document with the same hash, the opponent could then claim that the second document was also signed by the original entity.

The variations on the above attack lead to the following requirements for hash algorithms that are to be used in cryptosystems [31, page 233].

Definition 7.1 (Weakly Collision Free) *A hash function is said to be weakly collision free if, given a message, it is computationally infeasible to find a second message that hashes to the same message digest.*

Definition 7.2 (Strongly Collision Free) *A hash function is said to be strongly collision free if it is computationally infeasible to find two messages that hash to the message digest.*

Definition 7.3 (One-Way) *A hash function is said to be one-way if given a message digest it is computationally infeasible to find a message with a hash equal to the given message digest.*

7.1.1 The Birthday Attack

As with any cryptosystem, the simplest attack is an exhaustive search. For a hash algorithm to be secure it must be computationally infeasible to carry out any exhaustive search. The birthday attack is a probabilistic attack that can be used to attempt to find two messages that hash to the same message digest – that is, the birthday attack attempts to compromise the strongly collision free property of a hash function.

The birthday attack is named after the so-called birthday paradox which asserts that in a group of 23 people there is a probability of just over 50% that at least two people share a birthday. Clearly this is not paradoxical but it is most likely to be counter-intuitive.

The birthday attack states that if a hash function produces n distinct message digests, then a randomly chosen set of approximately

$$k \approx 1.17\sqrt{n} \tag{7.1}$$

messages will have a $1/2$ chance of containing at least two message that hash to the same message digest. We will sketch the derivation of this result below.

With respect to the birthday paradox, the set of all people can be interpreted to be the set of messages, and the 365 days of year can be interpreted as the set of message digests. Hence, $k \approx 1.17\sqrt{365} \approx 22.35$ leads us to the result that a group of 23 people has a probability of just over 50% of containing a duplicate birthday.

When applied to hash functions, (7.1) indicates that a 40-bit hash ($n = 2^{40}$) would be insecure, since, by calculating the message digest of slightly more that 2^{20} (about 1.2 million) messages, a collision could be found with a probability of $1/2$. With current computational resources this could be achieved quite quickly.

For this reason, it is expedient to use a hash of at least 128 bits. This would require more than 2^{64} hashes for the probabilistic attack to be likely to succeed in its search for a collision. In fact, the DSS works with a 160-bit hash to thwart future improvements in computational resources.

Derivation of the Birthday Attack Formula

The birthday attack consists of hashing a set of k messages and checking for duplicate message digests. These k message digests form a subset of the n possible distinct message digests that the hash function can create. We will first derive a formula for the probability of a collision occurring. Using the known number of different message digests and choosing a desired probability, ϵ, of finding a collision, we can calculate an estimate of the number of messages we would likely need to test, to find a collision.

To find the probability of a collision occurring within a set of k randomly chosen messages, we note that it is simpler to calculate the inverse probability – that is, we calculate the probability that k randomly chosen message digests are all distinct. There are n^k total number of ways of

choosing k message digests from a set of n message digests. To choose k distinct message digests, we see that there are n possibilities for the first choice, $n-1$ for the second, and so forth. Thus, k distinct message digests can be chosen in $\frac{n!}{(n-k)!}$ ways. The probability of choosing k message digests with no duplicates would then be

$$\frac{1}{n^k} \cdot \frac{n!}{(n-k)!} = \prod_i^{k-1} \left(\frac{n-i}{n} \right) = \prod_i^{k-1} \left(1 - \frac{i}{n} \right).$$

The series expansion for e^{-x} is as follows:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} \ldots$$

This implies that if x is a small real number then $1 - x \approx e^{-x}$. In the setting of hash functions, n will be large compared to k and thus we can put forward the following estimate for the probability of no collisions:

$$\prod_i^{k-1} \left(1 - \frac{i}{n} \right) \approx \prod_i^{k-1} e^{\frac{-i}{n}} = e^{\frac{-k(k-1)}{2n}}.$$

We then obtain the following approximation for the probability, ϵ, of there being at least one collision

$$\epsilon \approx 1 - e^{\frac{-k(k-1)}{2n}}.$$

It is now possible to solve for k in terms of n and ϵ.

$$\begin{array}{rcl} e^{\frac{-k(k-1)}{2n}} & \approx & 1 - \epsilon \\ \frac{-k(k-1)}{2n} & \approx & \ln(1-\epsilon) \\ -k(k-1) & \approx & 2n \ln(1-\epsilon) \\ k^2 - k & \approx & 2n \ln \frac{1}{1-\epsilon} \end{array}$$

Since k is large we can ignore the $-k$ term and obtain

$$k \approx \sqrt{2n \ln \frac{1}{1-\epsilon}}.$$

If we let $\epsilon = \frac{1}{2}$ then we get

$$k \approx 1.17\sqrt{n}.$$

So, by calculating the message digest of a little more than $\sqrt{n}$ elements there is a 50% chance of finding a collision.

7.1.2 Extending Hash Functions

The basic hash function usually only works on a fixed length input. However, as mentioned above, many applications require that the hash function can operate on an arbitrary length input. Thus, it would be advantageous to be able to take a secure hash function that only operates on input of a given fixed length and extend it to create a hash function that operates on arbitrary length data. The difficulty with this is proving that the extended version is also secure, in the sense that it is strongly collision free and one-way [31, page 241].

The following algorithm demonstrates how to extend a hash function $h(x)$ so as to create $h^*(x)$ that is secure and operates on arbitrary length input. The algorithm is based on work done by Damgård but we do not present the proof showing that h^* is secure if h is secure.

ALGORITHM: **Hash Extension**:
Let $h : (\mathbb{Z}_2)^m \to (\mathbb{Z}_2)^t$ be a strongly collision free hash with $m \geq t + 1$. The following algorithm constructs a strongly collision free hash function $h^* : X \to (\mathbb{Z}_2)^t$, where X is the set of arbitrary length binary strings given by

$$X = \bigcup_{i=m}^{\infty} (\mathbb{Z}_2)^i.$$

The notation, $x \parallel y$, is used to denote the concatenation of the two binary strings x and y. Given an n-bit ($n > m$) string, $x \in X$, we can represent x as a concatenation of strings x_i

$$x = x_1 \parallel x_2 \parallel \ldots \parallel x_k,$$

where the length of each of the $x_1 \ldots x_{k-1}$ is $m - t - 1$ and the length of x_k is $m - t - 1 - d$, with $0 \leq d \leq m - t - 2$. Clearly

$$k = \left\lceil \frac{n}{m - t - 1} \right\rceil.$$

1. **let** $y_i = x_i$ **for each** $i \in [1, k - 1]$.
2. **let** $y_k = x_k \parallel 0^d$ (0^d denotes the string of d 0s).
3. **let** y_{k+1} be the binary representation of d
   (padded to the right with zeros to obtain an $(m - t - 1)$-bit string).
4. **set** $g_1 = h(0^{t+1} \parallel y_1)$.
5. **for** $i \in [1, k]$ **{**
6. **set** $g_{i+1} = h(g_i \parallel 1 \parallel y_{i+1})$.
   }
7. **define** $h^*(x) = g_{k+1}$.

□

7.1.3 Cryptographic Hashing Algorithms

For the purposes of cryptographic applications it would be beneficial if the hash functions used could be proven to be secure, under reasonable computational assumptions. The Chaum-van Heijst-Pfitzmann Hash Function is one such example, the security of which is based on the discrete logarithm problem. The problem with most provably secure hash functions is that they are insufficiently efficient for practical use.

For practical applications, cryptosystems usually use hash functions that have not been proven to be secure, yet have also withstood all cryptanalytic attempts.

One method for constructing a hash function is to use an existing private-key cryptosystem. The basic idea is to break the message that is to be hashed into a chain of blocks equal in length to the block size of the encryption function and to iteratively encrypt each block using the previous

output as the encryption key. If the block size of the encryption function $e_k(y)$ is n then the message bit string x would be broken into n-bit blocks such that

$$x = x_1 \parallel x_2 \parallel \ldots \parallel x_s.$$

We assume that the key, k, is also an n-bit block. We then define g_i, for $1 \leq i \leq s$, recursively by

$$g_i = f(x_i, g_{i-1}),$$

where f uses the encryption function as part of its definition, and g_0 is defined to be a fixed value initial vector. The message digest created is given by $h(x) = g_s$. Although many such schemes have been shown to be insecure, the following forms of f seem to be secure:

$$\begin{aligned}
f(x_i, g_{i-1}) &= e_{g_{i-1}}(x_i) \oplus x_i, \\
f(x_i, g_{i-1}) &= e_{g_{i-1}}(x_i) \oplus x_i \oplus g_{i-1}, \\
f(x_i, g_{i-1}) &= e_{g_{i-1}}(x_i \oplus g_{i-1}) \oplus x_i, \\
f(x_i, g_{i-1}) &= e_{g_{i-1}}(x_i \oplus g_{i-1}) \oplus x_i \oplus g_{i-1}.
\end{aligned}$$

Many algorithms designed specifically to create one-way hash functions have also been proposed. Most of these algorithms exhibit similarities to private-key block encryption algorithms. For this reason, methods such as differential cryptanalysis have often been successfully employed in breaking the hash functions (either by finding collisions faster than the birthday attack, or by finding a message that hashes to a given value faster than an exhaustive search).

We will not present complete descriptions of the algorithms here, but provide a short list of some of the more commonly used algorithms and some of their attributes [29, pages 432–446].

Snefru: This hash algorithm was designed by Ralph Merkle and creates either a 128-bit or a 256-bit message digest from an arbitrary length message. Biham and Shamir used differential cryptanalysis to successfully break the algorithm. A pair of messages with the same message digest can be found in $2^{28.5}$ operations for the three pass version of Snefru (as opposed to 2^{64} using the birthday attack). For a given message digest, a corresponding message can be found in 2^{56} operations for the three pass version of Snefru (as opposed to 2^{128} for a brute for attack). Merkle thus recommends using an eight pass version, but this is much slower than MD5 or SHA.

N-Hash: This hash algorithm was designed by researchers at Nippon Telephone and Telegraph. N-Hash produces a 128-bit message digest using a randomising function similar to the FEAL encryption algorithm. This algorithm has also been broken by Biham and Shamir using differential cryptanalysis. Bert den Boer also found a method for producing collisions.

MD4: This is a 128-bit hash algorithm designed by Ron Rivest. The algorithm was designed with attention given to security as well as to practical implementation. It is fast, simple, requires small data structures and favours little-endian architectures (such as Intel processors). No known cryptanalytic attack has been extended to the full algorithm, but in separate attempts the first two rounds and the last two rounds have been successfully attacked.

MD5: To prevent the attacks that MD4 is susceptible to Ron Rivest created an improved version called MD5. This version uses 4 rounds and introduces a different fixed value in each step of each round (MD4 uses the same fixed value throughout a given round). MD5 is more secure but work by Bert den Boer and Bosselaers has been able to produce collisions.

MD2: This is another 128-bit one-way hash function designed by Ron Rivest. It depends on a random permutation of bytes for its security. There are no known weaknesses in MD2 but it is slower than most other hash function.

SHA: This algorithm was designed by NIST along with the NSA. The SHA algorithm is a cousin to MD5 in that it is based on the MD4 algorithm. It also improves on MD4 to overcome the weaknesses of MD4. In addition, it creates a 160-bit message digest and is thus more resistant to brute force attacks. There are no known cryptographic attacks against SHA and it is probably the best choice for many practical implementations.

Ripe-MD: This algorithm was developed for the European Community's RIPE project. It is also a variation of MD4, but runs two instances of the variant in parallel and combines the results for each block. This seems to make the algorithm highly resistant to cryptanalytic efforts.

HAVAL: This is an interesting hash algorithm that can be used to produce message digests of variable length (128, 160, 192, 224, or 256 bits). The actual implementation borrows many ideas from the MD5 algorithm.

7.2 Digital Signatures

Cryptography is not only used to provide encryption so as to keep the contents of the message secret. Messages must also be proven to be authentic. With hard copies of messages this can be achieved via a handwritten signature inscribed onto the paper that the message is printed on, or it can be achieved using a wax seal and a stamp.

Such signatures meet three important requirements

- The signature can be used to verify that a particular person signed the document and thus assumes responsibility for the document.
- The signature is created in a manner that makes it difficult to forge. A handwritten signature is unique to the author and an impression created by a stamp is unique to the stamp used.
- The signature becomes part of the document as the ink or wax would be difficult to remove without it being noticeable.

Unfortunately, traditional schemes such as handwritten signatures or impressions created by stamps can often be forged sufficiently well so as to fool the verifier of the document. Furthermore, even if the signature cannot be forged, it often suffices to alter the document – as is the case with cheque fraud, where the recipient of the cheque is fraudulently changed after it has been written and before it has been cashed. Additionally, such schemes of authenticating documents can only be applied to documents created on paper or other physical media.

If there is potential for the signature to be forged or the signed documents to be later altered, or the document only exists in a digital form, then we need to employ new techniques to achieve the same effect. To do this, we use cryptographic techniques to create digital signatures [29, pages 483–502] and [31, pages 202–231].

Digital signature schemes provide a mechanism for creating the equivalent of a physical signature for digital data. That is, one creates a digital signature, which is a block of data that can be used to verify that a particular identity is responsible for the digital document. Again, the digital signature must be linked directly to the given document and must be difficult to forge.

Table 7.1: Some Digital Signature Algorithms

▷ ElGamal Signature Scheme
▷ RSA Signature Scheme
▷ Digital Signature Algorithm
▷ GOST Digital Signature Algorithm
▷ Ong-Schnorr-Shamir Signature Scheme
▷ ESIGN

Many different digital signature algorithms have been developed. A list of some of the better known algorithms is given in Table 7.1. To better understand how digital signatures are implemented, we shall briefly discuss the Digital Signature Algorithm that is used as part of the Digital Signature Standard, which was adopted by the United States Federal departments in 1994.[1]

7.2.1 Digital Signature Standard

DSS has been adopted by the United States and is a de-facto standard elsewhere in the world. DSS is thus commonly used and is a useful scheme to discuss. The Digital Signature Algorithm is a modification of the ElGamal algorithm and we will thus first introduce the ElGamal algorithm.

ElGamal

ElGamal introduced this algorithm in 1984 [14]. The algorithm uses a public/private key pair. The private key (together with the public key) is used to sign a document and the public key alone is used to verify the authenticity of the signature of the document.

First we choose a prime p. Then we generate two random positive integers, α and a, such that both α and a are less then p and α is a primitive element of $\mathbb{Z}_p^*$. We define $\beta = \alpha^a \bmod p$. The values p, α and β together form the public key while a forms the private key. The message is then represented as an integer, m, such that $0 \leq m \leq p-1$. We can now calculate the signature of m, which consists of a pair of integers (γ, δ) such that $0 \leq \gamma, \delta < p-1$.

The quantities γ and δ are chosen such that the equation

$$\alpha^m = \beta^\gamma \gamma^\delta \bmod p \tag{7.2}$$

is satisfied. To find values for γ and δ, we first choose a random integer, k, uniformly from the range $[0, p-1]$ such that $(k, p-1) = 1$ (k is relatively prime to $p-1$). γ is then defined by

$$\gamma = \alpha^k \bmod p.$$

We can thus rewrite (7.2) as

$$\alpha^m = \alpha^{a\gamma}\alpha^{k\delta} \bmod p. \tag{7.3}$$

[1] There is often some confusion between the usage of "Digital Signature Algorithm" (DSA) and "Digital Signature Standard" (DSS) in the literature. DSA refers to the core algorithm used in the Digital Signature Standard and is a modification of the ElGamal algorithm. DSS, however, is a complete standard that incorporates the DSA as well as recommended algorithms for creating the moduli used to form the keys, and guidelines for the general implementation and use of the signature scheme.

Through an application of Fermat's little theorem (*For any integer* α *and any prime* p *we have* $\alpha^p \equiv \alpha \pmod{p}$), we see that the following relation must hold true

$$m = a\gamma + k\delta \bmod (p-1). \tag{7.4}$$

Since k was chosen relatively prime to $p-1$, (7.4) can be used to find a solution for δ (the Extended Euclidean Algorithm can be used to solve for δ). We have now formed the signature (γ, δ).

To verify that the signature is valid, we use the public values p, α and β together with the message m and the signature (γ, δ) to independently calculate both sides of (7.2) and check that the results are equal. The security of the scheme relies on the privacy of a. a can not be recovered from the publicly known values of p, α and β, and the relation $\beta = \alpha^a \bmod p$, because of the difficulty of solving the discrete logarithm problem.

Digital Signature Algorithm

As we have seen above, the security of the ElGamal Scheme is based on the difficulty of solving the Discrete Logarithm problem. Thus, to achieve a practical level of security the modulus, p, should be chosen to be at least 512 bits in length – many people advocate using a 1024 bit modulus so as to provide suitable security for the foreseeable future. The requirement of a large modulus presents a practical difficulty since the signature generated by a 512-bit modulus would be 1024 bits long, and likewise a 1024-bit modulus would create a signature that is 2048 bits long. Signatures of such length are unwieldy for use in practical implementations (e.g. smart cards).

DSS has been designed as a modification of the ElGamal signature algorithm which addresses the issue of the length of the signature. The result is a system which can be used to sign a 160-bit message, creating a 320-bit signature. We now briefly present the modifications made to the ElGamal algorithm to produce the DSS system and then provide a description of DSA.

The main innovation of DSA is to carry out calculations in a subgroup of $\mathbb{Z}_p^*$ of size 2^{160}, which achieves the necessary reduction in the length of the signature produced. This relies on the assumption that finding discrete logarithms in this subgroup is secure – this is believed to be the case, and there is no strong evidence against this.

Like ElGamal the basic workings of DSA can be derived from the verification condition. The DSA condition is very similar but the β^γ term is on the left hand side:

$$\alpha^m \beta^\gamma \equiv \gamma^\delta \pmod{p}. \tag{7.5}$$

Again p is chosen to be a large prime, α and a are integers in the range $[0, p)$ (though we will see that a further constraint on α will be needed) and β is defined by $\beta \equiv \alpha^a \pmod{p}$.

As mentioned above, DSA endeavours to carry out the calculations in a subgroup of $\mathbb{Z}_p^*$. This is achieved by introducing a second prime, q, and performing calculations modulo q. As we have seen with ElGamal, γ and δ form the digital signature. We hope to reduce these modulo q. Hence, we choose to define γ by

$$\gamma = (\alpha^k \bmod p) \bmod q, \tag{7.6}$$

where k is chosen at random uniformly from the range $[1, q-1]$ (note that k is kept secret).

We now wish to use (7.5) to solve for δ. However, we recall that γ is reduced modulo q. To enable us to work with this, we first rewrite (7.5) as

$$\alpha^{m\delta^{-1}} \beta^{\gamma\delta^{-1}} \equiv \gamma \pmod{p}. \tag{7.7}$$

This can be done if $\delta^{-1} \bmod (p-1)$ exists, which is the case if $(m + a\gamma, p-1) = 1$. (7.7) enables us to reduce both the left hand side and the right hand side modulo q.

To complete the transformation to working modulo q, we must also be able to reduce the exponents of (7.7) modulo q. This can be achieved by adding an additional constraint to the choice of α, namely choose α to be a qth root of unity modulo p (this additionally implies that $q|\phi(p)$ and thus $q|p-1$ [4, page 204]). By definition, β and γ will also be qth roots of unity. By substituting for β and γ in (7.7) and using the fact that α is a qth root of unity we can see that the following relation will enable us to solve for δ [2]

$$m\delta^{-1} + a\gamma\delta^{-1} = k \bmod q. \tag{7.8}$$

This completes the description of how the DSA signature is calculated. To verify the signature we show that both sides of (7.7) are equal, where the values are reduce modulo q as is appropriate.

More explicitly, given the public key (consisting of p,q,α and β), the message m and the signature (consisting of γ and δ), we can verify the signature by calculating

$$\begin{aligned} e_1 &= m\delta^{-1} \bmod q \\ e_2 &= \gamma\delta^{-1} \bmod q \end{aligned}$$

and then confirming that the follow relation is true

$$(\alpha^{e_1}\beta^{e_2} \bmod p) \bmod q = \gamma.$$

It should be noted that for any implementation of DSA to remain secure it is not sufficient that the secret key a remains secure. The value chosen for k must also be kept secret. Additionally, no two messages should ever be signed using the same value of k. If the value of k (which used to create a particular signature) is recovered, an adversary would be able to calculate the secret key a. Also, if an adversary acquires two separate messages signed using the same public/private key pair and the same value of k the adversary will be able to calculate the secrete key. These requirements imply that the PRNG used with DSA must be cryptographically secure.

7.2.2 Digital Signatures with Time Stamps

Proving the authenticity of a message is not necessarily sufficient. Often one also needs to prove when the document was created and signed. For this we need to be able to add a time stamp to the signature. There are a few ways in which this can be achieved. Broadly speaking, we can use a trusted third party, or we can use publicly available information that can be undeniably related to a particular region of time (e.g. headings of articles in a daily newspaper would be unique for a given day). When using a trusted third party the time-stamp is appended to a digital signature and the result is signed by the third party. If public information is used, this can be appended to a hash of the message and then the result is signed. Unfortunately the public information only enables one to prove that the document could not have been signed before the specified date, but it could still have been falsely created at a later date. One method for preventing this last problem is to link time-stamped messages to previous time-stamped messages and have subsequent time-stamped messages linked to the current time-stamped message. This then provides a provable window during which the message signature was time-stamped.

[2] Given an integer s such that $\alpha^s \equiv 1 \pmod p$ with α a qth root of unity modulo p implies that $q|s$ which is equivalent to the statement $s \equiv 0 \pmod q$.

7.3 Important Number Theoretic Problems

The security of many cryptosystems (be they hash functions, pseudo-random number generators or encryption algorithms) relies on number-theoretic problems. Although we have not emphasised the results pertaining to such problems, we now present, for completeness, a brief description of the two most important problems. The discussion of methods for solving these problems is beyond the scope of this work.

7.3.1 Discrete Logarithm Problem

Let G be a group and for $g \in G$, let $\langle g \rangle$ be the cyclic subgroup generated by g. Given $a \in \langle g \rangle$, the discrete logarithm problem consists of finding an integer x that satisfies

$$g^x = a.$$

The value x is called the discrete logarithm of a to the base g. The notation $x = \mathrm{ind}_g a$ is often used.

7.3.2 Factorisation Problem

Given a composite integer n, the problem of factoring is to determine the primes p_j and the corresponding exponents e_j such that

$$n = \prod_{j=1}^{s} p_j^{e_j}.$$

7.4 Key Distribution and Key Agreement

Private-key cryptosystems solve the problem of transferring data from one entity to another in a secure manner, thus creating a secure channel. However, to utilise the secure channel both entities need to share a secret key and this secret key needs to be transmitted via a secure channel. This clearly presents a problem. In some cases there are other secure channels that can be used, but often there are no other channels or if there are they are too slow or too costly. For example, a potentially costly and impractical solution would be for the members of the party who wish to communicate, to meet in a soundproof room and exchange a key verbally, this key could then be used for later digital communications.

One solution to the above problem is to use public-key (asymmetric) cryptosystems. Unfortunately most public-key cryptosystems are much slower than private-key systems and unless computational resources are abundant, often cause a bottle-neck for long messages. Furthermore, keys used in public-key cryptosystems often require more storage than their private-key counterparts (Rijndael [3] uses a 160-bit where as RSA is normally used with a 1024-bit key).

Another solution is to use key exchange algorithms. These algorithms can be broadly categorised into two types:

Key Distribution: Here one party chooses the secret key and then conveys it to one or more parties who wish to communicate securely with each other. Generally this is achieved using a central trusted authority that distributes the secret keys.

[3] Rijndael has been adopted by NIST as the Advanced Encryption Standard (AES) which replaces the Data Encryption Standard (DES).

Key Agreement: With key agreement two or more parties carry out calculations using inputs passed over a public channel so as to jointly establish a secret key. These methods emphasis a distributed system that does not require a trusted authority.

Protocols such as Kerberos and Blom's scheme are key distribution algorithms using a trusted authority. A basic system using a trusted authority would generate $\binom{n}{2}$ keys, one for each unique pair of users in a network of n users. Blom's scheme reduces this by using fewer keys but with the proviso that a number of users could work together to acquire information about a subset of keys (the size of the coalition required to attack the scheme can be chosen as required).

The Diffie-Hellman Key Exchange algorithm enables key agreement between two parties without the need for a trusted authority. There are other key agreement protocols, a number of which are modifications or extensions of Diffie-Hellman (e.g. MTI, Hughes, Diffie-Hellman with Three or More Parties).

Many systems also use public-key encryption algorithms (e.g. RSA, ElGamal, Elliptic Curve based systems or Knapsack Ciphers) to distribute a session key.

7.4.1 Diffie-Hellman Key Exchange

The Diffie-Hellman algorithm was the first public-key algorithm ever invented (circa 1976). To use this algorithm, the parties wishing to agree on a key, Alice and Bob, first agree on a large prime, p, and a primitive element of $\mathbb{Z}_p$, α. Both p and α can be publicly known and thus can be passed over an insecure channel.

1. Alice then chooses a large random integer, u, and calculates
$$U = \alpha^u \bmod p.$$
Alice then transmits U to Bob.

2. Bob similarly chooses a large random integer, v, and calculates
$$V = \alpha^v \bmod p.$$
Bob then transmits V to Alice.

3. Finally Alice computes
$$k = V^u \bmod p,$$

4. and Bob computes
$$k' = U^v \bmod p.$$

Now $k = (\alpha^{vu} \bmod p) = (\alpha^{uv} \bmod p) = k'$ and thus can be used as a shared secret. However, unless an adversary can solve the discrete logarithm problem, the publicly known values (α, p, U and V) can not be used to recover the shared secret.

Hughes

With the basic Diffie-Hellman algorithm, both parties must be in communication at the time that the key is agreed upon. In some situations this is a disadvantage, for example Alice may wish to select a secret key and use it to encrypt a message prior to contacting Bob. She could then, at a later time, contact Bob (or any number of people) and send both the encrypted message and exchange the secret key. Hughes modification allows for this.

1. Alice chooses a large random integer u and generates

$$k = \alpha^u \bmod p$$

which she uses as the secret key.

2. At some later time Bob chooses a large random integer v, computes

$$V = \alpha^v \bmod p$$

and sends it to Alice.

3. In turn Alice sends Bob

$$U = V^u \bmod p.$$

4. Bob computes $w = v^{-1}$ and reconstructs the secret key

$$k' = U^w \bmod p.$$

We should have $k = k'$.

Diffie-Hellman and The Man-in-the-middle Attack

An important disadvantage of the Diffie-Hellman Key Exchange algorithm is that it is susceptible to the man-in-the-middle attack. In this attack, when Alice and Bob are exchanging keys the opponent, Oscar, intercepts the communication and impersonates Bob when communicating with Alice and impersonates Alice when communication with Bob. Because the algorithm has no authentication built into it, Alice and Bob will be unaware that they are actually communicating with Oscar. When the protocol is complete Alice, Bob and Oscar will know the shared secret.

To prevent this type of attack, Alice and Bob need to authenticate any communication between themselves and thus prevent Oscar from altering data passed between them. This can be achieved by concurrently using a digital signature algorithm such as DSA. Each block of data transmitted will be sent with the corresponding signature which can then be verified using the appropriate public key.

7.5 Steganography

Steganography is the art and technique of hiding a true message inside some other medium. The intention is to prevent the detection of the mere existence of the message. This is related to subliminal channels.

The commonly described exemplars of steganography include encoding hidden messages using invisible ink, differences in handwritten characters or the number of words in a sentence. The implication is that steganography does not employ a key and the security depends on the secrecy of the algorithm.

For a more recent example, one can conceal digital data in a digital graphical image by using the least significant bit of each byte of the image, thus allowing one to store in the order of 300 kilobytes of data in a 1024×1024 image (where each pixel is represented by a byte for red, green and blue respectively) without noticeably affecting the visual content of the image [29, page 11].

Subliminal channels, however, generally use more cryptographically secure mechanisms. The encoding of the data is protected by a key and the hidden message can only be shown to exist if

the key is known. The most common example is the subtle alteration of signature schemes to hide a message.

This area also relates to water-marking, whereby an artifact (digital or otherwise) is subtly altered so as to prove the authenticity or origin of the artifact. The water-mark is usually designed to have a minimal impact upon the actual artifact. The water-marks are also designed to be difficult to forge. Below are descriptions of two uses of water-marks, one of which is used in physical objects while the other is used in digital data:

Paper Money: During the manufacturing phase, parts of the paper are made to be thinner than the rest of the paper. This creates a pattern in the paper that is noticeable when held up to light. This water-mark serves to prove the authenticity of the cash and is also designed to be difficult to recreate in forgeries.

Digital Music: Water-marks are used in the digital world to tag such data as digital music. The water-mark is not used to prevent illegal copying but rather it is designed to help monitoring of pirating and subsequent law enforcement. The water-marks used in the digital music industry are designed to alter the audio stream in a manner that can be detected if the key is known, but at the same time it does not noticeably degrade the quality experienced by the audience of the music. Furthermore, the digital music water-mark is designed to be difficult to remove and will even be present in a copy that is created by analogue recording.

7.5.1 Subliminal Channels

Some digital signature schemes make it possible to conceal information in the signature of a document. This information can only be recovered by entities privy to the concealment method and concealment keys.

Subliminal channels can be used in numerous ways. As with any technology, some of these are positive and others negative. The most obvious application is to use the subliminal channel as a hidden communication channel for a spy network. Subliminal channels in signatures could be used to tag signatures (and hence the corresponding documents) allowing the documents to be tracked over their lifespan. A similar tag could also be used by governments to "mark" digital cash. A malicious implementation of a signature algorithm could be used to leak information about the secret keys used during signing [29, page 79].

Herein we briefly describe how DSA (see §7.2.1) can be used to create a subliminal channel.

- The first method is very simple. During the construction of the signature, DSA requires the generation of a random 160-bit number k. This number is normally kept secret. If, however, the recipient of the signature knows both the public and the private key used to create a particular signature, it is then possible for the recipient to calculate what value of k was used. Thus, to pass a message (m' say) via the subliminal channel, instead of choosing k at random, k is chosen to be some representation of m' (160 bits of binary data, or an encryption of some text etc.). In this scheme, the value a becomes the private key for the signature as well as the secret key of the subliminal channel.

- DSA can also be used to create a subliminal channel in which the recipient does not need to know the secret key used for creating the signature. This scheme enables a single bit of information to be transmitted subliminally. To use this scheme the entities that wish to communicate using the subliminal channel first decide on a shared secret key, P. P is a randomly chosen prime. When a subliminal message (bit) is to be sent, the sender chooses k

such that δ (recall that γ and δ together form the signature of the message) is a quadratic residue modulo P if a *zero* is to be sent, and chooses k such that δ is a quadratic non-residue modulo P if a *one* is to be sent. The cover document together with its signature (γ, δ) are then transmitted to the recipients. All recipients will be able to verify the signature but the recipients that are intended to receive the subliminal message can use P to check if δ is a quadratic residue or a quadratic non-residue and hence receive a single bit of information. To send more that one bit of information multiple primes, $P_1 \dots P_n$ can be chosen and k can then be chosen such that δ is a quadratic residue or quadratic non-residue relative to each of the respective primes as is required to encode the n-bit subliminal message.

7.6 Secret Sharing

Secret sharing enables one to place a secret message into the care of more than one entity. The secret message can only be recovered if some or all of the participants reveal their share of the secret. If the subset of participants is not authorised to reveal the secret then their shares will not enable them to discover any information about the secret [31, pages 326–357] [29, pages 528–531].

Such schemes might be used to protect the launch code for a nuclear missile. The launch code it divided up amongst a small number of top generals in the military in a manner that if any 3 decide to launch the weapon they can use their shares to reconstruct the secret code.

In general a secret sharing scheme consists of a group of participants, $\mathcal{P}$ and a set of subsets, Γ, of $\mathcal{P}$ that defines groups of participants that can work together to reveal the secret message, M. Γ is called the access structure. The scheme defines how the M is to be partitioned into shares (sometimes called shadows) and distributed amongst the participants.

Definition 7.4 (Perfect Secret Sharing Scheme) *A secret sharing scheme is said to be perfect if given an arbitrary subset, $\mathcal{B}$, of $\mathcal{P}$ we have that M can be reconstructed if and only if $\mathcal{B}$ is an authorised group (that is $\mathcal{B} \in \Gamma$).*

7.6.1 Threshold Schemes

The first secret sharing schemes to be invented and studied form a special case referred to as threshold schemes. With a threshold scheme we have m participants and choose a threshold $t \leq m$. Any t participants can cooperate to recover the secret message. That is, the access structure for a threshold scheme could be defined by

$$\Gamma_T = \{\mathcal{G} \subseteq \mathcal{P} \mid |\mathcal{G}| \leq t\}.$$

This is denoted as a (t, m)-threshold scheme.

Threshold schemes were invented independently in the late 1970s by Shamir and Blakley. Shamir's secrete sharing scheme used polynomials and Lagrange interpolation, whereas Blakley's scheme used a geometric construction.

Shamir's Lagrange Interpolating Polynomial Scheme

Shamir's scheme utilises polynomials over a finite field. Suppose we wish to construct a (t, m)-threshold scheme. We would first choose a random prime, p, such that $p > m$ and p is large enough such that the secret message can be represented by an element in $\mathbb{Z}_p$.

We now construct a random polynomial, $F(x)$, of degree $t-1$ over $\mathbb{Z}_p$ by randomly choosing $t-1$ coefficient $a_j \in (\mathbb{Z}_p \backslash 0)$ and defining $F(x)$ by

$$F(x) = M + \sum_{j=1}^{t-1} a_j x^j \bmod p.$$

To calculate the shadows, we evaluate the polynomial at m distinct points. For example, we could calculate the shadows, $P_1, \ldots, P_m$, by using

$$P_i = F(i) \bmod p \ \forall \ i \in [1, m].$$

The P_i are given to each of the participants and the value of p is made public. The a_j and M are discarded.

Now it is clear that if any t shadows are revealed, say $P_{\pi_1}, \ldots, P_{\pi_t}$, then the system of linear equations

$$\begin{array}{rcl} M + \sum_{j=1}^{t-1} a_j (\pi_1)^j & \equiv & P_{\pi_1} \pmod p \\ M + \sum_{j=1}^{t-1} a_j (\pi_2)^j & \equiv & P_{\pi_2} \pmod p \\ & \vdots & \\ M + \sum_{j=1}^{t-1} a_j (\pi_t)^j & \equiv & P_{\pi_t} \pmod p \end{array}$$

is sufficiently determined to uniquely find the a_j and M. However, if any fewer than t shadows are revealed then no information can be gained about M.

Blakley's Geometric Construction Scheme

Blakley's threshold scheme is an elegant and simple system. To construct a (t, m)-threshold scheme the secret message, M, is encoded as a point in a t-dimensional space. Each shadow is a randomly chosen but distinct $(t-1)$-dimensional hyperplane that includes the point. Clearly, the intersection of any t such hyperplanes will uniquely define the point. However, if only $u < t$ shadows are known then the best we can do is to construct a $(t-u)$-dimensional subspace in which the point is known to reside.

7.7 Time–Memory Trade-offs

When designing a cryptosystem, one wishes to create a system that will thwart all attacks that cryptanalysts can carry out. Some attacks are specific to the type of cryptosystem or even the particular cryptosystem. Other techniques are more general.

The brute-force attack is the simplest general attack, but is also not very effective. In the late 1970s Hellman devised a new general attack that had far reaching implications for what the lower bound of a cryptosystems key size should be.

Hellman's attack, known as a "cryptanalytic time–memory trade-off," was published in 1980 [17]. In its basic form, the attack works as a chosen plaintext attack on block ciphers (it can be modified to provide a ciphertext-only attack). The method can be configured to require $O(N^{2/3})$ operations and require $O(N^{2/3})$ words of memory, where N is the number of possible keys. It also requires a precalculation phase of $O(N)$ operations. When considering an actual implementation, this means that if the precalculation can be completed in the order of one year of computing time, then the attack on a given message will be completed in the order of one day of computing time.

Figure 7.1: Precalculation for Time–Memory Trade-Off

$$\begin{array}{ccccccc} X(1,0) & \rightarrow & X(1,1) & \rightarrow & \dots & \rightarrow & X(1,t) \\ X(2,0) & \rightarrow & X(2,1) & \rightarrow & \dots & \rightarrow & X(2,t) \\ \vdots & & & & & & \vdots \\ X(m,0) & \rightarrow & X(m,1) & \rightarrow & \dots & \rightarrow & X(m,t) \end{array}$$

The basic idea of the time–memory trade-off is precompute a large set of possible states that the cryptosystem could be in and store these on a hard disk. When one wishes to find the key for a plaintext/ciphertext pair, the pair is used to calculate a number of possible states and check for an intersection with the stored set. If there is an intersection then it is possible to work out what the original key was. In many ways time–memory trade-off algorithms are related to dynamic-programming. Dynamic programming can often be used to solve traditionally recursive algorithms more efficiently by applying a caching strategy that saves intermediate results. The more in depth description given below is derived from the explanation given by Stinson [31, page 86].

The description we give is based on an application to DES. We start by choosing a random plaintext, x. We then choose two positive integers, m and t – these represent the memory requirements and the time requirements. Next we choose m 56-bit keys at random, denoted by $X(i,0)$, $1 \leq i \leq m$, and iteratively construct the matrix shown in Figure 7.1 using the function $g(k) : (\mathbb{Z}_2)^{56} \rightarrow (\mathbb{Z}_2)^{56}$. $g(k)$ is based on the encryption function $e_k(x)$. However, the output of $e_k(x)$ is 64 bits long so we use a reduction function R. R can be any function reducing the input from 64-bits to 56-bits (e.g. discarding the first 8 bits). Thus, g is defined by $g(k) = R(e_k(x))$.

Although all the entries in the matrix need to be calculated ($O(mt)$ operations), only the first and last columns need to be stored (that is $2 \times m$ 56-bit values). We denote this two column table by T.

When an actual ciphertext has been obtained for analysis, we use the stored values to test if the correct key, K, is in the matrix. This can be done by taking the ciphertext, y, and iteratively applying g to it. At each step we check if the output $y_j = g^j(y_{j-1}) = g(g(\dots(g(y_1))))$ is in the last column of the table T. $y_1 = R(y) = e_K(x)$ where y and x are known and K is being searched for. If for some y_j there exists an i such that $y_j = X(i,t)$ (i.e. y_j is in the last column of the matrix) then there is a possibility that $K = X(i, t-j)$ (R is not an injection so on average there are $2^8 = 256$ images that could match the reduction of y). Although $X(i, t-j)$ is not stored we can recalculate it using $X(i,0)$.

By analysing the probability of success of the key search algorithm, Hellman showed that if $mt^2 \approx N = 2^{56}$, then the probability of finding a matching key is approximately $0.8mt/N$. The 0.8 arises because the entries in the last column are not necessarily distinct.

To use this method it is suggested that one chooses $t \approx m \approx N^{1/3}$ and constructs $N^{1/3}$ distinct tables using a different reduction function, R, for each one. This results in a storage requirement of $2 \times 56 \times N^{2/3}$ bits and a precomputation time of $O(N)$ operations.

7.7.1 Application to Cellular Phones

More recently, Golić [16] applied time–memory trade-offs to stream ciphers to cryptanalyse the A5 cipher. The A5 cipher is used in GSM cellular phone networks to encrypt the cellular phone traffic and thus protect the over-the-air privacy of voice and data communications.

In 1999 Shamir and Biyrukov enhanced Golić's work to provide practical attack on GSM cellular phone communications [10]. In addition to the time–memory trade-off, this attack combines a number of techniques and exploits a number of weakness of the A5 stream cipher. The result is an attack that requires $O(2^{48})$ operations for preprocessing to create about 80-Gigabytes of data. This stored data can then be used to find the session key for a data stream in about 1 minute on a standard desktop PC. The session key can be used to decrypt the entire conversation. For the attack to be successful, the algorithm requires that about 2 minutes of conversation be intercepted.

7.8 Side Channel Attacks

Generally when one talks about a cryptographic algorithm, one talks about the mathematical context and the logical structures that are manipulated. If the mathematical model of the cryptosystem can be shown to be secure against all known types of attacks then one is usually confident in using the system to protect information.

When a cryptosystem is chosen for a real application, an implementation of the system needs to be constructed. This implementation is usually comprised of special hardware, special software or both.

Since traditional cryptanalysis attacks the mathematical structure of the cryptosystem, it can be applied to any implementation. This is true of attacks such as differential cryptanalysis, linear cryptanalysis, time–memory trade-offs and related-key cryptanalysis.

Recently, however, much work has been done that demonstrates that the specific implementations may also provide the cryptanalyst with useful information that can be used to compromise the system. Examples of measurements useful for attacks include aspects such as power consumption, execution times and network bandwidth usage. Collectively these sources of information are referred to as *side-channels*.

As shown in [21], side-channel attacks can be very effective at compromising algorithms that are traditionally considered secure. Although many traditionally secure cryptosystem are susceptible to theoretical attacks, many such attacks can not be practically implemented, either due to lack of computing resources or lack of data. Side-channel attacks are often sufficiently effective to make practical implementation a possibility.

The advent of side-channel attacks will lead to implementations being reviewed so as to harden them against side-channel attacks. This new strategy of cryptanalysis will also lead to the design of cryptographic algorithms that are more secure against attacks targeted at the implementation. This can be achieved by explicitly including assumptions about the implementation environment when designing the algorithms. The designer would then show that any implementation that fits into the designated class will be secure against a range of known side-channel attacks.

Below we describe, in more detail, some of the known side-channel attacks.

7.8.1 Timing Attacks

Kocher was among the first researchers to study side-channel attacks [24]. Kocher's first attack works against modular exponentiation and could be used to break cryptosystems that rely on the difficulty of the discrete logarithm problem for their security (e.g. Diffie-Hellman key exchange).

In such systems one computes values of the form $R = y^x \bmod n$, where both n and y are passed over insecure public channels. The attacker wishes to recover the secret key x.

The attack relies on differences in execution time when calculating modular exponentiation with different parameters. To see that there would be a difference, consider the following pseudo-code for the repeated squaring algorithm that could be used (x is w bits long).

ALGORITHM: **Modular Exponentiator**:

1. **let** $s_0 = 1$.
2. **for** $k = 0, \ldots, w-1$ **{**
3. **if** bit k of x is 1 **then**
4. **let** $R_k = (s_k \cdot y) \bmod n$.
5. **else**
6. **let** $R_k = s_k$.
7. **let** $s_{k+1} = R_k^2 \bmod n$.

}

8. **return** R_{w-1}.

□

Whenever the kth bit of x is 1 the algorithm would need to calculate a modular multiplication, whereas if the kth bit of x is 0 the algorithm would only need to perform an assignment. Most implementations would exhibit a clear difference in timing between the two branches.

For a practical attack, it might not be feasible to obtain timing information about individual iterations of the modular exponentiation loop when performed as part of the cryptosystem. Instead Kocher measures the total time of the complete modular exponentiation operation, for multiple values of y, as carried out by the cryptosystem. The attack then proceeds by guessing individual bits of the exponent (starting at the most significant bit) and measuring the total time of the operation up to the current guessed position. By comparing, for each guess, the variances between the cryptosystems times and the generated times (for multiple samples) one can verify which guess, 0 or 1, is more likely to be correct. This relies on the observation that if the guess for the exponent bit is correct the variance will be lower than when the guess is incorrect. Once a bit has been chosen the attack proceeds to guess the next less significant bit.

SSH and Network Traffic Timing

Above we described an attack that uses the timing of computational operations. We now describe an attack that uses timing and quantity of network traffic so as to reveal information that is supposed to be secure.

This attack [30] is directed at the SSH network protocol. SSH is designed to provide a secure channel between two hosts attached to the Internet. To achieve this, SSH implements authentication protocols that are used to establish a connection and a session key. SSH then uses the session key together with a block cipher to encrypt all traffic. Primarily, SSH is used to provide an interactive remote shell to a user who can then type in commands that are executed by the remote system.

When commands are typed, the result of each key-stroke is transmitted in a separate packet of data. The output resulting from the execution of the command may then be returned in a single larger packet. If an opponent were to eavesdrop on the network he would not be able to extract any meaning from the contents of the packets because they are all encrypted. However, the pattern of packet sizes and time-lags can be linked to specific commands (e.g. the command `su` would exhibit a particular pattern and since the last part of the pattern is created by typing a password the eavesdropper could infer the length of the password, see Figure 7.2).

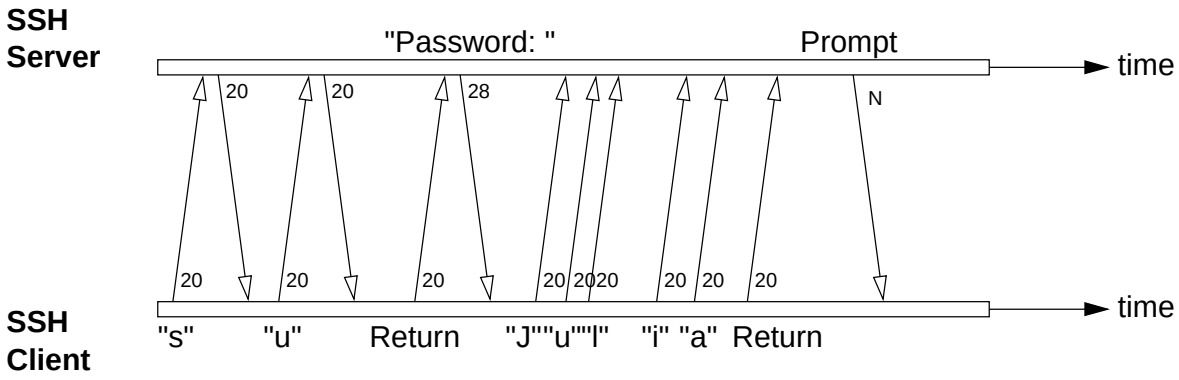


Figure 7.2: Timing and size of SSH packets when an `su` command is initiated

Furthermore, the time-lag between each of the packets generated by a key-stroke very closely matches the time-lag between the actual key-strokes performed by the user. It has been shown that individual typists exhibit a unique pattern when typing (this information can be used as an identification method). Additionally, the timing information provides statistical data about the actual content of messages being typed. This can be used to identify word lengths or possible letter combinations. Since this data provides statistical data about the plaintext, it can be used to augment cryptanalysis of the encryption scheme.

To extract information about the content, one can analyse the timing between pairs of key-strokes using a Hidden Markov Model (HMM). The states of the HMM are possible key-pairs and the output associated with the states is the inter-keystroke timing. In the work on SSH, the packet size patterns were used to find potential password sequences and the inter-keystroke timings were used to rank potential passwords which could then be tested for validity. The ranking procedure produced a 50 times speed-up in the search for a correct password when compared to an exhaustive search.

7.8.2 Differential Power Analysis

Most cryptosystems are implemented on digital electronic devices. These devices consume electrical power when performing calculations. The amount of power that is consumed is dependent on the type of calculations being performed (i.e. the particular machine code instruction that is executed), the values of the variables used during the calculation and the quantity of calculations performed. Hence the power consumption of a semiconductor device shows correlations to the calculations that are being performed and these correlations can be used to infer information about the internal state of the cryptographic algorithms [23].

One type of device where this correlation is particularly evident is the smart-card. This is primarily due to the relative simplicity of the device, the ease in which the power consumption of

the microprocessor can be measured and the fact that the microprocessor is usually dedicated to the cryptographic task (there is no multiprocessing). If one measures the current passing through the processor whilst performing a DES encryption, the 16 rounds can be clearly observed by the recurring pattern of increases and decreases in power consumption that occur as similar sequences of instructions are repeated 16 times.

In addition to the large-scale power variations due to instruction sequences, there are small-scale effects that can be correlated to the actual data being processed. This opens up the possibility of using statistical techniques to enhance the correlations and guess the secret key bits. Differential Power Analysis (DPA) achieves this by constructing a selection function that is correlated to the value of a particular bit in the internal state of the algorithm. Since the power consumption is correlated to the value of the given bit, the selection function can be used in a convolution with the power trace to create a differential trace that exhibits constructive superposition when the guess for the key bits is correct. When the guess is incorrect the differential trace will tend to zero as more ciphertext results are included in the convolution.

In particular the selection function, $D(C, b, K_s)$, is defined by computing the value of bit $0 \leq b \leq 32$ as output by the S-boxes in the 16th round. K_s represents the 6 key bits that enter the S-box that corresponds to the output that contains bit b. C is the final cipher text. If $T_{1\ldots m}[1 \ldots k]$ are the power traces, containing k samples, for each of the m encryption operations and $C_{1\ldots m}$ are the resultant ciphertexts then we define the k-sample differential trace, $\Delta_D[1 \ldots k]$ by

$$\Delta_D[1 \ldots k] = \frac{\sum_{i=1}^{m} D(C_i, b, K_s) T_i[j]}{\sum_{i=1}^{m} D(C_i, b, K_s)} - \frac{\sum_{i=1}^{m} (1 - D(C_i, b, K_s)) T_i[j]}{\sum_{i=1}^{m} (1 - D(C_i, b, K_s))}.$$

$\Delta_D[1 \ldots k]$ is the difference between the average of the power trace for which $D(C, b, K_s)$ is one and the average for which $D(C, b, K_s)$ is zero. If K_s is incorrect then the bit computed by D will differ from the actual target bit for about half of the ciphertexts, C_i. D will then be uncorrelated to the power traces and will effectively divide the power traces into two random sets for which the difference of the averages if found. Hence an incorrect value of K_s implies $\lim_{m\to\infty} \Delta_D[j] \approx 0$. If K_s is correct, the value computed by D will equal the value of the actual target bit and as a result the differential trace will approach the effect that the target bit has on the power consumption as $m \to \infty$.

By testing all possible values of K_s we could then pick out the correct values by monitoring the positive correlation indicated by non-zero differential traces. Once this has been completed for each of the S-boxes, 48 key bits would be recovered. The remaining 8 bits can be easily discovered using an exhaustive search.

7.8.3 Differential Fault Analysis

As we have seen, side-channels can be measured by the opponent even when the implementation is designed to be tamper-proof. It is possible for the opponent to affect the environment and cause the implementation to produce faults, which can then be used to obtain information about the internal states of the cryptosystems [9].

Biham and Shamir showed that by inducing faults in a smart-card they could recover a secret key from an implementation of DES using as few as 200 ciphertexts. Faults can be induced via strong radiation sources, cutting a single wire or using an accurate laser to destroy a single memory cell. Two methods for fault analysis are suggested:

Differential Fault Analysis: This is a chosen plaintext attack. The same plaintext is encrypted multiple times until a difference in the output, due to an induced fault, is observed. Using

the difference, it is possible to find out during which round of DES the fault occurred (this is achieved by looking at the number of differences and noting that the earlier a fault occurs, the more it will propagate throughout the ciphertext). Once the round is found, it is possible to use the differences between the two runs to identify the S-boxes that correspond to the faulty bit and ultimately guess six key bits. By extending this approach, the key can be recovered.

Non-Differential Fault Analysis: This method relies on creating a permanent fault in one bit of the register that stores the intermediate results of the Feistel network whilst the round functions are being processed. As a result there is a fixed known affect on the ciphertext output. This known affect can be used to guess subkey bits for the last round. Once the subkey has been recovered, it is easy to recover the rest of the secret key. The advantage of this method is that it is a ciphertext only attack.

7.9 Zero-Knowledge Proofs

Zero-Knowledge Proofs provide mechanisms creating protocols where one entity, Peggy The Prover, wishes to prove to another entity, Victor The Verifier, that she possesses some knowledge regarding some particular secret. However, Peggy does not wish to reveal the actual information and furthermore, she does not wish to enable Victor to prove to any other entity that he has knowledge of the secret information that he does not actually possess.

This is different to an authentication scheme where the verifier has a copy of the secret information that the prover wishes to demonstrate a knowledge of.

To prove that Victor gains no knowledge from Peggy, we use the concept of a *simulator*. Firstly as will be seen below, when Peggy proves to Victor that she possesses some item of information, the process generates a *transcript* which consists of the initial publicly shared data and all the messages that are transfered between Victor and Peggy. A simulator is an algorithm that uses the same publicly shared initial information and generates valid but forged transcripts. The fact that Victor can generate forged transcripts that are indistinguishable (i.e. they exhibit the same probability distributions) from a real transcript is used as the basis for proving that the verification protocol does not enable Victor to gain any information about the secret that Peggy holds.

7.9.1 Zero-Knowledge Proof Using Graph Isomorphisms

The zero knowledge protocol we are going describe here, uses graphs and graph isomorphisms as its fundamental building blocks.

Definition 7.5 (Graph) *A graph is a collection of vertices together with a collection of edges joining a subset of pairs of vertices. Let* V *be a set of labels defining the vertices. Let* $E \subseteq V \times V$ *define a set of edges. Together* V *and* E *define a graph* G.

Definition 7.6 (Graph Isomorphism) *Let* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *be two graphs.* G_1 *and* G_2 *are said to be isomorphic if there exists a bijection* $\pi : V_1 \to V_2$ *such that* $\{u, v\} \in E_1$ *if and only if* $\{\pi(u), \pi(v)\} \in E_2$.

Given two graphs G_1 and G_2, the problem of checking whether or not they are isomorphic and then constructing an isomorphism is an NP-complete problem. This makes it possible to make the graphs publicly known but keep the bijection defining the isomorphism secret. If the graphs are

sufficiently large the computation resources required will be to great for anybody else to construct the isomorphism.

With this setting, we can now describe the zero-knowledge protocol. Firstly, Peggy constructs two graphs, G_1 and G_2, that are isomorphic. This can be done by constructing a graph and then randomly choosing a permutation which defines the isomorphism. Peggy makes G_1 and G_2 publicly known. At some time Victor requires that Peggy prove that she knows the isomorphism between G_1 and G_2 (this could be to prove her identity). To do this, Victor and Peggy follow the following steps:

1. Peggy constructs a second random permutation of G_1 to produce a third graph H. Since she knows the isomorphism between G_1 and G_2, she can also construct the isomorphism between H and G_2. The knowledge of the isomorphism between H and G_1 or between H and G_2 will not help anyone else to construct the secret isomorphism between G_1 and G_2.

2. Peggy sends H to Victor.

3. Victor flips a coin to randomly decide to ask Peggy to:

   - demonstrate that H is isomorphic to G_1, or
   - demonstrate that H is isomorphic to G_2.

4. Peggy answers Victor by either:

   - proving that H is isomorphic to G_1, or
   - proving that H is isomorphic to G_2.

5. Peggy and Victor repeat steps 1. to 4. n times.

If Peggy does not know the isomorphism between G_1 and G_2 then she can not construct a graph H that is isomorphic to both G_1 and G_2. The best she can achieve is to construct H that is isomorphic to exactly one or the other. If at any stage Peggy can not provide the appropriate isomorphism, Victor will know that Peggy does not actually know the secret isomorphism. When, in step 3., Victor asks her to prove that she knows the isomorphism between H and either G_1 or G_2 she has a 50% chance of being able to complete this task. After 16 rounds she has a 1 in 65536 chance of being able to successfully fool Victor.

Once the protocol is complete, Victor knows whether or not Peggy has the secret isomorphism. However, during each round Victor receives a random permutation, H, and an isomorphism between it and either G_1 or G_2. He could have generated this himself. Thus he has not gained any knowledge of the secret and can not use any transcripts of the protocol to prove to a third party that he knows the secret.

Chapter 8

Conclusions

8.1 Mathematics and Cryptology

In this survey I have covered the basics of cryptology. By starting with simple cryptosystems, such as affine ciphers and Vigenere ciphers, the main principles were introduced. I then introduced cryptanalysis and demonstrated techniques that could be used to attack simple cryptosystems.

With this grounding it was possible to discuss block ciphers. These are the workhorses of most cryptographic implementations. Through the short descriptions of various block cipher algorithms we see that the field has matured greatly over the second half of the 20th century.

After discussing the general concepts of block ciphers, I provided a detailed description of the DES algorithm. This provided a good setting for explaining how differential cryptanalysis can be used to break block ciphers like DES. The techniques learnt from differential cryptanalysis can also be used to break pseudo-random number generators and the concepts provide greater insight into other cryptosystems. I then presented a brief overview of linear cryptanalysis. This technique is in some senses the dual of differential cryptanalysis. Linear cryptanalysis has also produced some of the most successful attacks on DES-like cryptosystems.

I then discussed Knapsack cryptosystems. These systems are based on the subset-sum problem and provide an interesting setting for using results from fields of lattice theory and integer programming.

Pseudo-random number generators are very important in many cryptosystems and are used for tasks such as creating encryption keys or initialising block cipher modes. I introduce PRNGs, amongst which the Blum-Blum-Shub (BBS) generator is one of the more important algorithms. The BBS generator is designed on concrete mathematical ideas taken from the field of number theory. This facilitates a thorough understanding and investigation, as a result of which it is possible to provide a mathematical analysis that motivates the acceptance of the BBS generator as a secure PRNG.

For the purpose of understanding how cryptosystems work it is important to realise that any one implementation usually consists of a number of distinct components. Furthermore, a secure cryptosystem must be able to withstand a wide range of different attacks. Each of these components or attacks is based on separate concepts and techniques. In the interest of providing a more rounded survey I chose to include brief overviews of many different ideas and algorithms. These included hash functions, digital signatures, key distribution and key agreement, subliminal channels, secret sharing, differential power analysis and zero-knowledge proofs.

Cryptology is a broad and rapidly advancing field and the purpose of this survey was to provide an overview that could be used to understand how the field fits together, the role of mathematics

in the field, and to highlight some of the areas that have been seen as important to the field's development. With this overview in mind, it should be possible to identify particular areas of interest or areas that need more attention and then embark on a more in-depth study.

8.2 Applications and Implications

Cryptanalysis is not just a theoretical subject. Rather, it is a field that has many real-world applications with real implications.

During many past wars the outcomes have been radically altered because of information that has been obtained via the cryptanalysis of intercepted messages.

Improved cryptanalytic attacks and faster cryptanalytic implementations have rendered algorithms such as DES insecure. This has direct implications for banks since many use the DES algorithm to secure PIN codes and other sensitive information. Thus banks that continue to use DES are potentially vulnerable to fraud.

Cryptography is no longer only important to military organisations, diplomatic communications or even large organisations, but instead, it is becoming more important to individuals in an everyday civilian context. Cryptography is used to protect pin numbers and passwords, to enable details entered into World Wide Web based forms to be passed securely and to prevent unwanted eavesdropping on cellular phone calls.

However, poorly designed cryptosystems may mean that the applications do not protect individuals as well as was intended. Many cellular phones use an algorithm known as A5. This has subsequently been cracked, with the implication that the cellular phones can be "cloned" and thus an adversary could use the cellular phone network while a victim is fraudulently billed for the time. It is also possible for an adversary to decrypt the intercepted phone call data and recover the full conversation [16, 10].

As another example of current cryptanalytic efforts, the Wired Equivalent Protocol (WEP), which is a link layer security protocol for wireless network cards using the 802.11 standard, was recently demonstrated to be insecure after the protocol was cryptanalysed and shown to contain flaws [32]. The WEP protocol is supposed to provide wireless 802.11 based networks with security equivalent to a network implemented using cables that can not be accessed for wire-tapping. This has an impact on many networks using 802.11 wireless connectivity because it is possible to eaves-drop on all communications. To compound the problem, the solution requires the hardware implementation to be updated.

The continued cryptanalytic efforts demonstrate how difficult it is to develop secure systems, and furthermore how difficult it is to correctly implement the cryptographic systems. The results of poor designs or implementations could have adverse side-effects for both individuals and organisations.

8.3 Conclusion

As in the past, cryptology is not a static field. It is instead a game of spy versus spy, where new cryptosystems are designed to thwart new attacks and new attacks are discovered to break new cryptosystems.

Cryptography will continue to develop to meet the needs of people communicating via new methods and with new levels of risk. Cryptography will also evolve to counter new theoretical cryptanalytical attacks (new methods) as well as the attacks that are made practical by advances in related fields of mathematics (e.g. faster factorisation algorithms) and computing technology

(e.g. speed improvements, massively parallel computers, massively distributed computers, or fundamentally new types of computing such as quantum computers).

Bibliography

[1] Data Encryption Standard (DES). Federal Information Processing Standards Publication (FIPS PUB 46-3), 1999. U.S. Department of Commerce/National Institute of Standards and Technology.

[2] A.Menezes, P.van Oorschot, and S.Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[3] A.M.Odlyzko. The rise and fall of knapsack cryptosystems. In *Cryptology and Computational Number Theory*, volume 42, pages 75–88. AMS, 1989.

[4] Tom M. Apostol. *Introduction to Analytic Number Theory*. Springer, 1976. Undergraduate Texts in Mathematics.

[5] Ishai Ben-Aroya and Eli Biham. Differential cryptanalysis of Lucifer. In *Advances in Cryptology*, volume CRYPTO '93, pages 187–199. Springer, 1993.

[6] Eli Biham. On Matsui's linear cryptanalysis. In *Advances in Cryptology*, volume EUROCRYPT '94, pages 341–355. Springer, 1994.

[7] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems (extended abstract). In *Advances in Cryptology*, volume CRYPTO '90, pages 2–21. Springer, 1990.

[8] Eli Biham and Adi Shamir. Differential cryptanalysis of full 16-round DES. In *Advances in Cryptology*, volume CRYPTO '92, pages 487–496. Springer, 1992.

[9] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology*, volume CRYPTO '97, pages 513–525. Springer, 1997.

[10] Alex Biryukov and Adi Shamir. Real time cryptanalysis of the alleged A5/1 on a PC. World Wide Web, 1999. http://cryptome.org/a51-bs.htm.

[11] Ernest F. Brickell. Solving low density knapsacks. In *Advances in Cryptology*, volume CRYPTO '83, pages 25–37. Springer, 1997.

[12] Keith W. Campbell and Michael J. Wiener. DES is not a group. In *Advances in Cryptology*, volume CRYPTO '92, pages 512–520. Springer, 1992.

[13] C.P.Schnorr and H.H.Hörner. Attacking the Chor Rivest cryptosystem by improved lattice reduction. In *Advances in Cryptology*, volume EUROCRYPT '95, pages 1–12. Springer, 1995.

[14] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, volume CRYPTO '84, pages 10–18. Springer, 1984.

[15] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In *Advances in Cryptology*, volume CRYPTO '97, pages 112–131. Springer, 1997.

[16] Jovan Dj. Golić. Cryptanalysis of alleged A5 stream cipher. In *Advances in Cryptology*, volume EUROCRYPT '97, pages 239–255. Springer, 1997.

[17] Martin E. Hellman. A cryptanalytic time-memory trade-off. In *IEEE Transactions on Information Theory*, volume IT-26, pages 401–406, 1980.

[18] J.C.Lagarias. Pseudorandom number generators in cryptography and number theory. In *Cryptology and Computational Number Theory*, volume 42, pages 115–143. AMS, 1989.

[19] H. W. Lenstra junior. Integer programming with a fixed number of variables. In *Mathematics of Operations Research*, volume 8, No. 4., November 1983, pages 538–548, 1983.

[20] David Kahn. *The Code Breakers*. Scribner, revised edition, 1996.

[21] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. World Wide Web: Counterpane Systems and U.C. at Berkeley, *circa* 2000. http://www.counterpane.com, http://www.cs.berkeley.edu.

[22] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998. Sorting and Searching.

[23] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. World Wide Web: Cryptography Research, Inc., *circa* 2000. http://www.cryptography.com.

[24] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology*, volume CRYPTO '96, pages 104–113. Springer, 1996.

[25] K.S.McCurley. Odds and ends from cryptology and computational number theory. In *Cryptology and Computational Number Theory*, volume 42, pages 145–166. AMS, 1989.

[26] Xuejia Lai and James Massey. Markov ciphers and differential cryptanalysis. In *Advances in Cryptology*, volume EUROCRYPT '91, pages 17–38. Springer, 1997.

[27] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology*, volume EUROCRYPT '93, pages 386–397. Springer, 1993.

[28] R.L.Graham, D.E.Knuth, and O.Patashnik. *Concrete Mathematics, A Foundation for Computer Science*. Addison-Wesley, 2nd edition, 1998.

[29] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., 2nd edition, 1996.

[30] Dawn X. Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on SSH. World Wide Web, 2001. University of California, Berkeley.

[31] Douglas R. Stinson. *Cryptography Theory and Practice*. CRC Press, 1995.

[32] Adam Stubblefield, John Ioannidis, and Aviel Rubin. Using the Fluhrer, Mantin, and Shamir attack to break WEP. World Wide Web, 2001. AT&T Labs Technical Report TD-4ZCPZZ.

[33] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

Appendix A

DES Implementation

For the purposes of understanding the DES algorithm I wrote a C++ implementation. The emphasis of this implementation was on understandability. Thus the implementation is implemented in a modular fashion with many attributes of the algorithms behaviour being easily modifiable. The attributes that can be easily changed are:

- toggle the use of the initial permutation
- toggle the swap in last round of Feistel network
- change the number of rounds (the default is 16)
- toggle the display of verbose output

In §A.1 I show the output resulting from full 16-round DES. This shows the progress of the transformations during each round within the Feistel network.

In §A.2 I list the C++ source code for the DES implementation. I have not included some of the simple support code, nor have I included the unit test code that was used during development.

In Appendix B I use these DES routines when implementing the differential cryptanalytic attack on 3-round DES.

A.1 Results

Below we see the process of encrypting 159CA017A12CA37D using 1A624C89520DEC46 as the key. The resultant cipher text is DCA153BE2712A011 (each string is the hex representation of a 64-bit block of data).

OUTPUT: 16 **Rounds of DES Encryption**

```
plaintext=00010101 10011100 10100000 00010111 10100001 00101100 10100011 01111101
key=00011010 01100010 01001100 10001001 01010010 00001101 11101100 01000110
LR_start=808babd956f4a248
      L_0 = 10000000 10001011 10101011 11011001
L_1 = R_0 = 01010110 11110100 10100010 01001000
E(R_0) = 00101010 11010111 10101001 01010000 01000010 01010000
K_1 = 11101000 00000001 01001010 10110100 11010101 10100100
E(R_0) xor K_1     = 11000010 11010110 11100011 11100100 10010111 11110100
```

```
S-box outputs     = 11110100 10111111 10100111 01101010
f(R_0,K_1) = 10001101 11110010 10110111 11011110
L_2 = R_1 = 00001101 01111001 00011100 00000111
        ----        ----        ----
E(R_1) = 10000101 10101011 11110010 10001111 10000000 00001110
K_2 = 00000011 00110010 11110010 00011111 01111000 10001100
E(R_1) xor K_2     = 10000110 10011001 00000000 10010000 11111000 10000010
S-box outputs     = 11110011 01000111 00010101 01000010
f(R_1,K_2) = 11100000 11010011 11110010 00001010
L_3 = R_2 = 10110110 00100111 01010000 01000010
        ----        ----        ----
E(R_2) = 01011010 11000001 00001110 10101010 00000010 00000101
K_3 = 10111100 01010100 11000000 01100000 01110001 11110001
E(R_2) xor K_3     = 11100110 10010101 11001110 11001010 01110011 11110100
S-box outputs     = 10100011 11101010 10011100 10101010
f(R_2,K_3) = 01111001 11000011 01000111 01001101
L_4 = R_3 = 01110100 10111010 01011011 01001010
        ----        ----        ----
E(R_3) = 00111010 10010101 11110100 00101111 01101010 01010100
K_4 = 01010010 01000011 01001000 10100011 10101000 00101111
E(R_3) xor K_4     = 01101000 11010110 10111100 10001100 11000010 01111011
S-box outputs     = 10011000 01001000 10000110 01000101
f(R_3,K_4) = 00000001 10111001 00001000 01101010
L_5 = R_4 = 10110111 10011110 01011000 00101000
        ----        ----        ----
E(R_4) = 01011010 11111100 11111100 00101111 00000001 01010001
K_5 = 00001000 11010001 00010101 11100110 00011111 10010010
E(R_4) xor K_5     = 01010010 00101101 11101001 11001001 00011110 11000011
S-box outputs     = 01101110 00111010 10010110 00101111
f(R_4,K_5) = 01101101 01101010 10001110 01111100
L_6 = R_5 = 00011001 11010000 11010101 00110110
        ----        ----        ----
E(R_5) = 00001111 00111110 10100001 01101010 10101001 10101100
K_6 = 00000101 00001001 01001111 00011101 00000011 01111111
E(R_5) xor K_6     = 00001010 00110111 11101110 01110111 10101010 11010011
S-box outputs     = 01001000 00011101 10001101 01000101
f(R_5,K_6) = 10010101 00011000 10111000 01101000
L_7 = R_6 = 00100010 10000110 11100000 01000000
        ----        ----        ----
E(R_6) = 00010000 01010100 00001101 01110000 00000010 00000000
K_7 = 00100011 01100000 10100001 01010111 11011000 11000000
E(R_6) xor K_7     = 00110011 00110100 10101100 00100111 11011010 11000000
S-box outputs     = 10110110 11010111 01001000 01001101
f(R_6,K_7) = 11011100 11010101 00011011 00110010
L_8 = R_7 = 11000101 00000101 11001110 00000100
        ----        ----        ----
E(R_7) = 01100000 10101000 00001011 11100101 11000000 00001001
```

```
K_8 = 10011001 00001101 10100000 01000000 10100101 01111101
E(R_7) xor K_8     = 11111001 10100101 10101011 10100101 01100101 01110100
S-box outputs     = 00000000 01110001 00010100 01011010
f(R_7,K_8) = 10101110 00010011 00000000 00001100
L_9 = R_8 = 10001100 10010101 11100000 01001100
        ----        ----        ----
E(R_8) = 01000101 10010100 10101011 11110000 00000010 01011001
K_9 = 00000011 10001000 01001010 11110100 11010100 10011001
E(R_8) xor K_9     = 01000110 00011100 11100001 00000100 11010110 11000000
S-box outputs     = 10101101 11110011 11101001 11111101
f(R_8,K_9) = 10011111 11011101 01101111 10110101
L_10 = R_9 = 01011010 11011000 10100001 10110001
        ----        ----        ----
E(R_9) = 10101111 01010110 11110001 01010000 00111101 10100010
K_10 = 00101000 01101000 10100110 00001011 00110110 01101011
E(R_9) xor K_10     = 10000111 00111110 01010111 01011011 00001011 11001001
S-box outputs     = 11110110 10111100 11110111 01111010
f(R_9,K_10) = 01101111 10110110 10110111 11011110
L_11 = R_10 = 11100011 00100011 01010111 10010010
        ----        ----        ----
E(R_10) = 01110000 01101001 00000110 10101010 11111100 10100101
K_11 = 10110000 00101101 00001000 10111110 11111001 00100000
E(R_10) xor K_11     = 11000000 01000100 00001110 00010100 00000101 10000101
S-box outputs     = 11111000 00011010 00101100 01111101
f(R_10,K_11) = 00011110 11011000 10001110 11101010
L_12 = R_11 = 01000100 00000000 00101111 01011011
        ----        ----        ----
E(R_11) = 10100000 10000000 00000000 00010101 11101010 11110110
K_12 = 01000000 00100110 00110001 00100000 01001111 01110110
E(R_11) xor K_12     = 11100000 10100110 00110001 00110101 10100101 10000000
S-box outputs     = 00111011 10111001 11010111 01111101
f(R_11,K_12) = 11101111 00111100 01101111 01101110
L_13 = R_12 = 00001100 00011111 00111000 11111100
        ----        ----        ----
E(R_12) = 00000101 10000000 11111110 10011111 00010111 11111000
K_13 = 11000101 10011100 00010100 11011101 10101000 10010010
E(R_12) xor K_13     = 11000000 00011100 11101010 01000010 10111111 01101010
S-box outputs     = 11110011 11111011 10000101 00111100
f(R_12,K_13) = 11001111 11000001 11100111 01101110
L_14 = R_13 = 10001011 11000001 11001000 00110101
        ----        ----        ----
E(R_13) = 11000101 01111110 00000011 11100101 00000001 10101011
K_14 = 01000110 10100010 11000010 11100101 01000110 01011001
E(R_13) xor K_14     = 10000011 11011100 11000001 00000000 01000111 11110010
S-box outputs     = 01001110 11111101 00101010 01100110
f(R_13,K_14) = 11010100 00111011 10010101 11110100
L_15 = R_14 = 11011000 00100100 10101101 00001000
```

```
        ----        ----        ----
E(R_14) = 01101111 00000001 00001001 01010101 10101000 01010001
K_15 = 00111010 11010100 00100010 00011011 10110010 01001110
E(R_14) xor K_15     = 01010101 11010101 00101011 01001110 00011010 00011111
S-box outputs     = 11001011 11000001 00000100 11000010
f(R_14,K_15) = 11000000 10011011 11000001 00001001
L_16 = R_15 = 01001011 01011010 00001001 00111100
        ----        ----        ----
E(R_15) = 00100101 01101010 11110100 00000101 00101001 11111000
K_16 = 10000101 10011010 00000001 01011000 00000111 00111111
E(R_15) xor K_16     = 10100000 11110000 11110101 01011101 00101110 11000111
S-box outputs     = 11011110 01110101 10101101 00101000
f(R_15,K_16) = 11011101 10001001 10110100 10011110
L_17 = R_16 = 00000101 10101101 00011001 10010110
        ----        ----        ----
LR_end=4b5a093c05ad1996
ciphertext=11011100 10100001 01010011 10111110 00100111 00010010 10100000 00010001
k:1a624c89520dec46 p:159ca017a12ca37d c:dca153be2712a011
```

A.2 Source Code

```
des/
      Makefile
      des.cc
      des.hh
      getmask.cc
      keys.cc
      keys.hh
      perm.cc
      perm.hh
      rundes.cc
      runperm.cc
      sbox.cc
      sbox.hh
```

SOURCE: **des/Makefile**

```
# Makefile for des implementation

TOP=../
SRC+=des.cc             \
     sbox.cc            \
     perm.cc            \
     keys.cc
include $(TOP)/config.mk
```

```
all: libsg_des.so

libs: libsg_des.so

#PG=-pg
PG=

timeDES: timeDES.cc $(SRC)
        $(CXX) $(CXXFLAGS) $(PG) -L../common -I../common -o timeDES
timeDES.cc $(SRC) -lsg_common

key_text: key_text.cc $(SRC)
        $(CXX) $(CXXFLAGS) -L./ -I./ -L../common -I../common -o key_text
key_text.cc -lsg_common -lsg_des

test_perm: test_perm.cc $(SRC)
        $(CXX) $(CXXFLAGS) -L./ -I./ -L../common -I../common -o test_perm
test_perm.cc -lsg_common -lsg_des

rundes: rundes.cc libsg_des.so
        $(CXX) $(CXXFLAGS) -L./ -I./ -L../common -I../common -o rundes
rundes.cc -lsg_common -lsg_des

runperm: runperm.cc libsg_des.so
        $(CXX) $(CXXFLAGS) -L./ -I./ -L../common -I../common -o runperm
runperm.cc -lsg_common -lsg_des

getmask: getmask.cc libsg_des.so
        $(CXX) $(CXXFLAGS) -L./ -I./ -L../common -I../common -o getmask
getmask.cc -lsg_common -lsg_des

libsg_des.so: $(OBJS)
        $(CXX) $(CXXFLAGS) $(LIBDIR) -shared -o libsg_des.so $(OBJS)
$(LIBS)
#       strip ppp_applet

clean:
        rm -f *.o
        rm -f *~
        rm -f .depend
        rm -f libsg_des.so
        rm -f timeDES
        rm -f rundes
        rm -f runperm
```

SOURCE: **des/des.cc**

```
static char cvsid[] = { "@(#) $Id: des.cc,v 1.2 2001/11/25 10:46:00
stewart Exp $" };
#include "common.h"
USE(cvsid);

#include "des.hh"

#undef USEDISP
#define USEDISP
#undef USEIP
#define USEIP
#undef USERLSWAP
#define USERLSWAP

DES::DES(void)
{
    display=false;
    useip=true;
    userlswap=true;
    numrounds=16;

#ifndef USEPERMCLASS
    IDENT.setPerm(Permutation::ID::IDENT);
    IP.setPerm(Permutation::ID::IP);
    IPINV.setPerm(Permutation::ID::IPINV);
    P.setPerm(Permutation::ID::P);
    E.setPerm(Permutation::ID::E);
#endif

    S1.setTable(SBox::S1);
    S2.setTable(SBox::S2);
    S3.setTable(SBox::S3);
    S4.setTable(SBox::S4);
    S5.setTable(SBox::S5);
    S6.setTable(SBox::S6);
    S7.setTable(SBox::S7);
    S8.setTable(SBox::S8);
}
DES::~DES(void)
{}

void DES::setDisplay(bool b)
```

```
{ display=b; }

void DES::setUseIP(bool b)
{ useip=b; }

void DES::setUseRLSwap(bool b)
{ userlswap=b; }

void DES::setRounds(int r)
{ numrounds=r; }

void DES::setKey(uchar_ptr k)
{
    K.buildK(k);
}

//#define UEQ u=
#define UEQ

#define XOR6(a,b,out) \
out[0]=a[0]^b[0]; out[1]=a[1]^b[1]; out[2]=a[2]^b[2]; out[3]=a[3]^b[3];
out[4]=a[4]^b[4]; out[5]=a[5]^b[5]

#define XOR4(a,b,out) \
out[0]=a[0]^b[0]; out[1]=a[1]^b[1]; out[2]=a[2]^b[2]; out[3]=a[3]^b[3]

inline void DESxor4(const unsigned char* a, const unsigned char* b,
                    unsigned char* out)
{
    (*((long*)out))=(*((long*)a))^(*((long*)b));
}

inline void DESxor6(const unsigned char* a, const unsigned char* b,
                    unsigned char* out)
{
    //first 4 bytes
    (*((long*)out))=(*((long*)a))^(*((long*)b));
    //last 2 bytes
    (*((short*)(out+4)))=(*((short*)(a+4)))^(*((short*)(b+4)));
}

void DESxor(const unsigned char* a, const unsigned char* b,
                    unsigned char* out, unsigned int n)
{
    switch(n)
    {
        case 4: //DESxor4(a,b,out);
```

```
                (*((long*)out))=(*((long*)a))^(*((long*)b));
                break;
        case 6: //DESxor6(a,b,out);
                (*((long*)out))=(*((long*)a))^(*((long*)b));
                (*((short*)(out+4)))=(*((short*)(a+4)))^(*((short*)(b+4)));
                break;
        default:
                register int i;
                for(i=0;i<n;i++)
                    out[i]=a[i]^b[i];
    }
}


inline void DES::DESRoundFunction(uchar_ptr in, uchar_ptr out, int round)
{
    //register uchar s;
    register uchar s1;
    register uchar s2;
    register uchar Sout[4];
    uchar Ein[6];
    uchar cXk[6];
//    uchar u;
    //expand input
    E(in,Ein);
#ifdef USEDISP
if(display) cout<<"E(R_"<<round-1<<")\t\t\t= "<<tobin(Ein,6)<<endl<<flush;
if(display) cout<<"K_"<<round<<"\t\t\t= "<<tobin(K[round],6)<<endl<<flush;
#endif
    //xor with key (from key schedule for round)
    DESxor(Ein,K[round],cXk,6);
#ifdef USEDISP
if(display) cout<<"E(R_"<<round-1<<") xor K_"<<round<<"    \t=
"<<tobin(cXk,6)<<endl<<flush;
#endif
    //apply SBoxes
    //   1        2      3      4         5        6      7      8
    // 0       0  1       1  2       2  3       3  4       4  5       5
    //[123456|78][1234|5678][12|345678][123456|78][1234|5678][12|345678]
    // 123456|12  3456|1234  56|123456| 123456|12  3456|1234  56|123456
    // Masks 11111100 0x3f 1
    //       00000011 0xc0 2
    //       11110000 0x0f 3
    //       00001111 0xf0 4
    //       11000000 0x03 5
    //       00111111 0xfc 6
#define M1 0x3f
```

```
#define M2 0xc0
#define M3 0x0f
#define M4 0xf0
#define M5 0x03
#define M6 0xfc
#if 0
    s=S1(UEQ cXk[0]&M1 );
    Sout[0]=s&0x0f;
//cout<<"<"<<tobit(u,6)<<"|"<<tobit(s,4)<<">"<<flush;
    s=S2(UEQ ((cXk[0]&M2)>>6) | ((cXk[1]&M3)<<2) );
    Sout[0]|=((s<<4)&0xf0);
//cout<<"<"<<tobit(u,6)<<"|"<<tobit(s,4)<<">"<<flush;

    s=S3(UEQ ((cXk[1]&M4)>>4) | ((cXk[2]&M5)<<4) );
    Sout[1]=s&0x0f;
//cout<<"<"<<tobit(u,6)<<"|"<<tobit(s,4)<<">"<<flush;
    s=S4(UEQ (cXk[2]&M6)>>2 );
    Sout[1]|=((s<<4)&0xf0);
//cout<<"<"<<tobit(u,6)<<"|"<<tobit(s,4)<<">"<<flush;
    //-------
    s=S5(UEQ cXk[3]&M1);
    Sout[2]=s&0x0f;
//cout<<"<"<<tobit(u,6)<<"|"<<tobit(s,4)<<">"<<flush;
    s=S6(UEQ ((cXk[3]&M2)>>6) | ((cXk[4]&M3)<<2) );
    Sout[2]|=((s<<4)&0xf0);
//cout<<"<"<<tobit(u,6)<<"|"<<tobit(s,4)<<">"<<flush;

    s=S7(UEQ ((cXk[4]&M4)>>4) | ((cXk[5]&M5)<<4) );
    Sout[3]=s&0x0f;
//cout<<"<"<<tobit(u,6)<<"|"<<tobit(s,4)<<">"<<flush;
    s=S8(UEQ (cXk[5]&M6)>>2 );
    Sout[3]|=((s<<4)&0xf0);
//cout<<"<"<<tobit(u,6)<<"|"<<tobit(s,4)<<">"<<flush;
//cout<<endl;
#else
    s1=S1(cXk[0]&M1);
    s2=S2(((cXk[0]&M2)>>6) | ((cXk[1]&M3)<<2) );
    Sout[0]=s1&0x0f | ((s2<<4)&0xf0);

    s1=S3(((cXk[1]&M4)>>4) | ((cXk[2]&M5)<<4) );
    s2=S4((cXk[2]&M6)>>2 );
    Sout[1]=s1&0x0f | ((s2<<4)&0xf0);
    //-------
    s1=S5(cXk[3]&M1);
    s2=S6(((cXk[3]&M2)>>6) | ((cXk[4]&M3)<<2) );
    Sout[2]=s1&0x0f | ((s2<<4)&0xf0);
```

```
    s1=S7(((cXk[4]&M4)>>4) | ((cXk[5]&M5)<<4) );
    s2=S8((cXk[5]&M6)>>2 );
    Sout[3]=s1&0x0f | ((s2<<4)&0xf0);
#endif
#ifdef USEDISP
if(display) cout<<"S-box outputs    \t= "<<tobin(Sout,4)<<endl<<flush;
#endif
    //apply post permutation
    P(Sout,out);
}

//Encryption
//both the input and output buffers are 8 bytes long
uchar_ptr DES::operator()(const uchar_ptr in, uchar_ptr out, bool d)
{
    if(d)
        return decrypt(in,out);
    else
        return encrypt(in,out);
}
uchar_ptr DES::encrypt(const uchar_ptr in, uchar_ptr out, int rounds)
{
    uchar LR[8];
    uchar RL[8];
    uchar fout[4];
    uchar Rsave[4];
    uchar_ptr L=&LR[0];
    uchar_ptr R=&LR[4];
    uchar_ptr t;

#ifdef USEDISP
if(display) cout<<endl;
if(display) cout<<"plaintext="<<tobin(in,8)<<endl<<flush;
if(display) cout<<"key="<<tobin(K.getKey(),8)<<endl<<flush;
#endif

    //apply initial permutation
#ifdef USEIP
    if(useip)
        IP(in,LR);
    else {
        IDENT(in,LR);
        if(display) cout<<"IP not used"<<endl<<flush; }
#else
    IP(in,LR);
#endif
#ifdef USEDISP
```

```
if(display) cout<<"LR_start="<<tohex(LR,8)<<endl<<flush;
#endif

#ifdef USEDISP
if(display) cout<<"      L_0 = "<<tobin(L,4)<<endl<<flush;
if(display) cout<<"L_1 = R_0 = "<<tobin(R,4)<<endl<<flush;
#endif

    //apply DES rounds
    for(int r=1;r<=rounds;r++)
    {
        //apply sboxes
        DESRoundFunction(R,fout,r);
#ifdef USEDISP
if(display) cout<<"f(R_"<<r-1<<",K_"<<r<<")\t\t=
"<<tobin(fout,4)<<endl<<flush;
#endif
        //save R
        //memcpy(Rsave,R,4);
        *((long*)Rsave)=*((long*)R);
        //xor function output in Feistel network style
        DESxor(L,fout,R,4);
        //swap R and L
        //memcpy(L,Rsave,4);
        *((long*)L)=*((long*)Rsave);
#ifdef USEDISP
if(display) cout<<"L_"<<r+1<<" = R_"<<r<<"\t\t= "<<tobin(R,4)<<endl<<flush;
if(display) cout<<"        ----        ----        ----"<<endl<<flush;
#endif
    }

#ifdef USEDISP
if(display) cout<<"LR_end="<<tohex(LR,8)<<endl<<flush;
#endif
    //populate RL with previous R and L
    //memcpy(&RL[0],R,4);
    //memcpy(&RL[4],L,4);
#ifdef USERLSWAP
    if(userlswap || useip)
    {
        *((long*)&RL[0])=*((long*)R);
        *((long*)&RL[4])=*((long*)L);
    }
    else
    {
        *((long*)&RL[0])=*((long*)L);
        *((long*)&RL[4])=*((long*)R);
```

```
        if(display) cout<<"Last round not swapped"<<endl<<flush;
    }
#else
    *((long*)&RL[0])=*((long*)R);
    *((long*)&RL[4])=*((long*)L);
#endif
    //apply inverse permutation
#ifdef USEIP
    if(useip)
        IPINV(RL,out);
    else {
        IDENT(RL,out);
        if(display) cout<<"IP not used"<<endl<<flush; }
#else
    IPINV(RL,out);
#endif

#ifdef USEDISP
if(display) cout<<"ciphertext="<<tobin(out,8)<<endl<<flush;
#endif

    return out;
}

uchar_ptr DES::encrypt(const uchar_ptr in, uchar_ptr out)
{ return encrypt(in,out,numrounds); }
uchar_ptr DES::encrypt3(const uchar_ptr in, uchar_ptr out)
{ return encrypt(in,out,3); }
uchar_ptr DES::encrypt6(const uchar_ptr in, uchar_ptr out)
{ return encrypt(in,out,6); }

//Decryption
uchar_ptr DES::decrypt(const uchar_ptr in, uchar_ptr out,int rounds)
{
    uchar LR[8];
    uchar RL[8];
    uchar fout[4];
    uchar Rsave[4];
    uchar_ptr L=&LR[0];
    uchar_ptr R=&LR[4];
    uchar_ptr t;

#ifdef USEDISP
if(display) cout<<endl;
if(display) cout<<"ciphertext="<<tobin(in,8)<<endl<<flush;
if(display) cout<<"key="<<tobin(K.getKey(),8)<<endl<<flush;
#endif
```

```
    //apply initial permutation
#ifdef USEIP
    if(useip)
        IP(in,LR);
    else {
        IDENT(in,LR);
        if(display) cout<<"IP not used"<<endl<<flush; }
#else
    IP(in,LR);
#endif

#ifdef USELRSWAP
    if(!uselrswap && !useip)
    {
        long tmp=*((long*)L);
        *((long*)L)=*((long*)R);
        *((long*)R)=tmp;
        if(display) cout<<"Input LR swapped"<<endl<<flush; }
    }
#endif

#ifdef USEDISP
if(display) cout<<"      L_0 = "<<tobin(L,4)<<endl<<flush;
if(display) cout<<"L_1 = R_0 = "<<tobin(R,4)<<endl<<flush;
#endif

    //apply DES rounds in reverse
    for(int r=rounds;r>=1;r--)
    {
        //apply sboxes
        DESRoundFunction(R,fout,r);
#ifdef USEDISP
if(display) cout<<"f(R_"<<r-1<<",K_"<<r<<")\t\t=
"<<tobin(fout,4)<<endl<<flush;
#endif
        //save R
//      memcpy(Rsave,R,4);
        *((long*)Rsave)=*((long*)R);
        //xor function output in Feistel network style
        DESxor(L,fout,R,4);
        //swap R and L
//      memcpy(L,Rsave,4);
        *((long*)L)=*((long*)Rsave);
#ifdef USEDISP
if(display) cout<<"L_"<<r+1<<" = R_"<<r<<"\t\t= "<<tobin(R,4)<<endl<<flush;
if(display) cout<<"        ----        ----        ----"<<endl<<flush;
```

```
#endif
    }
#ifdef USEDISP
if(display) cout<<"LR_end="<<tohex(out,8)<<endl<<flush;
#endif

    //populate RL with previous R and L
    //memcpy(&RL[0],R,4);
    //memcpy(&RL[4],L,4);
    *((long*)&RL[0])=*((long*)R);
    *((long*)&RL[4])=*((long*)L);
    //apply inverse permutation
#ifdef USEIP
    if(useip)
        IPINV(RL,out);
    else {
        IDENT(RL,out);
        if(display) cout<<"IP not used"<<endl<<flush; }
#else
    IPINV(RL,out);
#endif

#ifdef USEDISP
if(display) cout<<"plaintext="<<tobin(out,8)<<endl<<flush;
#endif

    return out;
}

uchar_ptr DES::decrypt(const uchar_ptr in, uchar_ptr out)
{ decrypt(in,out,16); }
uchar_ptr DES::decrypt3(const uchar_ptr in, uchar_ptr out)
{ decrypt(in,out,3); }
uchar_ptr DES::decrypt6(const uchar_ptr in, uchar_ptr out)
{ decrypt(in,out,6); }
```

SOURCE: **des/des.hh**

```
#ifndef _DES_HH
#define _DES_HH

#include "common.hh"
#include "perm.hh"
#include "sbox.hh"
#include "keys.hh"
```

```
class DES
{
    private:
        bool display;
        bool useip;
        bool userlswap;

        int numrounds;

#undef USEPERMCLASS
#define USEPERMCLASS

#ifndef USEPERMCLASS
        Permutation IDENT;
        Permutation IP;
        Permutation IPINV;
        Permutation P;
        Permutation E;
#endif

#ifdef USEPERMCLASS
        IP_perm IP;
        IDENT_perm IDENT;
        IPINV_perm IPINV;
        P_perm P;
        E_perm E;
#endif

        SBox S1;
        SBox S2;
        SBox S3;
        SBox S4;
        SBox S5;
        SBox S6;
        SBox S7;
        SBox S8;
        Keys K;
    public:
        NEWEXCEPTION(DESException);
        DES(void);
        ~DES(void);

        void setDisplay(bool b);
        void setUseIP(bool b);
        void setUseRLSwap(bool b);
```

```
        void setRounds(int r);

        void setKey(uchar_ptr k);
        //encrypt=false/decrypt=true
        uchar_ptr operator()(const uchar_ptr in, uchar_ptr out, bool
d=false);
        uchar_ptr encrypt(const uchar_ptr in, uchar_ptr out, int rounds);
        uchar_ptr decrypt(const uchar_ptr in, uchar_ptr out, int rounds);
        uchar_ptr encrypt(const uchar_ptr in, uchar_ptr out);
        uchar_ptr decrypt(const uchar_ptr in, uchar_ptr out);
        uchar_ptr encrypt3(const uchar_ptr in, uchar_ptr out);
        uchar_ptr decrypt3(const uchar_ptr in, uchar_ptr out);
        uchar_ptr encrypt6(const uchar_ptr in, uchar_ptr out);
        uchar_ptr decrypt6(const uchar_ptr in, uchar_ptr out);

        void DESRoundFunction(uchar_ptr in, uchar_ptr out, int round);
};


#endif /* _DES_HH */
```

SOURCE: **des/getmask.cc**

```
static char cvsid[] = { "@(#) $Id: getmask.cc,v 1.1 2001/11/16 11:37:15
stewart Exp $" };
#include "common.h"
USE(cvsid);

/*
 * test code for keys and key mask
 */

#include <fstream>
#include <iostream>

#include "common.hh"
#include "des.hh"

void usage(int argc, char* argv[])
{
    cerr<<"Usage: "<<argv[0]<<" -r round [ -s hex 48 bit subkey ] ";
    cerr<<"[ -S bin 48 bit subkey ]"<<endl;
    exit(-1);
}

int main(int argc, char* argv[])
```

```
{
    Keys K;
    uchar in_subkey[6];
    uchar out_subkey[8];
    uchar out_mask[8];
    int round;
    char *rarg=NULL, *sarg=NULL;

    if(argc!=3 && argc!=5)
        usage(argc,argv);

    if(argv[1][0]!='-' && argv[1][1]!='r')
        usage(argc,argv);

    rarg=argv[2];
    round=strtol(rarg,NULL,10);

    if(argc==5)
    {
        if(argv[3][0]!='-' && (argv[3][1]!='s' || argv[3][1]!='S'))
            usage(argc,argv);

        sarg=argv[4];
        if(argv[3][1]=='s')
            hexstrtobuf(sarg,in_subkey);
        else
            binstrtobuf(sarg,in_subkey);

    }

    K.setMask(out_mask,round);
    K.setParity(out_mask);
    cout<<"Round: "<<round<<endl<<flush;
    cout<<"Mask: "<<tohex(out_mask,8)<<" "<<tobin(out_mask,8)<<endl<<flush;

    K.buildK(out_mask);
    cout<<"Subkey: "<<tohex(K[round],6)<<"
"<<tobin(K[round],6)<<endl<<flush;

    if(sarg)
    {
        K.expandSubkey(in_subkey,out_subkey,round);
        cout<<"Subkey Part: "<<tohex(out_subkey,8);
        cout<<" "<<tobin(out_subkey,8)<<endl<<flush;
    }

    return 0;
```

```
}
```

SOURCE: **des/keys.cc**

```
static char cvsid[] = { "@(#) $Id: keys.cc,v 1.1 2001/11/16 11:37:15
stewart Exp $" };
#include "common.h"
USE(cvsid);

#include "keys.hh"

Keys::Keys(void)
{
    b_key=(uchar_ptr)&(key[0]);
    b_np_key=(uchar_ptr)&(np_key[0]);
    b_perm_key=(uchar_ptr)&(perm_key[0]);
    PC1.setPerm(Permutation::ID::PC1);
    PC2.setPerm(Permutation::ID::PC2);
    PC1_inv.setPerm(Permutation::ID::PC1_inv);
    PC2_inv.setPerm(Permutation::ID::PC2_inv);
    LS1.setPerm(LS1_perm,56);
    LS2.setPerm(LS2_perm,56);
    LS1_inv.setPerm(LS1_perm_inv,56);
    LS2_inv.setPerm(LS2_perm_inv,56);
}

Keys::~Keys(void)
{ }

void Keys::setKey(ullong k)
{
    *((ullong*)b_key)=k;
}
//assumes the block is 8 bytes long
void Keys::setKey(uchar_ptr k)
{
    for(int i=0;i<8;i++)
        b_key[i]=k[i];
}

uchar_ptr Keys::getKey(void)
{ return b_key; }

bool Keys::isKeyValid(void)
{
    short p;
```

```
    for(int i=0;i<8;i++)
    {
        p=0;
        for(int j=0;j<8;j++)
            if(((b_key[i])>>j)&0x1)
                p++;
        if(!(p&0x1))
            return false;
    }
    return true;
}
//8th bit of each byte is a parity bit
uchar_ptr Keys::setParity(void)
{
    return setParity(b_key);
}

uchar_ptr Keys::setParity(uchar_ptr k)
{
    short p;
    for(int i=0;i<8;i++)
    {
        p=0;
        for(int j=0;j<7;j++)
            if(((k[i])>>j)&0x1)
                p++;
        if(!(p&0x1))
        { //first 7 bits give even parity
            k[i]|=0x80;
        }
        else
            k[i]&=0x7f;
    }
    return k;
}

/* remove the parity bits from each byte */
ullong Keys::stripKey(uchar_ptr k)
{
    register unsigned long long tmp_key=0;
    for(int i=0;i<8;i++)
        tmp_key|=((ullong)(b_key[i]&0x7f))<<(i*7);
    *((ullong*)b_np_key)=tmp_key;
    if(k)
        *((ullong*)k)=tmp_key;
    return tmp_key;
}
```

```
const unsigned short Keys::LS1_perm[56]={
         2,  3,  4,  5,  6,  7,  8,  9,
        10, 11, 12, 13, 14, 15, 16, 17,
        18, 19, 20, 21, 22, 23, 24, 25,
        26, 27, 28,  1,
        30, 31, 32, 33, 34, 35, 36, 37,
        38, 39, 40, 41, 42, 43, 44, 45,
        46, 47, 48, 49, 50, 51, 52, 53,
        54, 55, 56, 29
    };
const unsigned short Keys::LS2_perm[56]={
         3,  4,  5,  6,  7,  8,  9, 10,
        11, 12, 13, 14, 15, 16, 17, 18,
        19, 20, 21, 22, 23, 24, 25, 26,
        27, 28,  1,  2,
        31, 32, 33, 34, 35, 36, 37, 38,
        39, 40, 41, 42, 43, 44, 45, 46,
        47, 48, 49, 50, 51, 52, 53, 54,
        55, 56, 29, 30
    };
//perform the cyclic left shit on the first and second 28 bits
//if i=1,2,9,16 shift by one otherwise by two
void Keys::LSi(uchar CiDi[7],int i)
{
    /* 1.........................22.........................5
       1.........................89.........................6
       <      ><      ><      ><  ><  ><      ><      ><      >

       Use permutations to perform the rotations
       A signed 56 bit permutation;
     */
    uchar copy[7];
    memcpy(copy,CiDi,7);
//    cout<<"["<<(i<10?"
":"")<<i<<"]"<<"copy="<<tobin(copy,7)<<endl<<flush;
    if(i==1 || i==2 || i==9 || i==16)
        LS1(copy,CiDi);
    else
        LS2(copy,CiDi);
//    cout<<"["<<(i<10?"
":"")<<i<<"]"<<"CiDi="<<tobin(CiDi,7)<<endl<<flush;
}

const unsigned short Keys::LS1_perm_inv[56]={
        28,  1,  2,  3,  4,  5,  6,  7,
         8,  9, 10, 11, 12, 13, 14, 15,
```

```
        16, 17, 18, 19, 20, 21, 22, 23,
        24, 25, 26, 27,
        56, 29, 30, 31, 32, 33, 34, 35,
        36, 37, 38, 39, 40, 41, 42, 43,
        44, 45, 46, 47, 48, 49, 50, 51,
        52, 53, 54, 55
    };
const unsigned short Keys::LS2_perm_inv[56]={
        27, 28,  1,  2,  3,  4,  5,  6,
         7,  8,  9, 10, 11, 12, 13, 14,
        15, 16, 17, 18, 19, 20, 21, 22,
        23, 24, 25, 26,
        55, 56, 29, 30, 31, 32, 33, 34,
        35, 36, 37, 38, 39, 40, 41, 42,
        43, 44, 45, 46, 47, 48, 49, 50,
        51, 52, 53, 54
    };
//perform the cyclic right shit on the first and second 28 bits
//if i=1,2,9,16 shift by one otherwise by two
void Keys::LSi_inv(uchar CiDi[7],int i)
{
    uchar copy[7];
    memcpy(copy,CiDi,7);
    if(i==1 || i==2 || i==9 || i==16)
        LS1_inv(copy,CiDi);
    else
        LS2_inv(copy,CiDi);
}

//build the key schedule
void Keys::buildK(void)
{
    //at this stage the parity bits of the key have been checked
    //and striped out
    uchar CiDi[7]; //Ci=lower 28 bits, Di=upper 28bits

    PC1(key,PC1_key);
    memcpy(CiDi,PC1_key,7);
//    cout<<"PC1_key:  "<<tobin(PC1_key,7)<<endl<<flush;
    for(int i=0;i<16;i++)
    {
        LSi(CiDi,i+1);
        PC2(CiDi,K[i]);
    }
}

void Keys::buildK(uchar_ptr k)
```

```
{
    setKey(k);
    if(!isKeyValid())
        throw KeyException("Parity check failed for key");
#if 0
    //this serves no purpose as b_np_key is never used
    stripKey();
#endif
    buildK();
}

uchar_ptr Keys::operator[](int i)
{
    /*
    if(i<0 || i>15)
        throw KeyException("Key Schedule index out of range");
    */
    return K[i-1];
}

/* this creates a mask representing the bits of the total
 * key that are used in round r
 * this can be calculated by by presenting expandSubkey()
 * with all ones 0xFFFFFFFFFFFF
 */
void Keys::setMask(uchar_ptr m, int r)
{
    static uchar ones[6]= { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };
    expandSubkey(ones,m,r);
}

/* this takes a 48bit subkey and expands it back into
 * the 64bits based on the round number
 */
void Keys::expandSubkey(uchar_ptr ink, uchar_ptr okey, int r)
{
    if(r>16 || r<1)
        throw KeyException("Round out of bounds");
    uchar bits56[7];
    // reverse PC-2
    PC2_inv(ink,bits56);
    // reverse shifts
    for(int i=r;i>0;i--)
        LSi_inv(bits56,i);
    // reverse PC-1
    PC1_inv(bits56,okey);
}
```

SOURCE: **des/keys.hh**

```
#ifndef _KEYS_HH
#define _KEYS_HH

#include "common.hh"
#include "perm.hh"

class Keys
{
    private:
        uchar key[8];
        uchar np_key[7]; //key without parity bits
        uchar perm_key[7];
        uchar_ptr b_key;
        uchar_ptr b_np_key;
        uchar_ptr b_perm_key;
        uchar K[16][6];  //key schedule that is built from the key
        uchar PC1_key[7];
        Permutation PC1;
        Permutation PC2;
        Permutation PC1_inv;
        Permutation PC2_inv;
        static const unsigned short LS1_perm[56];
        static const unsigned short LS2_perm[56];
        static const unsigned short LS1_perm_inv[56];
        static const unsigned short LS2_perm_inv[56];
        Permutation LS1;
        Permutation LS2;
        Permutation LS1_inv;
        Permutation LS2_inv;
        void LSi(uchar CiDi[7],int i);
        void LSi_inv(uchar CiDi[7],int i);
    public:
        NEWEXCEPTION(KeyException);
        Keys(void);
        ~Keys(void);
        void setKey(ullong k);
        void setKey(uchar_ptr k);
        uchar_ptr getKey(void);
        uchar_ptr setParity(void);
        static uchar_ptr setParity(uchar_ptr k);
        bool isKeyValid(void);  //check the parity bits and return
stripped key
        ullong stripKey(uchar_ptr k=NULL);
        void buildK(void);
```

```
        void buildK(uchar_ptr k);
        uchar_ptr operator[](int i); // returns keys from the schedule
        void setMask(uchar_ptr m, int r); //sets the mask for the key in
round r
        void expandSubkey(uchar_ptr ink, uchar_ptr okey, int r);
};

#endif /* _KEYS_HH */
```

SOURCE: **des/perm.cc**

```
static char cvsid[] = { "@(#) $Id: perm.cc,v 1.1 2001/11/16 11:37:15
stewart Exp $" };
#include "common.h"
USE(cvsid);

#include "perm.hh"

// Permutations needed by DES
const unsigned short Permutation::IDENT[64]={
     1,  2,  3,  4,  5,  6,  7,  8,
     9, 10, 11, 12, 13, 14, 15, 16,
    17, 18, 19, 20, 21, 22, 23, 24,
    25, 26, 27, 28, 29, 30, 31, 32,
    33, 34, 35, 36, 37, 38, 39, 40,
    41, 42, 43, 44, 45, 46, 47, 48,
    49, 50, 51, 52, 53, 54, 55, 56,
    57, 58, 59, 60, 61, 62, 63, 64
};
const unsigned short Permutation::IP[64]={
    58, 50, 42, 34, 26, 18, 10,  2,
    60, 52, 44, 36, 28, 20, 12,  4,
    62, 54, 46, 38, 30, 22, 14,  6,
    64, 56, 48, 40, 32, 24, 16,  8,
    57, 49, 41, 33, 25, 17,  9,  1,
    59, 51, 43, 35, 27, 19, 11,  3,
    61, 53, 45, 37, 29, 21, 13,  5,
    63, 55, 47, 39, 31, 23, 15,  7
};
const unsigned short Permutation::IPINV[64]={
    40,  8, 48, 16, 56, 24, 64, 32,
    39,  7, 47, 15, 55, 23, 63, 31,
    38,  6, 46, 14, 54, 22, 62, 30,
    37,  5, 45, 13, 53, 21, 61, 29,
    36,  4, 44, 12, 52, 20, 60, 28,
    35,  3, 43, 11, 51, 19, 59, 27,
```

```
    34,  2, 42, 10, 50, 18, 58, 26,
    33,  1, 41,  9, 49, 17, 57, 25
};
const unsigned short Permutation::P[32]={
    16,  7, 20, 21,
    29, 12, 28, 17,
     1, 15, 23, 26,
     5, 18, 31, 10,
     2,  8, 24, 14,
    32, 27,  3,  9,
    19, 13, 30,  6,
    22, 11,  4, 25
};
const unsigned short Permutation::P_inv[32]={
     9, 17, 23, 31,     // 1 - 4
    13, 28,  2, 18,     // 5 - 8
    24, 16, 30,  6,     // 9 - 12
    26, 20, 10,  1,     // 13 - 16
     8, 14, 25,  3,     // 17 - 20
     4, 29, 11, 19,     // 21 - 24
    32, 12, 22,  7,     // 25 - 28
     5, 27, 15, 21      // 29 - 32
};
const unsigned short Permutation::PC1[56]={
    57, 49, 41, 33, 25, 17,  9,
     1, 58, 50, 42, 34, 26, 18,
    10,  2, 59, 51, 43, 35, 27,
    19, 11,  3, 60, 52, 44, 36,
    63, 55, 47, 39, 31, 23, 15,
     7, 62, 54, 46, 38, 30, 22,
    14,  6, 61, 53, 45, 37, 29,
    21, 13,  5, 28, 20, 12,  4
};
const unsigned short Permutation::PC2[48]={
    14, 17, 11, 24,  1,  5,
     3, 28, 15,  6, 21, 10,
    23, 19, 12,  4, 26,  8,
    16,  7, 27, 20, 13,  2,
    41, 52, 31, 37, 47, 55,
    30, 40, 51, 45, 33, 48,
    44, 49, 39, 56, 34, 53,
    46, 42, 50, 36, 29, 32
};
const unsigned short Permutation::PC1_inv[64]={
     8, 16, 24,     56, 52, 44, 36,   0,        //  1 -  8
     7, 15, 23,     55, 51, 43, 35,   0,        //  9 - 16
     6, 14, 22,     54, 50, 42, 34,   0,        // 17 - 24
```

```
     5, 13, 21,     53, 49, 41, 33,   0,        // 25 - 32
     4, 12, 20, 28,     48, 40, 32,   0,        // 33 - 40
     3, 11, 19, 27,     47, 39, 31,   0,        // 41 - 48
     2, 10, 18, 26,     46, 38, 30,   0,        // 49 - 56
     1,  9, 17, 25,     45, 37, 29,   0         // 57 - 64
};
const unsigned short Permutation::PC2_inv[56]={
     5, 24,  7, 16,  6, 10, 20, //  1 -  7
    18,  0, 12,  3, 15, 23,  1, //  8 - 14
     9, 19,  2,  0, 14, 22, 11, // 15 - 21
     0, 13,  4,  0, 17, 21,  8, // 22 - 28
    47, 31, 27, 48, 35, 41,  0, // 29 - 35
    46, 28,  0, 39, 32, 25, 44, // 36 - 42
     0, 37, 34, 43, 29, 36, 38, // 43 - 49
    45, 33, 26, 42,  0, 30, 40  // 50 - 56
};
const unsigned short Permutation::E[48]={
    32,  1,  2,  3,  4,  5,
     4,  5,  6,  7,  8,  9,
     8,  9, 10, 11, 12, 13,
    12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21,
    20, 21, 22, 23, 24, 25,
    24, 25, 26, 27, 28, 29,
    28, 29, 30, 31, 32, 1
};
//////

Permutation::Permutation(void)
{
    count=0;
    perm=NULL;
    perm_size=0;
}

Permutation::~Permutation(void)
{
    delete perm;
}

void Permutation::setPermutation(const unsigned short *p, size_t s)
{
    count=0;
    perm_size=s;
    if(perm)
        delete perm;
    perm=new unsigned short[s];
```

```
    for(int i=0;i<s;i++)
        perm[i]=p[i]-1;
    /* perm=p; */
}
void Permutation::setPerm(const unsigned short p[], size_t s)
{ setPermutation(p,s); }
void Permutation::setPerm(ID::Id id)
{
    /* divide size by sizeof(unsigned short) */
    switch(id)
    {
        case ID::IDENT:
            setPerm(IDENT,sizeof(IDENT)/2);
            break;
        case ID::IP:
            setPerm(IP,sizeof(IP)/2);
            break;
        case ID::IPINV:
            setPerm(IPINV,sizeof(IPINV)/2);
            break;
        case ID::P:
            setPerm(P,sizeof(P)/2);
            break;
        case ID::P_inv:
            setPerm(P_inv,sizeof(P_inv)/2);
            break;
        case ID::PC1:
            setPerm(PC1,sizeof(PC1)/2);
            break;
        case ID::PC2:
            setPerm(PC2,sizeof(PC2)/2);
            break;
        case ID::E:
            setPerm(E,sizeof(E)/2);
            break;
        case ID::PC1_inv:
            setPerm(PC1_inv,sizeof(PC1_inv)/2);
            break;
        case ID::PC2_inv:
            setPerm(PC2_inv,sizeof(PC2_inv)/2);
            break;
        default:
            throw PermException("No such predefined exception");
    }
}

unsigned long long Permutation::operator()(unsigned long long in)
```

```
{
//    count++;
    return permute(perm,perm_size,in);
}
inline unsigned long long Permutation::permute(const unsigned short
*ext_perm,
                                        size_t size,
                                        unsigned long long in)
{
    register unsigned long long out=0;

    for(register int i=0;i<size;i++)
        out|=((0x1 & (in>>(ext_perm[i])))<<i);

    return out;
}

uchar_ptr Permutation::operator()(uchar_ptr in,uchar_ptr out)
{
//    count++;
    return permute(perm,perm_size,in,out);
}

inline void bitzero(unsigned char* buf, size_t n)
{
    int bytes=n/8;
    int bits=n%8;
    bzero(buf,bytes);
    if(bits)
        buf[bytes+1]&=~((1<<bits)-1);

}

inline uchar_ptr Permutation::permute(const unsigned short *ext_perm,
                               size_t size,
                               const uchar_ptr in, uchar_ptr out)
{
    register unsigned short out_byte, out_bit;
    register unsigned short in_pos, out_pos;
    bitzero(out,size); // this is needed because below we use a bitwise or.

    for(out_pos=0;out_pos<size;out_pos++)
    {
#if 0
        //optimise divide and mod 8
        out_bit=out_pos&0x07;
        out_byte=out_pos>>3;
```

```
        in_pos=ext_perm[out_pos];

        out[out_byte]|=((in[in_pos>>3]>>(in_pos&0x07))&0x1)<<out_bit;
#endif
#if 1
        out_bit=out_pos%8;
        out_byte=out_pos/8;
        in_pos=ext_perm[out_pos];
        //NB 0xffff = (unsigned short)(0 - 1);
        if(in_pos==0xffff) //set the bit to zero
            out[out_byte]&=~(0x1<<out_bit);
        else
            out[out_byte]|=((in[in_pos/8]>>(in_pos%8))&0x1)<<out_bit;
#endif
    }
    return out;
}

/* construct an inverse of the given permutaion */
void Permutation::invert(const unsigned short *in_p, size_t in_s,
                              unsigned short *out_p,size_t out_s)
{
    /* initialise the inverse to blank */
    for(int i=0;i<out_s;i++)
        out_p[i]=0xffff;

    /* invert permutation */
    int pos;
    for(int i=0;i<in_s;i++)
    {
        pos=in_p[i];
        if(pos==0xffff)
            continue;
        if(pos<0 || pos>=out_s)
            throw PermException("Invert failed - input perm pos out of
bounds");
        out_p[pos]=i;
    }
}

/* Permutation out is overwritten with the inverse of this perm */
void Permutation::invert(Permutation &out)
{
    unsigned short *outp=NULL;
    size_t outsize=0;

    if(perm == NULL || perm_size == 0)
```

```
        throw PermException("No permutation array set");

    /* find the outperm size by looking for the input max */
    for(int i=0;i<perm_size;i++)
    {
        if(perm[i]>outsize)
            outsize=perm[i];
    }
    outsize++;

    outp=new unsigned short[outsize];

    invert(perm,perm_size,outp,outsize);

    /* normalise the outperm to start at 1 */
    for(int i=0;i<outsize;i++)
        outp[i]++;

    out.setPerm(outp,outsize);

    delete outp;
}

void Permutation::print(const char* nm, int width)
{
    if(perm==NULL)
        throw PermException("Permuation is null");
    cout<<"const unsigned short "<<nm<<"["<<perm_size<<"]={"<<endl;
    cout<<"    ";
    for(int i=0;i<perm_size-1;i++)
    {
        cout<<perm[i]+1<<", ";
        if((i+1)%width==0)
            cout<<"\t// "<<i+2-width<<" - "<<i+1<<endl<<"    ";
    }
    cout<<perm[perm_size-1]+1;
    cout<<"\t// "<<(
            (perm_size%width)==0?
            (perm_size-width+1):
            (perm_size-(perm_size%width))
            );
    cout<<" - "<<perm_size<<endl;
    cout<<"};"<<endl<<flush;
}
#define NEWPERMDEF(pname)                                                \
pname##_perm::pname##_perm()                                             \
{                                                                        \
```

```
    setPerm(pname##,sizeof(pname##)/sizeof(short));                    \
}                                                                      \
uchar_ptr pname##_perm::operator()(uchar_ptr in,uchar_ptr out)         \
{                                                                      \
    return Permutation::operator()(in,out);                            \
}

NEWPERMDEF(IP);
NEWPERMDEF(IPINV);
NEWPERMDEF(IDENT);
NEWPERMDEF(P_inv);
NEWPERMDEF(PC1);
NEWPERMDEF(PC2);
NEWPERMDEF(PC1_inv);
NEWPERMDEF(PC2_inv);

//NEWPERMDEF(E);
E_perm::E_perm()
{
    setPerm(E,sizeof(E)/sizeof(short));
}
uchar_ptr E_perm::operator()(uchar_ptr in,uchar_ptr out)
{
    //from 32 bits to 48 bits
#if 0
    return Permutation::operator()(in,out);
#else

    //32,  1,  2,  3,  4,  5,  4,  5,
    out[0]=((in[3]>>7)&0x01) | ((in[0]<<1)&0x3e) | ((in[0]<<3)&0xc0);

    // 6,  7,  8,  9,  8,  9, 10, 11,
    out[1]=((in[0]>>5)&0x07) | ((in[1]<<3)&0x08) |
           ((in[0]>>3)&0x10) | ((in[1]<<5)&0xe0);

    //12, 13, 12, 13, 14, 15, 16, 17,
    out[2]=((in[1]>>3)&0x03) | ((in[1]>>1)&0x7c) | ((in[2]<<7)&0x80);

    //16, 17, 18, 19, 20, 21, 20, 21,
    out[3]=((in[1]>>7)&0x01) | ((in[2]<<1)&0x3e) | ((in[2]<<3)&0xc0);

    //22, 23, 24, 25, 24, 25, 26, 27,
    out[4]=((in[2]>>5)&0x07) | ((in[3]<<3)&0x08) |
           ((in[2]>>3)&0x10) | ((in[3]<<5)&0xe0);

    //28, 29, 28, 29, 30, 31, 32, 1
    out[5]=((in[3]>>3)&0x03) | ((in[3]>>1)&0x7c) | ((in[0]<<7)&0x80);
```

```
#endif
}

//NEWPERMDEF(P);
P_perm::P_perm()
{
    setPerm(P,sizeof(P)/sizeof(short));
}
uchar_ptr P_perm::operator()(uchar_ptr in,uchar_ptr out)
{
    //from 32 bits to 32 bits
#if 0
    return Permutation::operator()(in,out);
#else
    //16,  7, 20, 21, 29, 12, 28, 17,
    out[0]=((in[1]>>7)&0x01) | ((in[0]>>5)&0x02) |
           ((in[2]>>1)&0x04) | ((in[2]>>1)&0x08) |
           ((in[3]   )&0x10) | ((in[1]<<2)&0x20) |
           ((in[3]<<3)&0x40) | ((in[2]<<7)&0x80);
    // 1, 15, 23, 26,  5, 18, 31, 10,
    out[1]=((in[0]   )&0x01) | ((in[1]>>5)&0x02) |
           ((in[2]>>4)&0x04) | ((in[3]<<2)&0x08) |
           ((in[0]   )&0x10) | ((in[2]<<4)&0x20) |
           ((in[3]   )&0x40) | ((in[1]<<6)&0x80);
    //  0    1      2      3
    //[0 7][8 15][16 23][24 31]
    //[1 8][9 16][17 24][25 32]

    // 2,  8, 24, 14, 32, 27,  3,  9,
    out[2]=((in[0]>>1)&0x01) | ((in[0]>>6)&0x02) |
           ((in[2]>>5)&0x04) | ((in[1]>>2)&0x08) |
           ((in[3]>>3)&0x10) | ((in[3]<<3)&0x20) |
           ((in[0]<<4)&0x40) | ((in[1]<<7)&0x80);
    //19, 13, 30,  6, 22, 11,  4, 25
    out[3]=((in[2]>>2)&0x01) | ((in[1]>>3)&0x02) |
           ((in[3]>>3)&0x04) | ((in[0]>>2)&0x08) |
           ((in[2]>>1)&0x10) | ((in[1]<<3)&0x20) |
           ((in[0]<<3)&0x40) | ((in[3]<<7)&0x80);
#endif
}
```

SOURCE: **des/perm.hh**

```
#ifndef _PERM_HH
#define _PERM_HH
```

```
#include "common.hh"

class Permutation
{
    private:
        int count;
        unsigned short *perm;
        size_t perm_size; //number of bits

    protected:
        unsigned long long permute(const unsigned short *ext_perm,
                                   size_t size, unsigned long long in);
        uchar_ptr permute(const unsigned short *ext_perm,
                          size_t size, const uchar_ptr in, uchar_ptr out);

    public:

        class ID
        {
            public:
                enum Id { IDENT, IP, IPINV, P, PC1,
                          PC2, E, PC1_inv, PC2_inv, P_inv };
        };
        // Permutation
        const static unsigned short IDENT[64];
        const static unsigned short IP[64];
        const static unsigned short IPINV[64];
        const static unsigned short P[32];
        const static unsigned short P_inv[32];
        const static unsigned short PC1[56];
        const static unsigned short PC2[48];
        const static unsigned short PC1_inv[64];
        const static unsigned short PC2_inv[56];
        const static unsigned short E[48];
        //////

        NEWEXCEPTION(PermException);
        Permutation(void);
        ~Permutation(void);

        void setPermutation(const unsigned short *p, size_t s);
        void setPerm(const unsigned short p[], size_t s);
        void setPerm(ID::Id id);

        void print(const char* nm, int width);

        void invert(const unsigned short *in_p, size_t in_s,
```

```
                        unsigned short *out_p,size_t out_s);
        void invert(Permutation &out);

        virtual unsigned long long operator()(unsigned long long in);
        virtual uchar_ptr operator()(uchar_ptr in,uchar_ptr out);
};
#if 1
#define NEWPERMCLASS(pname)                     \
class pname##_perm: public Permutation          \
{                                               \
    public:                                     \
        pname##_perm();                         \
        uchar_ptr operator()(uchar_ptr in,uchar_ptr out);       \
}

NEWPERMCLASS(IP);
NEWPERMCLASS(IPINV);
NEWPERMCLASS(IDENT);
NEWPERMCLASS(P);
NEWPERMCLASS(P_inv);
NEWPERMCLASS(PC1);
NEWPERMCLASS(PC2);
NEWPERMCLASS(PC1_inv);
NEWPERMCLASS(PC2_inv);
NEWPERMCLASS(E);
#endif

#endif /* _PERM_HH */
```

SOURCE: **des/rundes.cc**

```
static char cvsid[] = { "@(#) $Id: rundes.cc,v 1.2 2001/11/17 15:29:25
stewart Exp $" };
#include "common.h"
USE(cvsid);

/* This provides commandline access to DES
 * The first argument is the key in hex,
 * the second is the plaintext in hex
 * the output is the ciphertext in hex
 */

#include <fstream>
#include <iostream>
```

```
#include "common.hh"
#include "des.hh"

void usage(int argc, char* argv[])
{
    cerr<<"Usage: "<<argv[0]<<" -d key ciphertext | -e key
plaintext"<<endl;
    exit(-1);
}

int main(int argc, char* argv[])
{
    DES des;
    uchar ctxt_out[8];
    uchar ptxt_out[8];
    uchar ctxt_in[8];
    uchar ptxt_in[8];
    uchar key[8];
    char *karg, *carg, *parg;
    int rounds=16;

#if 0
    des.setDisplay(true);
    des.setUseIP(true);
    des.setUseRLSwap(true);
#endif
    des.setDisplay(true);
    des.setUseIP(true);
    des.setUseRLSwap(true);
    des.setRounds(rounds);

    if(argc!=4)
        usage(argc,argv);

    if(argv[1][0]!='-')
        usage(argc,argv);
    try {
        switch(argv[1][1])
        {
            case 'd': //decrypt
                karg=argv[2];
                carg=argv[3];
                hexstrtobuf(karg,key);
                hexstrtobuf(carg,ctxt_in);

                Keys::setParity(key);
                des.setKey(key);
```

```
                des.decrypt(ctxt_in,ptxt_out);

                cout<<"k"<<":"<<tohex(key,8)<<" "<<flush;
                cout<<"p"<<":"<<tohex(ptxt_out,8)<<" "<<flush;
                cout<<"c"<<":"<<tohex(ctxt_in,8)<<flush;
                cout<<endl<<flush;
                break;
            case 'e': //encrypt
                karg=argv[2];
                parg=argv[3];
                hexstrtobuf(karg,key);
                hexstrtobuf(parg,ptxt_in);

                Keys::setParity(key);
                des.setKey(key);
                des.encrypt(ptxt_in,ctxt_out);

                cout<<"k"<<":"<<tohex(key,8)<<" "<<flush;
                cout<<"p"<<":"<<tohex(ptxt_in,8)<<" "<<flush;
                cout<<"c"<<":"<<tohex(ctxt_out,8)<<flush;
                cout<<endl<<flush;
                break;
            default:
                usage(argc,argv);
        }
    } catch(DES::DESException e) {
        cerr<<"Caught DESExcetpion: "<<e.getMessage()<<endl<<flush;
        return -1;
    } catch(Keys::KeyException e) {
        cerr<<"Caught KeyExcetpion: "<<e.getMessage()<<endl<<flush;
        return -1;
    }


    return 0;
}
```

SOURCE: **des/runperm.cc**

```
static char cvsid[] = { "@(#) $Id: runperm.cc,v 1.1 2001/11/16 11:37:15
stewart Exp $" };
#include "common.h"
USE(cvsid);

#include <fstream>
#include <iostream>
```

```
#include "common.hh"
#include "perm.hh"

int main(int argc, char* argv[])
{
    Permutation P;
    Permutation P_inv;

    P.setPerm(Permutation::ID::P);
    P.invert(P_inv);

    P_inv.print("Permutation::P_inv",4);

    return 0;
}
```

SOURCE: **des/sbox.cc**

```
static char cvsid[] = { "@(#) $Id: sbox.cc,v 1.2 2001/11/25 10:46:00
stewart Exp $" };
#include "common.h"
USE(cvsid);

#include "sbox.hh"

#define USEARY
//#undef USEARY

/*
 * S-Boxes
 *
 * - These provide the non-linear compenent of DES.
 * - An S-Box can be represented as a 4x16 array of
 *   numbers in the range [0,15].
 * - S-Box can be thought of as a function that takes
 *   6 bits in and return 4 bits.
 * - Given B=b_1 b_2 ... b_6 use b_1 b_6 as in index
 *   to the row and b_2 b_3 b_4 b_5 as in index to
 *   the column.
 */

const unsigned char SBox::S1[4][16]= {
    { 14,  4, 13,  1,  2, 15, 11,  8,  3, 10,  6, 12,  5,  9,  0,  7 },
    {  0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,  8 },
    {  4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,  0 },
```

```
    { 15, 12,  8,  2,  4,  9,  1,  7,  5, 11,  3, 14, 10,  0,  6, 13 }
};
const unsigned char SBox::S2[4][16]= {
    { 15,  1,  8, 14,  6, 11,  3,  4,  9,  7,  2, 13, 12,  0,  5, 10 },
    {  3, 13,  4,  7, 15,  2,  8, 14, 12,  0,  1, 10,  6,  9, 11,  5 },
    {  0, 14,  7, 11, 10,  4, 13,  1,  5,  8, 12,  6,  9,  3,  2, 15 },
    { 13,  8, 10,  1,  3, 15,  4,  2, 11,  6,  7, 12,  0,  5, 14,  9 }
};
const unsigned char SBox::S3[4][16]= {
    { 10,  0,  9, 14,  6,  3, 15,  5,  1, 13, 12,  7, 11,  4,  2,  8 },
    { 13,  7,  0,  9,  3,  4,  6, 10,  2,  8,  5, 14, 12, 11, 15,  1 },
    { 13,  6,  4,  9,  8, 15,  3,  0, 11,  1,  2, 12,  5, 10, 14,  7 },
    {  1, 10, 13,  0,  6,  9,  8,  7,  4, 15, 14,  3, 11,  5,  2, 12 }
};
const unsigned char SBox::S4[4][16]= {
    {  7, 13, 14,  3,  0,  6,  9, 10,  1,  2,  8,  5, 11, 12,  4, 15 },
    { 13,  8, 11,  5,  6, 15,  0,  3,  4,  7,  2, 12,  1, 10, 14,  9 },
    { 10,  6,  9,  0, 12, 11,  7, 13, 15,  1,  3, 14,  5,  2,  8,  4 },
    {  3, 15,  0,  6, 10,  1, 13,  8,  9,  4,  5, 11, 12,  7,  2, 14 }
};
const unsigned char SBox::S5[4][16]= {
    {  2, 12,  4,  1,  7, 10, 11,  6,  8,  5,  3, 15, 13,  0, 14,  9 },
    { 14, 11,  2, 12,  4,  7, 13,  1,  5,  0, 15, 10,  3,  9,  8,  6 },
    {  4,  2,  1, 11, 10, 13,  7,  8, 15,  9, 12,  5,  6,  3,  0, 14 },
    { 11,  8, 12,  7,  1, 14,  2, 13,  6, 15,  0,  9, 10,  4,  5,  3 }
};
const unsigned char SBox::S6[4][16]= {
    { 12,  1, 10, 15,  9,  2,  6,  8,  0, 13,  3,  4, 14,  7,  5, 11 },
    { 10, 15,  4,  2,  7, 12,  9,  5,  6,  1, 13, 14,  0, 11,  3,  8 },
    {  9, 14, 15,  5,  2,  8, 12,  3,  7,  0,  4, 10,  1, 13, 11,  6 },
    {  4,  3,  2, 12,  9,  5, 15, 10, 11, 14,  1,  7,  6,  0,  8, 13 }
};
const unsigned char SBox::S7[4][16]= {
    {  4, 11,  2, 14, 15,  0,  8, 13,  3, 12,  9,  7,  5, 10,  6,  1 },
    { 13,  0, 11,  7,  4,  9,  1, 10, 14,  3,  5, 12,  2, 15,  8,  6 },
    {  1,  4, 11, 13, 12,  3,  7, 14, 10, 15,  6,  8,  0,  5,  9,  2 },
    {  6, 11, 13,  8,  1,  4, 10,  7,  9,  5,  0, 15, 14,  2,  3, 12 }
};
const unsigned char SBox::S8[4][16]= {
    { 13,  2,  8,  4,  6, 15, 11,  1, 10,  9,  3, 14,  5,  0, 12,  7 },
    {  1, 15, 13,  8, 10,  3,  7,  4, 12,  5,  6, 11,  0, 14,  9,  2 },
    {  7, 11,  4,  1,  9, 12, 14,  2,  0,  6, 10, 13, 15,  3,  5,  8 },
    {  2,  1, 14,  7,  4, 10,  8, 13, 15, 12,  9,  0,  3,  5,  6, 11 }
};


SBox::~SBox(void)
```

```
{ }
SBox::SBox(void)
{
    count=0;
    clearTable();
}
SBox::SBox(unsigned char tbl[4][16])
{
    SBox();
    setTable(tbl);
}
void SBox::setTable(unsigned char tbl[4][16])
{
    register uchar s,t; //sbox out
    for(int r=0;r<4;r++)
        for(int c=0;c<16;c++)
        {
            t=tbl[r][c];
            //reverse the bit order of the sbox entries
            s=((t>>3)&0x01)|((t>>1)&0x02)|((t<<1)&0x04)|((t<<3)&0x08);
            sbox_tbl[r][c]=s;
        }
#ifdef USEARY
    setArray(tbl);
#endif
}
void SBox::clearTable(void)
{
    for(int r=0;r<4;r++)
        for(int c=0;c<16;c++)
            sbox_tbl[r][c]=0xff;
#ifdef USEARY
    clearArray();
#endif
}
void SBox::clearArray(void)
{
    for(int i=0;i<64;i++)
        sbox_ary[i]=0xff;
}
void SBox::setArray(unsigned char tbl[4][16])
{
    uchar s,t; //sbox out
    int r,c; //row, column
    int in;
    for(in=0;in<64;in++)
    {
```

```
        r=((in>>5)&0x01)|((in<<1)&0x02);
        c=((in>>4)&0x01)|((in>>2)&0x02)|((in)&0x04)|((in<<2)&0x08);
        t=tbl[r][c];
        //reverse the bit order of the sbox entries
        s=((t>>3)&0x01)|((t>>1)&0x02)|((t<<1)&0x04)|((t<<3)&0x08);
#if 0
        r=((in>>4)&0x02)|((in>>0)&0x01);
        c=((in>>1)&0x0f);
        t=tbl[r][c];
#endif
        sbox_ary[in]=s;
    }
}

unsigned char SBox::operator()(int r, int c)
{
    if(r<0 || r>3)
        throw SBoxException("Row out of bounds");
    if(c<0 || c>15)
        throw SBoxException("Column out of bounds");

    return sbox_tbl[r][c];
}

unsigned char SBox::calc(unsigned char in)
{
    return operator()(in);
}
/* this implement the hard work */
inline unsigned char SBox::operator()(unsigned char in)
{
//    register uchar s,t; //sbox out
    /* sanity check
     *  - the input value must be in the range [0,64)
     */
    /*
    if(in>63)
        throw SBoxException("input out of range");

    count++;
    */

    /* One would expect b_1 to be the least signification but it is the
most
     * signification bit when performing look ups in the S-box tables
     */
#if 0
```

```
    // with b_1 least significant and b_6 most significant
    // b_1 b_6
    r=(in&0x01)|((in>>4)&0x02);
    // b_2 b_3 b_4 b_5
    c=(in>>1)&0x0f;
    s=sbox_tbl[r][c];
#endif
#ifndef USEARY
    register int r,c; // row and column
    // with b_6 least significant and b_1 most significant
    r=((in>>5)&0x01)|((in<<1)&0x02);
    c=((in>>4)&0x01)|((in>>2)&0x02)|((in)&0x04)|((in<<2)&0x08);

    return sbox_tbl[r][c];
#endif
#ifdef USEARY
    //see setArray
    return sbox_ary[in];
#endif

//    cout<<"{"<<tobit(in,6)<<","<<r<<","<<c<<","<<tobit(s,4)<<"}"<<flush;

//    return s;
}
```

SOURCE: **des/sbox.hh**

```
#ifndef _SBOX_HH
#define _SBOX_HH

#include "common.hh"

class SBox
{
    private:
        unsigned char sbox_tbl[4][16];
        unsigned char sbox_ary[64];
        int count;
    public:
        //DES SBoxes
        const static unsigned char S1[4][16];
        const static unsigned char S2[4][16];
        const static unsigned char S3[4][16];
        const static unsigned char S4[4][16];
        const static unsigned char S5[4][16];
        const static unsigned char S6[4][16];
```

```
        const static unsigned char S7[4][16];
        const static unsigned char S8[4][16];
        //////
        NEWEXCEPTION(SBoxException);
        SBox(void);
        ~SBox(void);
        SBox(unsigned char tbl[4][16]);

        void setTable(unsigned char tbl[4][16]);
        void clearTable(void);
        void setArray(unsigned char tbl[4][16]);
        void clearArray(void);

        unsigned char operator()(unsigned char in);
        unsigned char operator()(int r, int c);
        unsigned char calc(unsigned char in);

};
#endif /* _SBOX_HH */
```

Appendix B

Differential Cryptanalysis Implementation

Differential Cryptanalysis is a powerful method for breaking DES like ciphers. To help my understanding I wrote a C++ implementation that could break 3-round DES. The implementation embodies most of the important aspects of differential cryptanalysis but does not incur the complexity of an attack against DES with more rounds.

In §B.1 I show the output of the calculations required to recover the encryption key.

In §B.2 I list the C++ source code that implements differential cryptanalysis of 3-round DES.

B.1 Results

Below is the output showing the process of recovering the encryption key. The key is recovered using test pairs where the plaintext pairs were been chosen to have a particular XOR difference. The recovered key is 1A624C89520DEC46.

OUTPUT: **Differential Cryptanalysis of 3-Round DES**

```
------------ Test Pairs ------------
---plaintext---- -- ---ciphertext---
cb80180dc2052a97 -- c13cad7c1a5876f0
c28c2500c2052a97 -- 59ec6fe383063d52
---------------- -- ----------------
3d3fcc7b24059435 -- 36df21feaf30ca8d
5c2bf10e24059435 -- c9690a21dbc4cf78
---------------- -- ----------------
3c666742ef2ef897 -- ffd51aeb89089445
b4578c3aef2ef897 -- bc61001d0d52ad88
---------------- -- ----------------
------------ tabulating ------------
---- Calculate the Input/Output Differences (E,E*,C') ----
----------- Pair 1 -----------
L0   = 11001011 10000000 00011000 00001101
L0*  = 11000010 10001100 00100101 00000000
L3   = 11000001 00111100 10101101 01111100
```

```
L3*  = 01011001 11101100 01101111 11100011
R3   = 00011010 01011000 01110110 11110000
R3*  = 10000011 00000110 00111101 01010010
R3'  = 10011001 01011110 01001011 10100010
L0'  = 00001001 00001100 00111101 00001101
R3' xor L0'  = 10010000 01010010 01110110 10101111
C'[0] = 00110001 00100111 00101101 11100110
E [0] = 01100000 00101001 11111001 01010101 10101011 11111001
E*[0] = 10101111 00111111 01011000 00110101 11111111 00000110

----------- Pair 2 -----------
L0   = 00111101 00111111 11001100 01111011
L0*  = 01011100 00101011 11110001 00001110
L3   = 00110110 11011111 00100001 11111110
L3*  = 11001001 01101001 00001010 00100001
R3   = 10101111 00110000 11001010 10001101
R3*  = 11011011 11000100 11001111 01111000
R3'  = 01110100 11110100 00000101 11110101
L0'  = 01100001 00010100 00111101 01110101
R3' xor L0'  = 00010101 11100000 00111000 10000000
C'[1] = 10000000 00010110 10101011 00000001
E [1] = 00011010 11010110 11111110 10010000 00111111 11111100
E*[1] = 11100101 00101011 01010010 10000101 01000001 00000011

----------- Pair 3 -----------
L0   = 00111100 01100110 01100111 01000010
L0*  = 10110100 01010111 10001100 00111010
L3   = 11111111 11010101 00011010 11101011
L3*  = 10111100 01100001 00000000 00011101
R3   = 10001001 00001000 10010100 01000101
R3*  = 00001101 01010010 10101101 10001000
R3'  = 10000100 01011010 00111001 11001101
L0'  = 10001000 00110001 11101011 01111000
R3' xor L0'  = 00001100 01101011 11010010 10110101
C'[2] = 01101101 01110110 00100010 10001110
E [2] = 11111111 11111110 10101010 10001111 01010111 01010111
E*[2] = 11011111 10000011 00000010 10000000 00000000 11111011

---- Find suggested subkey bits T_j(E,E*,C') ----
----------- Pair 1 -----------
[ 011000, 101011, 0011, 1 ] {14:011100,61:101111}
[ 000010, 110011, 0001, 2 ] {30:011110, 7:111000, 9:100100,11:110100,
                             15:111100,61:101111,36:001001,40:000101,
                             42:010101,44:001101}
[ 100111, 111101, 0010, 3 ] {59:110111,50:010011,45:101101,36:001001,
                             22:011010, 0:000000}
[ 111001, 011000, 0111, 4 ] {38:011001,33:100001,42:010101,51:110011,
```

```
                              7:111000, 0:000000,11:110100,18:010010}
[ 010101, 001101, 0010, 5 ] {43:110101,45:101101, 0:000000, 1:100000,
                              6:011000, 7:111000,26:010110,28:001110}
[ 011010, 011111, 1101, 6 ] {20:001010,19:110010,16:000010,25:100110,
                             49:100011,60:001111,59:110111,56:000111}
[ 101111, 111100, 1110, 7 ] {56:000111,53:101011,55:111011,54:011011,
                             49:100011,50:010011,43:110101,25:100110,
                             10:010100, 5:101000, 4:001000, 7:111000,
                              0:000000, 3:110000}
[ 111001, 000110, 0110, 8 ] {39:111001,35:110001,51:110011,12:001100,
                             28:001110,24:000110}

----------- Pair 2 -----------
[ 000110, 111001, 1000, 1 ] { 2:010000, 6:011000,57:100111,61:101111}
[ 101101, 010010, 0000, 2 ] {40:000101,62:011111, 1:100000,23:111010}
[ 011011, 101101, 0001, 3 ] {50:010011,41:100101,17:100010,10:010100}
[ 111110, 010010, 0110, 4 ] {13:101100,11:110100, 6:011000, 0:000000,
                             61:101111,59:110111,54:011011,48:000011}
[ 100100, 100001, 1010, 5 ] {13:101100, 1:100000, 0:000000, 2:010000,
                              6:011000,17:100010,18:010010,22:011010,
                             41:100101,40:000101,42:010101,46:011101,
                             37:101001,57:100111,58:010111,62:011111}
[ 000011, 010100, 1011, 6 ] {51:110011,54:011011,55:111011,56:000111,
                              2:010000, 9:100100,12:001100,13:101100}
[ 111111, 000100, 0000, 7 ] {56:000111,15:111100}
[ 111100, 000011, 0001, 8 ] {29:101110,28:001110,22:011010,41:100101,
                             35:110001,34:010001}

----------- Pair 3 -----------
[ 111111, 110111, 0110, 1 ] {61:101111,57:100111,55:111011,53:101011,
                             51:110011,49:100011,44:001101,40:000101}
[ 111111, 111000, 1101, 2 ] {40:000101,39:111001,36:001001,31:111110,
                             28:001110,16:000010}
[ 111010, 001100, 0111, 3 ] {26:010110, 1:100000,50:010011,41:100101}
[ 101010, 000010, 0110, 4 ] { 5:101000, 0:000000,61:101111,56:000111}
[ 100011, 100000, 0010, 5 ] {54:011011,33:100001,17:100010, 6:011000}
[ 110101, 000000, 0010, 6 ] {41:100101,56:000111, 2:010000,19:110010}
[ 011101, 000011, 1000, 7 ] {42:010101,38:011001,35:110001,61:101111,
                             56:000111,52:001011,10:010100, 0:000000,
                              1:100000,30:011110,31:111110,20:001010}
[ 010111, 111011, 1110, 8 ] {46:011101,35:110001, 8:000100, 9:100100,
                             12:001100,13:101100, 0:000000, 1:100000,
                              4:001000, 5:101000}

------------ collecting ------------
61,40,50,0,6,56,56,35
---------- collected bits ----------
```

```
realsub  = 10111100 01010100 11000000 01100000 01110001 11110001
subkey48 = 10111100 01010100 11000000 01100000 01110001 11110001
mask   = 11111110 11111110 11011010 10111110 11111110 11110010 11101100 01111110
subkey = 00011010 01100010 01001000 10001000 01010010 00000000 11101100 01000110
realkey= 00011010 01100010 01001100 10001001 01010010 00001101 11101100 01000110
----------- brute search -----------
.
------------- key found ------------

Input Key:  1a624c89520dec46
Output Key: 1a624c89520dec46
```

B.2 Source Code

```
diffcrypt/
            3round/
                        Makefile
                        3round.cc
                        round3dd.cc
                        round3dd.hh
            brutesearch/
                        Makefile
                        bruteDES.cc
                        bruteDES.hh
                        runBruteDES.cc
```

SOURCE: **diffcrypt/3round/Makefile**

```
# Makefile for des implementation

TOP=../../
SRC+=round3dd.cc
include $(TOP)/config.mk
LIBS+=-lsg_des -lsg_brute_des

all: progs

progs: 3round

libs:

3round: $(OBJS) 3round.cc
        $(CXX) $(CXXFLAGS) $(INCLUDES) $(LIBDIR) -o 3round 3round.cc
$(OBJS) $(LIBS)
```

```
3test: $(OBJS) 3test.cc
        $(CXX) $(CXXFLAGS) $(INCLUDES) $(LIBDIR) -o 3test 3test.cc $(OBJS)
$(LIBS)

clean:
        rm -f *.o
        rm -f *~
        rm -f .depend
        rm -f 3round
```

SOURCE: **diffcrypt/3round/3round.cc**

```
static char cvsid[] = { "@(#) $Id: 3round.cc,v 1.1 2001/11/16 11:37:16
stewart Exp $" };
#include "common.h"
USE(cvsid);

#include <iostream>

#include "round3dd.hh"

void usage(int argc, char* argv[])
{
    cerr<<"Usage: "<<argv[0]<<" -k hex-key|-K bin-key"<<endl<<flush;
    exit(1);
}

int main(int argc, char* argv[])
{
    uchar in_key[8];
    uchar out_key[8];
    char* karg=NULL;

    if(argc!=3)
        usage(argc,argv);

    if(argv[1][0]!='-' || (argv[1][1]!='k' && argv[1][1]!='K'))
        usage(argc,argv);

    karg=argv[2];
    if(argv[1][1]=='k')
        hexstrtobuf(karg,in_key);
    else
```

```
        binstrtobuf(karg,in_key);

    try {
      Round3DD R3;

      R3.setKey(in_key);
      R3.recover(out_key);
    } catch(Round3DD::R3DDException e) {
        cerr<<"Caught R3DDException: <"<<e.getMessage()<<">"<<endl<<flush;
        return 0;
    } catch(Exception e) {
        cerr<<"Caught Exception: <"<<e.getMessage()<<">"<<endl<<flush;
        return 0;
    } catch(...) {
        cerr<<"Caught unknown exception"<<endl<<flush;
        return 0;
    }


    cout<<endl;
    cout<<"Input Key:  "<<tohex(in_key,8)<<endl<<flush;
    cout<<"Output Key: "<<tohex(out_key,8)<<endl<<flush;

    return 0;
}
```

SOURCE: **diffcrypt/3round/round3dd.cc**

```
static char cvsid[] = { "@(#) $Id: round3dd.cc,v 1.2 2001/11/17 15:29:27
stewart Exp $" };
#include "common.h"
USE(cvsid);
/*
 * This code implements a differential cryptanalytic attack
 * on 3-round DES.
 *
 * This particular implementation hardcodes the characteristic
 * used. That is it hardcodes the input difference and output
 * difference used to find (in this case construct) a right pair,
 * furthermore we then know up front which key bits are going
 * to be recovered, since we are recovering the key from a fixed
 * round and thus from a fixed position in the key schedule.
 *
 * Once we have recovered the bit values of the DES subkey we
 * proceed to find the remainder using a brute force/exhaustive search.
 *
```

```
 * - Display the key used.
 * - Initialise with key we are going to recover.
 * - Construct plaintxt/ciphertxt pairs with difference.
 * - Calculate and tabulate the suggested subkey bits.
 * - Recover the remainder of the key via an exhaustive search.
 * - Display the key found.
 */

#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <iostream.h>
#include <iomanip.h>

#include "round3dd.hh"

Round3DD::Round3DD()
{
    des.setUseIP(false);
    des.setUseRLSwap(false);
    des.setRounds(3);

    P.setPerm(Permutation::ID::P);
    P_inv.setPerm(Permutation::ID::P_inv);
    E.setPerm(Permutation::ID::E);

    S[1-1].setTable(SBox::S1);
    S[2-1].setTable(SBox::S2);
    S[3-1].setTable(SBox::S3);
    S[4-1].setTable(SBox::S4);
    S[5-1].setTable(SBox::S5);
    S[6-1].setTable(SBox::S6);
    S[7-1].setTable(SBox::S7);
    S[8-1].setTable(SBox::S8);

    numpairs=3;
}

Round3DD::~Round3DD()
{
}

/* set up the key we are going to look for */
void Round3DD::setKey(uchar_ptr k)
{
    memcpy(in_key,k,8);
```

```
    K.setParity(in_key);
    K.buildK(in_key);
    des.setKey(in_key);
}

void Round3DD::setNumPairs(int n)
{
    numpairs=n;
}

void Round3DD::getSuggestedKeys(uchar en,uchar es,uchar cp, int sbox,
                      short *skeys,int &numskeys)
{
#if 0
    cout<<"------------ getSuggestedKeys -------------"<<endl;
    cout<<"en = "<<tobit(en,8)<<endl;
    cout<<"es = "<<tobit(es,8)<<endl;
    cout<<"cp = "<<tobit(cp,8)<<endl;
    cout<<"sbox = "<<sbox<<endl;
    cout<<endl;
#endif

    numskeys=0;
    const uchar ep=en^es;//XOR(&en,&es,&ep,1);
    uchar t1;
    uchar t2,c1,c2,tc,tk;

    for(t1=0;t1<64;t1++) // 64 - range of 6 bits
    {
        t2=c1=c2=tc=tk=0;
        c1=S[sbox](t1); // S-box output for B_j
        t2=t1^ep;//XOR(&t1,&ep,&t2,1);  // find B_j^*
        c2=S[sbox](t2); // S-box output for B_j^*
        tc=c1^c2;//XOR(&c1,&c2,&tc,1); // S-box output difference
#if 0
        cout<<"============================"<<endl;
        cout<<"en = "<<tobit(en,8)<<endl;
        cout<<"t1 = "<<tobit(t1,8)<<endl;
        cout<<"c1 = "<<tobit(c1,8)<<endl;
        cout<<"c2 = "<<tobit(c2,8)<<endl;
        cout<<"tc = "<<tobit(tc,8)<<endl;
#endif
        // if the output difference is the one we're looking for then
        // calculate the suggested key bits and store them
        if(tc==cp)
        {
            tk=t1^en;//XOR(&t1,&en,&tk,1);
```

```
#if 0
            cout<<"++++++++++++++++++++++++++++"<<endl;
            cout<<"tk = "<<tobit(tk,8)<<endl;
            cout<<"============================"<<endl;
#endif
            skeys[numskeys]=tk;
            numskeys++;
        }
    }
    cout<<flush;
}


/*
 * here we go through each of the plaintxt/ciphertxt pairs and
 * caculate the input and output differences and then tabulate
 * the suggested keys, storing the results in counters
 */
void Round3DD::tabulate(void)
{
    for(int s=0;s<8;s++)
        for(int i=0;i<64;i++)
            counters[s][i]=0;

#if 1
    // n = normal, s = star, p = prime
    uchar Cp[numpairs][8];      // output XOR for each S-box
    uchar En[numpairs][8];      // an input for each S-box
    uchar Es[numpairs][8];      // another input for each S-box
#if 1
    for(int u=0;u<numpairs;u++)
        for(int v=0;v<8;v++)
        {
            Cp[u][v]=0;
            En[u][v]=0;
            Es[u][v]=0;
        }
#endif
    cout<<"---- Calculate the Input/Output Differences (E,E*,C')
----"<<endl;
    //For each pair calculate the input XOR, Ep, and the
    // output XOR, Cp, of the last round.
    for(int c=0;c<numpairs;c++)
    {
        uchar en[6]; // single expansion for input
        uchar es[6]; // other single expansion for input
        uchar cp[4]; // single output xor
```

```
        uchar l0n[4];
        uchar l0s[4];
        uchar l3n[4];
        uchar l3s[4];

        uchar r3n[4];
        uchar r3s[4];

        uchar r3p[4];
        uchar l0p[4];
        uchar l0pxr3p[4];

#if 1
        memset(en,0,6);
        memset(es,0,6);
        memset(cp,0,4);

        memset(l0n,0,4);
        memset(l0s,0,4);
        memset(l3n,0,4);
        memset(l3s,0,4);

        memset(r3n,0,4);
        memset(r3s,0,4);

        memset(r3p,0,4);
        memset(l0p,0,4);
        memset(l0pxr3p,0,4);
#endif

        memcpy(l0n,ppairs[c][0],4);
        memcpy(l0s,ppairs[c][1],4);
        memcpy(l3n,cpairs[c][0],4);
        memcpy(l3s,cpairs[c][1],4);
        memcpy(r3n,cpairs[c][0]+4,4);
        memcpy(r3s,cpairs[c][1]+4,4);

        cout<<"----------- Pair "<<c+1<<" -----------"<<endl;
        cout<<"L0  \t\t= "<<tobin(l0n,4)<<endl;
        cout<<"L0* \t\t= "<<tobin(l0s,4)<<endl;
        cout<<"L3  \t\t= "<<tobin(l3n,4)<<endl;
        cout<<"L3* \t\t= "<<tobin(l3s,4)<<endl;
        cout<<"R3  \t\t= "<<tobin(r3n,4)<<endl;
        cout<<"R3* \t\t= "<<tobin(r3s,4)<<endl;

        //calculate the output XOR
```

```
XOR(r3n,r3s,r3p,4);
XOR(l0n,l0s,l0p,4);
XOR(l0p,r3p,l0pxr3p,4);

cout<<"R3' \t\t= "<<tobin(r3p,4)<<endl;
cout<<"L0' \t\t= "<<tobin(l0p,4)<<endl;
cout<<"R3' xor L0' \t= "<<tobin(l0pxr3p,4)<<endl;

// Cp is calculated from the XOR of the left hand side of the
//  plaintext and the XOR of the right hand side of the ciphertext
//  using the inverse of the permutation P
P_inv(l0pxr3p,cp);
cout<<"C'["<<c<<"] = "<<tobin(cp,4)<<endl;

// The input and the input XOR are calculated from the expansion
// of the left hand side of the ciphtertext
//calculate input expansion
E(l3n,en);
cout<<"E ["<<c<<"] = "<<tobin(en,6)<<endl;
E(l3s,es);
cout<<"E*["<<c<<"] = "<<tobin(es,6)<<endl;
cout<<endl;

// now take the intermediate values and expand inputs into
// 6 bit bundles, and outputs into 4 bit bundles
// so that there are 8 bundles each corresponding to the
// contraction present in the S-boxes
En[c][0]=0x3F & en[0];
En[c][1]=(0x03 & (en[0]>>6)) | (0x3C & (en[1]<<2));
En[c][2]=(0x0F & (en[1]>>4)) | (0x30 & (en[2]<<4));
En[c][3]=0x3F & (en[2]>>2);
En[c][4]=0x3F & en[3];
En[c][5]=(0x03 & (en[3]>>6)) | (0x3C & (en[4]<<2));
En[c][6]=(0x0F & (en[4]>>4)) | (0x30 & (en[5]<<4));
En[c][7]=0x3F & (en[5]>>2);

Es[c][0]=0x3F & es[0];
Es[c][1]=(0x03 & (es[0]>>6)) | (0x3C & (es[1]<<2));
Es[c][2]=(0x0F & (es[1]>>4)) | (0x30 & (es[2]<<4));
Es[c][3]=0x3F & (es[2]>>2);
Es[c][4]=0x3F & es[3];
Es[c][5]=(0x03 & (es[3]>>6)) | (0x3C & (es[4]<<2));
Es[c][6]=(0x0F & (es[4]>>4)) | (0x30 & (es[5]<<4));
Es[c][7]=0x3F & (es[5]>>2);

Cp[c][0]=0x0F &  cp[0];
Cp[c][1]=0x0F & (cp[0]>>4);
```

```
        Cp[c][2]=0x0F &  cp[1];
        Cp[c][3]=0x0F & (cp[1]>>4);
        Cp[c][4]=0x0F &  cp[2];
        Cp[c][5]=0x0F & (cp[2]>>4);
        Cp[c][6]=0x0F &  cp[3];
        Cp[c][7]=0x0F & (cp[3]>>4);
    }

    //now for each of the eight S-boxes calculate the list
    // of possible inputs that correspond to the Ep-Cp combination
    // and then use the actual input to find the possible
    // corresponding key bits and increment the counter

    //we have the following data
    //uchar Cp[3][8];   // output XOR for each S-box
    //uchar En[3][8];   // an input for each S-box
    //uchar Es[3][8];   // another input for each S-box

    cout<<"---- Find suggested subkey bits T_j(E,E*,C') ----"<<endl;
    for(int c=0;c<numpairs;c++)
    {
        cout<<"----------- Pair "<<c+1<<" -----------"<<endl;
        short skeys[64];  //list of suggested keys filled out by
                          //getSuggestedKeys. 64 is trivial max, but
                          //normally numskeys would be 8 or 16
        int numskeys;
        for(int s=0;s<8;s++)
        {
            numskeys=0;
#if 1
            for(int u=0;u<64;u++)
                skeys[u]=0;
#endif
            getSuggestedKeys(En[c][s],Es[c][s],Cp[c][s],s,
                             skeys,numskeys);

            cout<<"[ "<<tobit(En[c][s],6);
            cout<<", "<<tobit(Es[c][s],6);
            cout<<", "<<tobit(Cp[c][s],4);
            cout<<", "<<s+1<<" ] ";
            cout<<"{";
            for(int n=0;n<numskeys;n++)
            {
                if(n && n%4==0)
                    cout<<endl<<"                                ";

cout<<setw(2)<<skeys[n]<<":"<<tobit(skeys[n],6)<<(n==numskeys-1?"":",");
```

```
                counters[s][skeys[n]]++;
            }
            cout<<"} "<<endl;
        }
        cout<<endl;
    }
#endif

#if 0
    /* test counters for key 1a624c89520dec46 */

    counters[0][61]=3;  // 47
    counters[1][40]=3;  // 5
    counters[2][50]=3;  // 19
    counters[3][0]=3;   // 0
    counters[4][6]=3;   // 24
    counters[5][56]=3;  // 7
    counters[6][56]=3;  // 7
    counters[7][35]=3;  // 49
#endif
}
/*
 * here we use the couters array to reconstruct 48 bits of
 * the subkey and expands this to set the 64 bit subkey and
 * 64 bit mask
 */
void Round3DD::collect(void)
{
    memset(subkey48,0,6);
    memset(subkey,0,8);
    memset(mask,0,8);

    /* for each counter array find the entry with a value
     * of three. Then based on its position set 6 bits of
     * the 48 bit subkey.
     */
    int i;
    for(int c=0;c<8;c++)
    {
        for(i=0;i<64;i++)
        {
            if(counters[c][i]==numpairs)
            {
                cout<<i<<(c==7?"":",");
                break;
            }
        }
```

```
        if(i==64)
            throw R3DDException(
                    string("No unique key bits suggested for sbox: ")+
                    tostring(c)
                    );
        /* now take six bits from i and append to subkey48 */
        /*
         *  (......)(......)(......)(......)(......)(......)(......)
         *  [....][....][....][....][....][....][....][....][....][....]
         *     0     1     2     3
         *
         */
        int byte=(c*6)/8;
        switch(c%4)
        {
            case 0:
                subkey48[byte]|=(0x3f&i);
                break;
            case 1:
                subkey48[byte]|=(0x3f&i)<<6;
                subkey48[byte+1]|=(0x3f&i)>>2;
                break;
            case 2:
                subkey48[byte]|=(0x3f&i)<<4;
                subkey48[byte+1]|=(0x3f&i)>>4;
                break;
            case 3:
                subkey48[byte]|=(0x3f&i)<<2;
                break;
        }
    }

    cout<<endl<<flush;

    /* expand the subkey and set the mask */
    K.setMask(mask,3);
    K.expandSubkey(subkey48,subkey,3);

}

/* call brute search to find the remaining bits */
void Round3DD::bsearch(void)
{
    bsd.setMask(mask);
    bsd.setSubkey(subkey);
    bsd.setPlaintxt(ppairs[0][0]);
    bsd.setCiphertxt(cpairs[0][0]);
```

```
    try {
        bsd.run(out_key);
    } catch (BruteSearchDes::BSDException e) {
        cerr<<"Caught BSDException: <"<<e.getMessage()<<">"<<endl<<flush;
        throw R3DDException("No key found");
    }
}

/* use differential cryptanalysis on 3 round des to recover the key */
void Round3DD::recover(uchar_ptr k)
{
    /* create the plaintxt/ciphertxt pairs */
    createPairs();

    /* now we recover the key */
    try {
    cout<<"------------ tabulating ------------"<<endl;
    tabulate();
    cout<<"------------ collecting ------------"<<endl;
    collect();
    cout<<"---------- collected bits ----------"<<endl;
    cout<<"realsub  = "<<tobin(K[3],6)<<endl;
    cout<<"subkey48 = "<<tobin(subkey48,6)<<endl;
    cout<<"mask   = "<<tobin(mask,8)<<endl;
    cout<<"subkey = "<<tobin(subkey,8)<<endl;
    cout<<"realkey= "<<tobin(in_key,8)<<endl;
    cout<<"----------- brute search -----------"<<endl;
    bsearch();
    cout<<"------------- key found ------------"<<endl;
    } catch (...) {
        cout<<endl<<"NOT FOUND"<<endl;
        throw;
    }

    /* set the output key */
    memcpy(k,out_key,8);
}

/* Here we create 3 different plaintxt/ciphertxt pairs that
 * satisfiy our characteristic.
 * The characteristic that we are using requires the upper
 * half of the xor to be zero
 * (i.e. the plaintxts must be equal in the upper four bytes)
 * The ciphertxts can be any value.
 *
 * To create these values we fill the plaintxts using /dev/urandom
```

```
 */
void Round3DD::createPairs(void)
{
    int urnd;
#if 0
    // for key 1a624c89520dec46
    hexstrtobuf("748502cd38451097",ppairs[0][0]);
    hexstrtobuf("3874756438451097",ppairs[0][1]);

    hexstrtobuf("486911026acdff31",ppairs[1][0]);
    hexstrtobuf("375bd31f6acdff31",ppairs[1][1]);

    hexstrtobuf("357418da013fec86",ppairs[2][0]);
    hexstrtobuf("12549847013fec86",ppairs[2][1]);
#else
    urnd=open("/dev/urandom",O_RDONLY);

    for(int p=0;p<numpairs;p++)
    {
        read(urnd,ppairs[p][0]+4,4);
        memcpy(ppairs[p][1]+4,ppairs[p][0]+4,4);
        read(urnd,ppairs[p][0],4);
        read(urnd,ppairs[p][1],4);
    }
#endif


    for(int p=0;p<numpairs;p++)
        for(int i=0;i<2;i++)
            des.encrypt(ppairs[p][i],cpairs[p][i]);

    cout<<"------------ Test Pairs ------------"<<endl;
    cout<<"---plaintext---- -- ---ciphertext---"<<endl;
    for(int p=0;p<numpairs;p++)
    {
        for(int i=0;i<2;i++)
        {
            cout<<tohex(ppairs[p][i],8)<<" -- ";
            cout<<tohex(cpairs[p][i],8)<<endl;
        }
        cout<<"---------------- -- ----------------"<<endl;
    }
    cout<<flush;
}
```

SOURCE: **diffcrypt/3round/round3dd.hh**

```
/* @(#) $Id: round3dd.hh,v 1.1 2001/11/16 11:37:16 stewart Exp $ */
#ifndef _ROUND3DD_HH
#define _ROUND3DD_HH

#include "des.hh"
#include "brute_des.hh"

#define MAXPAIRS 20

class Round3DD
{
    private:
        DES des;
        Keys K;
        BruteSearchDes bsd;

        uchar in_key[8];
        uchar out_key[8];

        uchar subkey48[6];
        uchar subkey[8];
        uchar mask[8];

        int numpairs;
        uchar ppairs[MAXPAIRS][2][8]; // plaintxt pairs (3 pairs of 64
bits)
        uchar cpairs[MAXPAIRS][2][8]; // ciphertxt pairs

        short counters[8][64]; // counter arrays for tabulating suggested
                               // key bits for each of the 8 sboxes

        Permutation P;
        Permutation P_inv;
        Permutation E;

        SBox S[8];

    public:
        NEWEXCEPTION(R3DDException);

        Round3DD();
        ~Round3DD();

        /* inards of 3-round diffcrypt */
        void createPairs(void);
        /* */
```

```
        void setNumPairs(int n);

        void setKey(uchar_ptr k);
        void recover(uchar_ptr k);

        void getSuggestedKeys(uchar en,uchar es,uchar cp, int sbox,
                              short *skeys,int &numskeys);

        void tabulate(void);
        void collect(void);
        void bsearch(void);
};

#endif /* _ROUND3DD_HH */
```

SOURCE: **diffcrypt/brutesearch/Makefile**

```
# Makefile for des implementation

TOP=../../
SRC+=brute_des.cc
include $(TOP)/config.mk
LIBS+=-lsg_des

all: libsg_brute_des.so

libs: libsg_brute_des.so

runBruteDES: runBruteDES.cc $(SRC)
        $(CXX) $(CXXFLAGS) -pg $(INCLUDES) $(LIBDIR) -o runBruteDES
runBruteDES.cc $(SRC) $(LIBS)

libsg_brute_des.so: $(OBJS)
        $(CXX) $(CXXFLAGS) $(LIBDIR) -shared -o libsg_brute_des.so $(OBJS)
$(LIBS)

clean:
        rm -f *.o
        rm -f *~
        rm -f .depend
        rm -f libsg_brute_des.so
        rm -f bruteDES
```

SOURCE: **diffcrypt/brutesearch/bruteDES.cc**

```
static char cvsid[] = { "@(#) $Id: brute_des.cc,v 1.1 2001/11/16 11:37:18
stewart Exp $" };
#include "common.h"
USE(cvsid);

/*
 * Inputs
 *  - a 64 bit mask where ones indicate the bits we
 *    already know paritybits are ignored
 *  - a 64 bit key with the subkey bits set appropiratly
 *    (the bits that fall outside of the mask are ignored)
 *  - 64 bit test input block
 *  - 64 bit test cipher block
 *
 * Method
 *  - using the mask construct test keys by running through
 *    all variations on the unknown key bits
 *  - for each key calculate the ciphertext of the test input
 *    and compare it to the test output
 *  - terminate as soon as a key is found which correctly
 *    encrypts the test input block
 */

#include "brute_des.hh"

BruteSearchDes::BruteSearchDes()
{
    display=false;
    des.setDisplay(false);
    des.setUseIP(false);
    des.setUseRLSwap(false);
    des.setRounds(3);
    shiftindex=NULL;
}

BruteSearchDes::~BruteSearchDes()
{
    if(shiftindex)
        delete shiftindex;
}

void BruteSearchDes::setDisplay(bool b)
{
    display=b;
}
```

```
//returns the number of unknown bits
void BruteSearchDes::setMask(uchar_ptr mask)
{
    //copy the mask
    bcopy(mask,keymask,8);

    //set the parity bits to one
    for(int i=0;i<8;i++)
        keymask[i]|=0x80;

    //count the number of zeros
    int zcnt=0;

    for(int i=0;i<8;i++)
    {
        for(int j=0;j<8;j++)
        {
            if(((keymask[i]>>j)&0x01)==0)
                zcnt++;
        }
    }
    searchlen=zcnt;
    if(searchlen>32)
        throw BSDException("Search to big!");

    buildShiftArray();
}

ulong BruteSearchDes::getSearchCnt(void)
{
    return searchcnt;
}

int BruteSearchDes::getSearchLen(void)
{
    return searchlen;
}

void BruteSearchDes::setSubkey(uchar_ptr skey)
{
    //copy the subkey
    bcopy(skey,subkey,8);
}

void BruteSearchDes::setPlaintxt(uchar_ptr ptxt)
{
```

```
    bcopy(ptxt,plaintxt,8);
}

void BruteSearchDes::setCiphertxt(uchar_ptr ctxt)
{
    bcopy(ctxt,ciphertxt,8);
}

void BruteSearchDes::buildShiftArray(void)
{
    //records the position of each zero so that the counter can
    //be spread over the key

    int c=0;
    shiftindex=new int[searchlen];
    for(int i=0;i<8;i++)
        for(int j=0;j<8;j++)
            if(((keymask[i]>>j)&0x01)==0)
                shiftindex[c++]=i*8+j;
}

int* BruteSearchDes::getShiftArray(void)
{
    return shiftindex;
}

void BruteSearchDes::buildTestKey(ulong cnt)
{
    //zero key
    bzero(testkey,8);

    //set test bits
    for(int c=0;c<searchlen;c++)
    {
        int byte,bit;
        byte=shiftindex[c]/8;
        bit=shiftindex[c]%8;

        if((cnt>>c)&0x01)
            testkey[byte]|=1<<bit;
    }

    //or in subkey bits
    for(int i=0;i<8;i++)
    {
        testkey[i]|=subkey[i]&keymask[i];
    }
```

```
}

inline void BruteSearchDes::doEncryption()
{
    Keys::setParity(testkey);
    des.setKey(testkey);
    des.encrypt(plaintxt,testctxt);
}

/* places the found key into outkey
 * otherwise an exception is thrown
 */
void BruteSearchDes::run(uchar_ptr outkey)
{
    //build the shift index array
    ulong cnt;
    ulong maxcnt=1<<searchlen;

    for(cnt=0;cnt<maxcnt;cnt++)
    {
        //create test key
        buildTestKey(cnt);

        //encrypt plaintxt
        doEncryption();

#if 0
        cout<<"Trying key: "<<tohex(testkey,8);
        cout<<" ctxt="<<tohex(testctxt,8);
        cout<<" ttxt="<<tohex(ciphertxt,8)<<endl<<flush;
#endif
#if 1
        if(cnt%100000==0)
            cout<<"."<<flush;
#endif

        //test result
        if(blockcmp(ciphertxt,testctxt,8)==0)
        {
            //found key
            bcopy(testkey,outkey,8);
            searchcnt=cnt;
#if 1
            cout<<endl<<flush;
#endif
            return;
        }
```

```
    }

    throw BSDException("No key found");
}
```

SOURCE: **diffcrypt/brutesearch/bruteDES.cc**

```
static char cvsid[] = { "@(#) $Id: brute_des.cc,v 1.1 2001/11/16 11:37:18
stewart Exp $" };
#include "common.h"
USE(cvsid);

/*
 * Inputs
 *  - a 64 bit mask where ones indicate the bits we
 *    already know paritybits are ignored
 *  - a 64 bit key with the subkey bits set appropiratly
 *    (the bits that fall outside of the mask are ignored)
 *  - 64 bit test input block
 *  - 64 bit test cipher block
 *
 * Method
 *  - using the mask construct test keys by running through
 *    all variations on the unknown key bits
 *  - for each key calculate the ciphertext of the test input
 *    and compare it to the test output
 *  - terminate as soon as a key is found which correctly
 *    encrypts the test input block
 */

#include "brute_des.hh"

BruteSearchDes::BruteSearchDes()
{
    display=false;
    des.setDisplay(false);
    des.setUseIP(false);
    des.setUseRLSwap(false);
    des.setRounds(3);
    shiftindex=NULL;
}

BruteSearchDes::~BruteSearchDes()
{
    if(shiftindex)
        delete shiftindex;
```

```
}

void BruteSearchDes::setDisplay(bool b)
{
    display=b;
}

//returns the number of unknown bits
void BruteSearchDes::setMask(uchar_ptr mask)
{
    //copy the mask
    bcopy(mask,keymask,8);

    //set the parity bits to one
    for(int i=0;i<8;i++)
        keymask[i]|=0x80;

    //count the number of zeros
    int zcnt=0;

    for(int i=0;i<8;i++)
    {
        for(int j=0;j<8;j++)
        {
            if(((keymask[i]>>j)&0x01)==0)
                zcnt++;
        }
    }
    searchlen=zcnt;
    if(searchlen>32)
        throw BSDException("Search to big!");

    buildShiftArray();
}

ulong BruteSearchDes::getSearchCnt(void)
{
    return searchcnt;
}

int BruteSearchDes::getSearchLen(void)
{
    return searchlen;
}

void BruteSearchDes::setSubkey(uchar_ptr skey)
{
```

```
    //copy the subkey
    bcopy(skey,subkey,8);
}

void BruteSearchDes::setPlaintxt(uchar_ptr ptxt)
{
    bcopy(ptxt,plaintxt,8);
}

void BruteSearchDes::setCiphertxt(uchar_ptr ctxt)
{
    bcopy(ctxt,ciphertxt,8);
}

void BruteSearchDes::buildShiftArray(void)
{
    //records the position of each zero so that the counter can
    //be spread over the key

    int c=0;
    shiftindex=new int[searchlen];
    for(int i=0;i<8;i++)
        for(int j=0;j<8;j++)
            if(((keymask[i]>>j)&0x01)==0)
                shiftindex[c++]=i*8+j;
}

int* BruteSearchDes::getShiftArray(void)
{
    return shiftindex;
}

void BruteSearchDes::buildTestKey(ulong cnt)
{
    //zero key
    bzero(testkey,8);

    //set test bits
    for(int c=0;c<searchlen;c++)
    {
        int byte,bit;
        byte=shiftindex[c]/8;
        bit=shiftindex[c]%8;

        if((cnt>>c)&0x01)
            testkey[byte]|=1<<bit;
    }
```

```
    //or in subkey bits
    for(int i=0;i<8;i++)
    {
        testkey[i]|=subkey[i]&keymask[i];
    }
}

inline void BruteSearchDes::doEncryption()
{
    Keys::setParity(testkey);
    des.setKey(testkey);
    des.encrypt(plaintxt,testctxt);
}

/* places the found key into outkey
 * otherwise an exception is thrown
 */
void BruteSearchDes::run(uchar_ptr outkey)
{
    //build the shift index array
    ulong cnt;
    ulong maxcnt=1<<searchlen;

    for(cnt=0;cnt<maxcnt;cnt++)
    {
        //create test key
        buildTestKey(cnt);

        //encrypt plaintxt
        doEncryption();

#if 0
        cout<<"Trying key: "<<tohex(testkey,8);
        cout<<" ctxt="<<tohex(testctxt,8);
        cout<<" ttxt="<<tohex(ciphertxt,8)<<endl<<flush;
#endif
#if 1
        if(cnt%100000==0)
            cout<<"."<<flush;
#endif

        //test result
        if(blockcmp(ciphertxt,testctxt,8)==0)
        {
            //found key
            bcopy(testkey,outkey,8);
```

```
            searchcnt=cnt;
#if 1
            cout<<endl<<flush;
#endif
            return;
        }
    }

    throw BSDException("No key found");
}
```

SOURCE: **diffcrypt/brutesearch/runBruteDES.cc**

```
static char cvsid[] = { "@(#) $Id: runBruteDES.cc,v 1.1 2001/11/16
11:37:18 stewart Exp $" };
#include "common.h"
USE(cvsid);

#include "brute_des.hh"
#include "keys.hh"

/*
 * This provides a command line interface to the brute force
 * DES key search
 */

int main(int argc, char* argv[])
{
    BruteSearchDes bsd;
    Keys K;

    uchar ptxt[8];
    uchar ctxt[8];
    uchar subkey48[6];
    uchar subkey64[8];
    uchar subkey[8];
    uchar maskII[8];
    uchar mask[8];
    uchar outkey[8];

    //see page 97 stinson
    hexstrtobuf("748502cd38451097",ptxt);
    hexstrtobuf("03c70306d8a09f10",ctxt);

#if 1
    binstrtobuf("11111111 11111111 11011011 10111111 "
```

```
                "11111111 11110011 11101101 01111111",mask);
#endif
#if 0
    binstrtobuf("10000001 10000001 10001011 10111111 "
                "10000001 10110011 11101101 01111111",mask);
#endif
    binstrtobuf("00011010 01100010 01001000 10001000 "
                "01010010 00000000 11101100 01000110",subkey);

    binstrtobuf("101111 000101 010011 000000 "
                "011000 000111 000111 110001",subkey48);

    K.setMask(maskII,3);
    K.expandSubkey(subkey48,subkey64,3);
    cout<<"Mask:    "<<tobin(mask,8)<<endl;
    cout<<"Mask II: "<<tobin(maskII,8)<<endl;
    cout<<"Subkey 48: "<<tobin(subkey48,6)<<endl;
    cout<<"Subkey   : "<<tobin(subkey,8)<<endl;
    cout<<"Subkey 64: "<<tobin(subkey64,8)<<endl;


    bsd.setMask(mask);
    bsd.setSubkey(subkey);
    bsd.setPlaintxt(ptxt);
    bsd.setCiphertxt(ctxt);

    try {
        bsd.run(outkey);
    } catch (BruteSearchDes::BSDException e) {
        cout<<"Caught BSDException: <"<<e.getMessage()<<">"<<endl<<flush;
        return 1;
    }

    cout<<"Found key: "<<tohex(outkey,8);
    cout<<", tested "<<bsd.getSearchCnt()<<" keys"<<endl<<flush;



    return 0;
}
```