Product cipher negotiation with on-line evaluation for private communication over computer networks

VASILIOS KATOS

Doctor of Philosophy

ASTON UNIVERSITY

February 1999

ASTON UNIVERSITY

Product cipher negotiation with on-line evaluation for private communication over computer networks

VASILIOS KATOS

Doctor of Philosophy, 1999

Thesis Summary

A method is proposed to offer privacy in computer communications, using symmetric product block ciphers. The security protocol involves a cipher negotiation stage, in which two communicating parties select privately a cipher from a public cipher space. The cipher negotiation process includes an on-line cipher evaluation stage, in which the cryptographic strength of the proposed cipher is estimated.

The cryptographic strength of the ciphers is measured by confusion and diffusion. A method is proposed to describe quantitatively these two properties. For the calculation of confusion and diffusion a number of parameters are defined, such as the confusion and diffusion matrices and the marginal diffusion. These parameters involve computationally intensive calculations that are performed off-line, before any communication takes place. Once they are calculated, they are used to obtain estimation equations, which are used for on-line, fast evaluation of the confusion and diffusion of the negotiated cipher. A technique proposed in this thesis describes how to calculate the parameters and how to use the results for fast estimation of confusion and diffusion for any cipher instance within the defined cipher space.

Keywords: confusion, marginal diffusion, diffusion/confusion matrix, cryptographic algorithm negotiation protocols, cryptographic strength

to my parents, Tasos and Eleni,

to my sister Anastasia

Acknowledgements

I wish to thank my supervisor, Bernard Doherty, for his continued support and guidance throughout the research and writing of the thesis.

I also wish to thank Tony Beaumont, for his excellent technical support.

Finally, I wish to thank the staff of the Department for making my stay enjoyable and interesting.

Contents

List of Tables

List of Figures

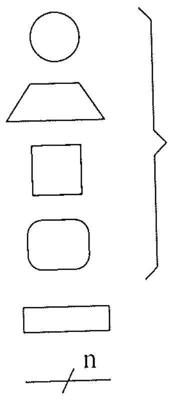
Glossary of symbols and abbreviations

BFN	Balanced Feistel Network
C	Ciphertext
$\mathcal{C}$	Confusion matrix
CBC	Cipher Block Chaining mode of operation
CPB	Cryptographic Block Profile
CFB	Cipher Feedback mode of operation
$\mathcal{D}$	Diffusion matrix
$\mathrm{D_K(X)}$	Decryption transformation of X with key K
DES	Data Encryption Standard
$\Delta\mathrm{X}$	Bitwise exclusive-**OR** difference of two values of variable X
$\mathrm{E_K(X)}$	Encryption transformation of X with key K
$\mathrm{E} : \mathcal{V}^* \rightarrow \mathcal{W}^*$	Encryption mapping transformation of plaintext space to ciphertext space
ECB	Electronic Codebook mode of operation
$\mathrm{GF(2)^n}$	Binary words of length n (bits)
IDEA	International Data Encryption Algorithm
IV	Initialisation Vector
LAN	Local Area Network

LFSR	Linear Feedback Shift Register
$M = \{\chi_1 \ldots x_\theta\}$	Encryption system with encryption steps $\chi_j, j = 1..\theta$
MAC	Message Authentication Code
MDC	Manipulation Detection Code
$\mathrm{M_D}$	Marginal diffusion
O()	Big-O notation, measure of complexity
OFB	Output Feedback mode of operation
P	Plaintext
PKC	Public Key Cryptosystem
$\mathbf{R^2}$	Determination coefficient
$\mathbf{\bar{R}^2}$	Determination coefficient corrected for the degrees of freedom
RSA	Rivest Shamir Adelman public key cryptosystem
S-Box	Substitution Box
S/MIME	Secure/Multipurpose Internet Mail Extensions
SPN	Substitution Permutation Network
SSL	Secure Sockets Layer protocol
UFN	Unbalanced Feistel Network
$\mathcal{V}$	The set of plaintext symbols
$\mathcal{V}^*$	Plaintext space
$\mathcal{V}^n$	Finite set of plaintexts length n
$\mathcal{V}^{(n)}$	Finite set of plaintexts length up to n
$\mathcal{W}$	The set of ciphertext symbols
$\mathcal{W}^*$	Ciphertext space
$\mathcal{V}^m$	Finite set of ciphertexts length m
$\mathcal{V}^{(m)}$	Finite set of ciphertexts length up to m
Ψ **matrix**	The matrices used for computation of the diffusion and confusion matrices
XOR, $\oplus$	Exclusive-**OR** boolean operation

$\forall i$	For all i
$\alpha \in B$	α is a member of set B
$\boxplus$	Integer addition modulo 2^n
$\odot$	Integer multiplication modulo $2^n + 1$
$<<< m$	Left circular shift of m bits
$>>> m$	Right circular shift of m bits
$s_1 \bullet s_2$	Concatenation of strings s_1 and s_2

Graphic symbols



Functions:
different symbols allow separation
of function types in diagrams

data block

n-bit data bus

Part I

Introduction

Chapter 1

Introduction

Civilization is the progress toward a society of privacy. The savage's whole existence is public, ruled by the laws of his tribe. Civilization is the process of setting man free from men.

Ayn Rand - The Fountainhead (1943)

With the fast growth of computer networks and their introduction to many environments ranging from domestic to organisational, the need for privacy and more generally security has become apparent, and cryptography, which plays a primary role in communications security, has become a science rather an art.

The Internet user community seeks a high standard of security, and the underlying research community, the Internet Engineering Task Force, is striving to develop and standardise a number of security protocols to encourage many types of electronic transactions, ranging from a routine email to on-line purchases.

1.1 Motivation

The motivation for this work can be explained with the following example. Consider the scenario where two companies which have never communicated in the past wish to start doing business via the Internet. The need for security for their transactions is assumed. Secure transactions require agreement of security protocols, agreement on cryptographic algorithm(s) to be used and possibly a password. In some protocols such as the RSA (Rivest *et al.*, 1978) and Diffie Hellman (1976) key agreeement, the cipher is integral to the protocol. These cryptographic algorithm dependent protocols will not be considered in this work, since this thesis focues on cipher independent protocols which can support a number of ciphers.

In the case of a protocol which does not specify the algorithm, the problem of selecting a suitable cipher remains. The term 'suitable' may be application specific and may include properties such as cryptographic strength, speed and key length.

A common approach is to choose a standard[1], published cipher such as the DES or IDEA. In some protocols the use of more than one standard cipher is allowed (eg. Netscape's SSL protocol, and S/MIME, the cryptographic services for MIME). Such protocols include a negotiation stage in which the cipher for each communication session is nominated from the set list of available ciphers.

An alternative approach is to use a fully private cipher. This approach contains serious flaws which may render the communication insecure. More specifically, it has been well established that the security of a private cipher is unknown, due to the fact that it has not been extensively analysed and tested, and even if it resists all known cryptanalytic attacks, it could be attacked by reverse engineering methods.

A standard, published cipher on the other hand, will have been widely analysed and its estimated security may be close to its actual security. However, standard ciphers have 'expiration dates' which relate to their key length and the feasibility of an exhaustive search.

Instead of increasing the key length to improve security, the approach of selecting one standard cipher from a set has been employed, as mentioned earlier. This approach serves to increment the search space, assuming that the selection of the standard cipher is kept secret. Using the cipher selection approach does not however result in a substantial increase in search space (The SSL for instance uses four symmetric ciphers

[1]In the contect of this thesis, a *standard cipher* is a cipher which is well known and widely accepted.

resulting in an approximate increase of four times the search space, if the negotiation is private).

This thesis proposes an alternative approach to increasing the search space. The approach is to define a large set of ciphers and to use one cipher from the set for each communication session. The large set of ciphers is generated by composition of a number of standard ciphers. The set of all ciphers which form the cipher space is public information. The selection of a specific cipher from the cipher space becomes part of the private information shared between the two communicating parties.

A cipher is described by its structure, and during session setup negotiation a proposed structure is passed as secret information and is evaluated by the parties to determine its acceptability.

More analytically, the constructed (composed) ciphers result from the sequential combination (cryptographic composition) of the public ciphers, combined further with feedback blocks.

This thesis investigates the hypothesis that the composed algorithm has in most cases greater strength than its constituent standard ciphers alone. The constructed ciphers can be of differing complexity, and have different cryptographic strength. If the cipher is to be acceptable to both parties it is necessary to provide a way to measure strength and thus distinguish weak from strong cipher instances.

An evaluation method is developed in this thesis, proposing measures that allow strength of the composed cipher to be related to strength of the individual standard ciphers.

In cipher negotiation there is an additional restriction: the evaluation of the cryptographic strength of the negotiated cipher has to be completed within acceptable time limits for effective on-line communications.

The approach outlined for evaluating the strength of a cipher is a direct consequence of the large cipher space. It is demonstrated that the cardinality of the cipher space is exponentially related to the cryptographic transformations used. Hence, it is infeasible to evaluate every instance, but a method to predict the cryptographic strength for every instance is required.

It is proposed by this thesis to store results of off-line testing of the encryption steps, and to estimate quickly the cryptographic strength of the negotiated cipher from these results. This thesis proposes suitable measures of strength and provides rules on use of these measures for "quick" evaluation of strength of a composed cipher proposed in negotiation.

1.2 Aims of thesis

The main aims of this thesis are:

1. to define a family of symmetric block ciphers constructed (composed) from a set of publicly known encryption steps,
2. to develop a framework for evaluating the cryptographic strength of the cipher instances from the family of composed ciphers, and
3. to develop a fast[2] on-line cipher evaluation method to be embedded in the cryptographic algorithm negotiation protocol and to give an indicative measure of the cipher evaluation speeds in relation to the algorithm

with the supporting aim:

4. to develop a prototype system for experiment and testing.

Subsidiary aims are:

1. (a) offer practical security against a known plaintext attack.
   (b) offer the option of trade-off between complexity over speed.

[2]The term *fast* should be interpreted as *acceptable delay in setting up a session*

   (c) use and allow standardisation.

   (d) accommodate evolution of cryptographic technology.

2. (a) analyse confusion and diffusion by investigating contributing parameters and describing them with quantitative means and relate the strength of the cipher to these values.

   (b) offer rules to filter out known weak instantiations.

3. (a) develop a method in which the results of long term computations are summarised, stored and utilised in the algorithm negotiation protocol, in order to speed the negotiation.

   (b) develop a message forwarding utility to test the algorithm negotiation protocol.

Some secondary aims are:

A. it should be feasible that the method for selecting and constructing any of the cipher instances of the family, is the same function call. This is a practical requirement; if there were billions of cipher instances and each cipher needed a different implementation, the method would not be practical.

B. calculate the total search space, which is the product of the key space and the cipher space.

More analytically, the definition of a family of ciphers defines also a cipher space. The cardinality of this space is equal to the number of all possible composed ciphers. The secrecy on the selection of the composed cipher increases the total search space, which becomes the product of this cipher space and the key space. This is calculated later in the thesis.

The reseach focuses only on block ciphers. Stream ciphers are not included because the proposed measures involve confusion and diffusion, and stream ciphers have inherently low confusion and diffusion and therefore the selected measures are not appropriate.

It was also required to provide a measure of cryptographic strength of the ciphers, in order to distinguish the cryptographically weak compositions from the strong compositions. The measures of cryptographic strength used are confusion and diffusion. These well known measures give good indication of the ability of the cipher to withstand a known plaintext attack. This thesis offers some theoretical results on diffusion and some experimental results and some calculation rules for confusion when applied to composed systems. Diffusion and confusion are also shown to be related.

The cipher space proposed in this thesis consists of the set of ciphers produced by cryptographic composition of publicly known encryption steps, optionally combined with feedbacks.

At this point it would be useful to draw the distinction between an 'encryption step', an 'encryption block' and a 'feedback block' - terms which are used throughout the thesis:

- An encryption step is an encryption function which could be used to construct the cipher. In this thesis, an encryption step is a publicly known 'standard' cipher.
- When the encryption step participates in the structure of a specific composed cipher instance, then it is considered to be an encryption block.
- If an encryption step is used in a feedback configuration, it would be referred as a feedback block. The feedback blocks operate on the intermediate results of the encryption steps.

Restating the work in this terminology: the proposed method specifies a set of encryption steps to define a cipher space. The cipher space consists of cryptographically composed encryption blocks combined with feedback blocks. Communication setup negotiation involves selection of one of the composed ciphers of the cipher space. The communicated information for this selection is the secure exchange of the identifiers of the standard encryption steps which are used as encryption blocks.

Since the information communicated during negotiation is the identifiers, the need for standardisation is apparent. It should be noted that the term 'standardisation' is used with the meaning of 'agreed and accepted between communicating parties'. That is, both parties should have knowledge on the encryption steps and also their identifiers. In the proposed method standardisation involves also prior agreement on the cryptographic strength and evaluation methods. This suggests that security measures of one party should provide the same results as the security measures of the other party, otherwise the proposed method could not be applied.

Security of a cryptosystem can be assessed theoretically and/or practically. Theoretical security, in which the provable security of a cryptosystem is investigated by mathematical proofs, is beneficial in the long-run and may provide conclusions which may be generalised and used for further cryptographic development. Practical security takes a more empirical view: the cryptosystem is assessed by taking into consideration the practical capabilities of the attacker, determining the attack alternatives, and assessing the security of the cryptosystem against these alternative attacks.

The security measures were selected in this thesis to offer practical security against a known plaintext attack, in which the attacker has access to all communicated ciphertext and some plaintext/ciphertext pairs.

The proposed method of constructing ciphers should also permit evolution of cryptographic technology. That is, it should be feasible for the proposed cryptosystem to incorporate advances in cryptography which may enhance the security of the cryptosystem. This can be done in the proposed cryptosystem by allowing updating or modification of the encryption steps by standardised agreement.

Turning to the second aim of the thesis, once the cipher space had been defined, there was a need to develop a framework for evaluating cryptographic strength of the composed ciphers. Since the communication scenario is assumed in which an attacker would be an eavesdropper gathering the ciphertext communicated between two parties and the attacker also knows the set of encryption steps (but not the details of the composed cipher), the strength of the cipher would be defined as its ability to conceal the information of the plaintext within its corresponding ciphertext. This ability is measured by the diffusion and confusion properties of a cipher.

Confusion and diffusion are investigated in terms of contributing parameters and described quantitatively. When a cipher consists of two or more encryption steps, the relation between the overall confusion (and diffusion) and the confusion (and diffusion) of the underlying encryption steps was investigated. A number of additional parameters were introduced in order to establish this relation for confusion with an adequate confidence level. One of these parameters is the marginal diffusion, which is introduced and defined in this thesis.

The cryptographic strength of the composed cipher is related to confusion and diffusion and, in extension, to the contributing parameter values. More specifically, after developing a way to measure the confusion and diffusion of a composed cipher, the confusion and diffusion of the composed cipher are related to the contributing parameters of the underlying encryption blocks (i.e. steps).

The proposition that the confusion and diffusion of a product cipher can be described from the parameters of the underlying encryption steps was investigated and confirmed.

In product ciphers, the (cryptographic) composition of a number of encryption steps results in increased values of confusion and diffusion (the term "avalanche" is used in the literature to address this property of product ciphers). This is proven in this thesis by a theoretical proof for diffusion and an experimental proof for confusion.

The confusion was modelled by applying linear regression techniques to the contributing parameters. Consequently, the strength of the composed cipher was able to be described using parameters from the underlying encryption steps.

The last of the aims was to develop a cipher evaluation method to be embedded in a cryptographic algorithm negotiation protocol. The main requirement for this evaluation was high speed, to allow it to be used in on-line negotiations.

As discussed above, when a composed cipher is proposed, the publicly known parameter values of each encryption step can be used to compute the cryptographic strength of the proposed cipher. These parameters are the confusion and diffusion of the encryption steps and also those additional parameters which contribute to calculation of confusion and diffusion of the composed cipher.

Once again the need to keep the encryption steps public is apparent, as the computations involving the estimation of the security parameters of the encryption steps need to be completed off-line, before any communication takes place. A "cryptographic block profile", CBP, of every encryption step was created in order to store the parameter values from the long-run tests of the encryption steps and the CBP becomes part of the public information on the cipher set.

Another requirement for the tests was invariability of the measurements, meaning that two parties independently performing the tests should obtain results that are statistically the same.

Finally, a message forwarding application between two users was developed in order to test the performance of the algorithm negotiation protocol.

In conclusion, these aims lead to a complete negotiation protocol with fast on-line cipher evaluation embedded, to allow two communicating users to set up a communication session with the level of security they agree, extending to significantly enhanced search space compared to current techniques.

1.3 Research methodology

1.3.1 Literature review

The literature review covers two main areas:

- **Cryptology.** In the area of cryptography, emphasis is given on symmetric cryptography and more precisely, in symmetric block ciphers, since the proposed method is concerned with such ciphers. The advances of cryptanalysis are also investigated, since cryptanalysis influences the design objectives of the ciphers. Furthermore, an indication of a cipher's strength is its resistance to known cryptanalytic attacks. Measures of the strength of ciphers which relate to statistical properties such as tests for randomness are also reviewed.

- **Network Security.** Security architectures in networks are investigated. This would give an insight to the trends of secure systems as well as the needs for security in internetworking environments.

1.3.2 Analysis and investigation

Analysis focuses on the properties of product ciphers - mainly Feistel transformations which are the main cryptographic transformations used in product ciphers - and multiple modes of operation. Conclusions found in the literature were used to develop new methods and then to evaluate these methods by confirming and extending published results. Investigations by experimental means contribute further to the results provided by this research.

1.3.3 Prototyping

A prototype was implemented to carry out the proposed tests. The prototype was implemented in the C language. This was due to the following reasons:

- the available libraries for client/server programming in a UNIX environment are mainly in C.
- Cryptographic material published in the literature was in C, so any published code could be incorporated in the prototype and tested and compared against other cryptographic components.
- To make coding practical for a large number of ciphers, it is useful to have a language with good pointer manipulation.

1.3.4 Tests used

In the project, randomness and statistical tests were used to investigate the confusion and diffusion properties of the block ciphers. Consequently, quantitative definitions for confusion and diffusion were proposed in order to support the evaluation methods of the developed ciphers.

The construction of the diffusion and confusion matrices also proposed in this project are the basis for computing the confusion and diffusion of the composed cipher instances. Furthermore, a property of the diffusion matrix was discovered which allows fast computation of the diffusion of a cryptographic composition of two or more encryption steps. Additionally, the quantity of marginal diffusion also proposed in this thesis has been shown to be highly related to the confusion.

Finally, the information in the confusion and diffusion matrices was utilised in two additional new tests which were proposed, namely the depth test and the diffusion distinguisher test.

1.3.5 Interpretation of results

After using the prototype to perform the tests, several classes of cipher instances were tested, allowing investigation of the number of rounds needed to achieve complete diffusion on a product cipher, and modelling the confusion, as defined in the thesis.

The modelling of confusion in terms of its contributing parameters was performed with regression analysis, in which the regression coefficients were examined to determine the proportion in which an equation described the experimental data. Furthermore, diagnostic tests were performed on the estimation equation, to determine its predictive power. The diffusion was derived directly from the diffusion matrices and therefore required no regression techniques.

Graphical representations of some parameters was used to aid visualising the experimental data.

1.4 Novel features of the thesis

The novel aspects of this thesis are:

- the definition of parameters related to the cryptographic strength of a product cipher:
  - the definition of diffusion and confusion matrices,
  - the quantitative definition of diffusion and confusion related to the matrices above,
  - the definition of the marginal diffusion,
- the evaluation of the cipher in terms of its encryption steps, resulting from:
  - the establishment of the multiplicative property of the diffusion matrices in product encryption which allowed modelling of diffusion,
  - the modelling of confusion with linear regression techniques,
- the on-line cipher evaluation approach, supported by:
  - the summary of the parameters of the encryption tests in the Cryptographic Block Profile introduced here,

- the description of the composed cipher for communication purposes,
- the depth test, and
- the diffusion distinguisher test.

1.5 Outline of thesis

The thesis is organised as follows. Chapter 2 presents a review of the relevant background literature. This involves description of the symmetric block ciphers, the cryptographic primitives which can be used as building blocks of these ciphers, their strength against certain cryptanalytic techniques, as well as their use in network security.

Chapters 3 and on describe the proposed method. Chapter 3 is a description and analysis of the proposed method, where the security framework and design objectives are also set. The chapter includes also a presentation and analysis of the developed methods - mainly statistical tests - which would be used to assess the cryptographic blocks. Chapter 4 describes the prototype developed to perform the long-run statistical tests, as well as the client/server application to examine the effectiveness of the short-run tests. Chapter 5 summarises the experimental results, followed by the interpretation of the results. The interpretation concludes with a regression analysis of the estimated parameters. Finally the conclusions and proposals for further research are presented at Chapter 6.

Part II

Literature Review

Chapter 2

Theoretical Background

2.1 Introduction

Cryptography[1], the practice of using encryption to conceal information, studies ways of *encrypting* (or *enciphering*) data by transforming it into apparent unintelligible forms, called *ciphertexts*. The description of such transformation process is called a *cipher*. The known history of codes and ciphers date back to 2000 B.C., when the Egyptians used a hieroglyphic code for inscription in tombs. Kahn (1976) presents a comprehensive historic survey of the use of cryptography throughout the years.

2.2 Terminology and definitions

The original form of a message is called a **plaintext**, whereas the encrypted form is called a **ciphertext**. Transformation from plaintext to ciphertext is called *encryption* or *enciphering*. The inverse process of transforming the ciphertext back to the plaintext, is called *decryption* or *deciphering*. A complete description of the encryption and decryption processes, is called a **cryptosystem**.

In a cryptosystem there is an extra piece of information which is needed for the encryption and decryption of the message. This information is the **key**.

[1]Cryptography, from the Greek κρυπτός (kryptos) and γραφή (graphy) which means *hidden writing*, is attributed to Thomas Browne in 1658, an English physician and writer (Bauer 1997).

There are two main categories of cryptosystems, the *symmetric* and *asymmetric cryptosystems.* A **symmetric cryptosystem** uses the same key both for encryption and decryption. The key is known only to the two parties which wish to communicate securely, and is called a *secret key.* An **asymmetric cryptosystem** employs two different keys, one for encryption and one for decryption. The sender of the message has knowledge only of one of the keys which is the *public key.* The public key is available to everyone; decryption though is not feasible with it. The second key is the *private key* and is known only to the intended recipient of the message. Only the private key is suitable for the decryption of the ciphertext. Such an arrangement has solved some important problems in modern communications, such as the key distribution and N-Square problem (Davies & Price, 1984).

2.2.1 Representations

The following representations were adopted from Bauer (1997) and will be used throughout the thesis.

Let $\mathcal{V}$ denote the set of symbols (characters) which may form a plaintext P. That is, $P = [p_1p_2 \ldots]$, where $p_i \in \mathcal{V}$, $i = 1, \ldots.$ Similarly, let $\mathcal{W}$ denote the set of symbols which may form a ciphertext C, where $C = [c_1c_2 \ldots]$, $c_i \in \mathcal{W}$, $i = 1, \ldots.$ $\mathcal{V}$ and $\mathcal{W}$ can be different, overlapping, or identical sets.

If $\mathcal{V}^*$ and $\mathcal{W}^*$ denote the sets of plaintext and cryptotext words (i.e. strings of symbols) constructed from $\mathcal{V}$ and $\mathcal{W}$ respectively, then $\mathcal{V}^*$ is the *plaintext space* and $\mathcal{W}^*$ is the *ciphertext space.* If $\mathcal{V}$ and $\mathcal{W}$ are nonempty finite sets, then $\mathcal{V}^*$ and $\mathcal{W}^*$ would be nonempty infinite sets. It would be assumed that $\mathcal{V}$ and $\mathcal{W}$ are nonempty finite sets. The cryptotext words are also called **cryptograms**. If a cryptogram consists of two ciphertext symbols, then it is called a **bigram**, if it consists of three symbols, then it is a **trigram** and so on.

In computing, the plaintext and ciphertext space usually consist of strings of binary digits. The set of symbols is $\{0, 1\}$ and denoted as $GF(2)$. A string of length n of binary digits is denoted as $GF(2)^n$ or $\{0,1\}^n$.

Let $\mathcal{V}^n$ ($\mathcal{W}^m$) denote the finite set of plaintext (ciphertext) words length n (m). Let ϵ denote the empty word. Then, if $\mathcal{V}^{(n)}$ is the set of all words of length up to n, it would be $\mathcal{V}^{(n)} = \{\epsilon\} \cup \mathcal{V} \cup \mathcal{V}^2 \cup \cdots \cup \mathcal{V}^n$, where $. \cup .$ denotes the union of two sets. Similarly, $\mathcal{W}^{(n)} = \{\epsilon\} \cup \mathcal{W} \cup \mathcal{W}^2 \cup \cdots \cup \mathcal{W}^n$.

An encryption E would be defined as a relation $E : \mathcal{V}^* \to \mathcal{W}^*$, where E specifies a mapping from set $\mathcal{V}^*$ to set $\mathcal{W}^*$. If the converse relation $E^{-1} : \mathcal{W}^* \to \mathcal{V}^*$ exists, then E^{-1} is a decryption. The decryption exists, if the relation $\mathcal{V}^* \to \mathcal{W}^*$ is unambiguous from right to left.

A rule finite set $\mathbf{M} = \{\chi_0, \chi_1, \dots, \chi_{\theta-1}\}$ of relations $\chi_i : \mathcal{V}^{(n_i)} \to \mathcal{W}^{(m_i)}$, is the *encryption system*, with χ_i the *encryption step* with θ denoting the cardinal number of the system. χ_i may be nondeterministic. If all encryption steps are injective[2], then a corresponding *decryption system* exists. Both encryption and decryption systems form a cryptosystem.

Let $\mathbf{X}$ denote a generated encryption $[\chi_{i_1}, \chi_{i_2}, \chi_{i_3}, \dots]$, where $\chi_{i_1}, \chi_{i_2}, \chi_{i_3}, \dots$ are encryption steps selected from $\mathbf{M}$. The encryption $\mathbf{X}$ is called *finitely generated*, if it is induced by a sequence $(\chi_{i_1}, \chi_{i_2}, \chi_{i_3}, \dots)$ of encryption steps in $\mathbf{M}$, under concatenation. It should be highlighted that a finitely generated encryption is not necessarily injective, even if the encryption steps are.

A **key** is used to select an encryption step from $\mathbf{M}$. Similar to $\mathcal{V}$ and $\mathcal{W}$, let $\mathcal{K}$ be the finite key space, with elements $k_i \in \mathcal{K}$, forming a key sequence $[k_1 k_2 k_3 \dots k_l]$.

2.2.2 Stream and block ciphers

Stream ciphers operate on one plaintext symbol at a time. In computers, the plaintext symbols are bits.

The independence of the symbols is a disadvantage from a security perspective. Because each symbol is separately encrypted, all information of a plaintext symbol is

[2]A transformation has an injective property, if it may be inverted without any ambiguities.

contained within one ciphertext symbol. Hence, the diffusion[3] is very low. Another disadvantage is that an intruder may insert a previous segment of the ciphertext, which may still look authentic.

Block ciphers operate on *blocks* of the plaintext, by grouping the plaintext symbols in the message and producing blocks of ciphertext, i.e. groups of ciphertext symbols. Plaintext and ciphertext blocks may have same or different length. It is generally required though (Shannon 1949) that the ciphertext blocks have the same length as the plaintext blocks.

In contrast to the stream ciphers, block ciphers may offer diffusion of a plaintext symbol within a block, in a way that it affects all ciphertext symbols of the generated block. This also makes it immune to insertions or modifications of ciphertext symbols in a given block.

2.2.3 Cryptographic strength

A well established approach to measuring the cryptographic strength of a cipher uses the concepts of confusion and diffusion. **Confusion** is the characteristic of the cipher, where changes in the plaintext result in unpredictable changes in the ciphertext. **Diffusion** is the characteristic, where changes in the plaintext can affect many parts of the ciphertext (Pfleeger 1989).

Confusion and diffusion have more meaning in block ciphers, which will be described in the next section. However stream ciphers lack high confusion and diffusion, which is an important disadvantage regarding cryptographic strength.

This thesis will be focusing on block ciphers, with binary strings used as plaintexts and ciphertexts, using 64-bit input and output blocks. Therefore, it would be useful to specify confusion and diffusion with respect to these parameters.

When mapping a 64-bit plaintext $[p_0 p_1 p_2 \dots p_{63}]$ to a 64-bit ciphertext $[c_0 c_1 c_2 \dots c_{63}]$, diffusion is defined as the relation of every input bit with every output bit, i.e.

[3]**Diffusion** is the characteristic which measures the extent to which a change in the plaintext affects many parts of the ciphertext, as explained at section 2.2.3.

$$c_i = f_{i,j}(p_j), \quad i,j = 0,1,\ldots,63$$

whereas the confusion implies that output bit c_i has 50% probability of being inverted, if bit p_j is inverted, for every i and j, i.e.

$$prob(c_i \oplus 1 = f_{i,j}(p_j \oplus 1)) = \tfrac{1}{2}, \quad i,j = 0,1,\ldots,63$$

where $\oplus$ denotes the bitwise **XOR** operation.

Confusion and diffusion are properties of the cipher and high values should be displayed for every key. However, it is not feasible to examine such a proposition in practice, since in principle the search space is large. Consequently, experimental measurement of these two characteristics could only be performed by statistical means, because measuring the whole space would require an effort greater than a brute force attack[4]. Another alternative would be to investigate the existence of properties of the underlying functions. Such properties, could be associativity, distributivity, isotopism, etc. If these properties do not hold, confusion would be high, since the existence of any additional property would make the relation between the input and output more apparent.

A method to measure confusion and diffusion is described in Chapter 3. Apart from these two measures, additional measures have been proposed in the literature which are mainly related to the underlying attacks and types of ciphers. For example differential characteristics are used in differential cryptanalysis and apply to product ciphers. These are described in Section 2.9.

2.3 Symmetric cryptography

As mentioned in Chapter 1, this thesis is more concerned with symmetric rather than asymmetric cryptography, because the proposed method uses a symmetric block cryptographic algorithm, with asymmetric cryptography used only to send structures and keys. Therefore secret key cryptosystems and more specifically block ciphers are presented.

[4]A *brute force* or *exhaustive search* is the most common and obvious attack, where all possible keys are tested until the correct key is obtained (Section 2.9)

2.3.1 Block ciphers

A block encryption is an encryption $[\chi_{i_1}, \chi_{i_2}, \chi_{i_3}, \ldots]$ where the mapping:

$$\chi_i : \mathcal{V}^n \rightarrow \mathcal{W}^m$$

holds for all $\chi_i \in \mathbf{M}$. A word from $\mathcal{V}^n$ is an encryption block. In todays computer technology, block encryption algorithms are mainly designed for encrypting $GF(2)^n \rightarrow GF(2)^n$, i.e. $n-$bit words. The set $\mathcal{V}^n$ is usually 64$-$bit words.

A block cipher may consist of permutations, substitutions and other functions which operate on the whole block or on a sub-block. If the block cipher consists of an encryption step applied more than one time, then the encryption step is the **round function** and the number of it being applied is the number of rounds of the algorithm. A block cipher which consist of many rounds belongs to the family of *product ciphers*, as described in the next section. In every round the encryption step may have same or different key values and operate on different segments of the block, i.e. on the *sub-blocks*. If different key values are used, then there is a part of the block cipher deals with the generation of these values from the main secret key. The process is called **key schedule**.

The importance of a key schedule was reported by Knudsen (1994a), as a distinction between *weak* and *strong* key schedules was made. According to Knudsen, a key schedule is strong if no *simple relations* could be found and weak in the opposite case. The following definition of a simple relation is adopted from Knudsen (1994b):

Definition Let E be a block cipher s.t. $E_K(.)$ denotes the encryption function using the key K and let f, g_1, g_2 be 'simple' functions such that the total complexity of one evaluation of each of f, g_1, g_2 is smaller than one evaluation of E (one encryption). Then if

$$E_K(P) = C \Rightarrow E_{f(K)}(g_1(P, K)) = g_2(C, K)$$

E is said to contain a **simple relation** between the encryption functions $E_K(.)$ and $E_{f(K)}(.)$

The well known complementation property of the DES for example (Davies & Price

1984), is a simple relation with $f(K) = \bar{K}$ and $g_i(X, K) = \bar{X}$.

An effective way to construct strong key schedules is to use a one-way[5] hash function, so the structure of the key would not be apparent at the end side of the schedule and therefore it would be very difficult to identify simple relations (Knudsen 1994a).

The main characteristic of a block cipher in terms of its security, is the high diffusion and confusion potential. Diffusion is offered mainly by permutations, whereas confusion is offered by substitutions. Usually a round's contribution in confusion and diffusion is trivial, but the iteration may produce substantially larger amounts of these two characteristics. There are various theorems and principles which investigate the conditions under which this desired phenomenon appears (Lai 1992); conditions which relate to this work are examined in this thesis. In general, security through multiple encryption depends on many parameters, ranging from the design principles of the substitution boxes, to the ways these primitives are combined with the key and plaintext information and the properties of the composition of the encryption steps. The properties and design principles of the cryptographic primitives are explained in Section 2.4. In this Section, some findings on cryptographic composition are presented, followed by the description of some block ciphers found in the literature.

Cryptographic composition

Ciphers which consist of composition of encryption steps are called **product ciphers**. It is necessary that the ciphertext space of the i-th encryption step coincides with the plaintext space of the $(i+1)-$th step. Bauer (1997) uses the term "superencryption" as a synonym of product encryption.

There are several requirements concerning the cryptographic strength of product ciphers. The result of cryptographic composition is not necessarily a stronger cipher. In some cases, cryptographic composition may result to a weaker system because one encryption step may counterbalance the effect of another step. In other cases, the

[5]One-wayness is the property in which it is relatively easy to evaluate a function, but computationally demanding to evaluate its inverse. One-way functions are presented in Section 2.6.

security of a cipher may be equal to that of the least secure encryption step. Desmedt (1991) for example demonstrated that if the encryption steps are homomorphisms the security of certain ciphers would not be strengthened.

In some cases it may be useful to examine whether a cryptosystem **M** forms a group (Bauer 1997), since if the group property holds, the effective key space would be reduced. The group property would hold for a cryptosystem if the composition of two encryption steps of **M** would be an encryption step in **M**. If a cryptosystem forms a group, then for two encryption steps χ_i and χ_j, there is an encryption step χ_k, such as for a plaintext p, $\chi_i(\chi_j(p)) = \chi_k(p)$. Thus, the key space forms a *key group*, i.e. $k = i \circ j$, with $.\circ.$ denoting the composition rule for the two keys. Another property a cryptosystem may have, is that it may be pure. A cryptosystem is called **pure** (Beker & Piper 1982), if for keys k_1, k_2, k_3 and k_4, the relation $\chi_{k_1}(\chi_{k_2}^{-1}(\chi_{k_3}(p))) = \chi_{k_4}(p)$ holds. A pure cryptosystem is a group, but a group is not necessarily pure.

Concerning a block cipher, the group property is an undesired characteristic, because not only does it provide the cryptanalyst more valuable relations to work with, but double or triple encryption does not influence the strength of a group cipher by any means. Consequently, there is a limit to the cardinality of the key space (and **M**) and it is related to the plaintext and ciphertext spaces. If the key space is large enough (comparable to the size of permutation mappings) to include all permutations of $\mathbf{V}^n \to \mathbf{V}^n$, then the cryptosystem is a group. However, a well designed cryptosystem would exclude the cases where the permutations reveal significant statistical relations between the plaintext and ciphertext.

Commutation[6] is also useful for examining the group properties of product ciphers; if two ciphers commute and each cipher forms a group, then their composition would also form a group (Shannon 1949). Consequently, to destroy the group property of a cipher, it is enough to find another cipher which does not commute with the first one and compose the latter with the former.

[6]Two encryption steps χ_1 and χ_2 commute if $\chi_1(\chi_2(m)) = \chi_2(\chi_1(m))$ for any plaintext m.

The Data Encryption Standard

The Data Encryption Standard, DES (Meyer & Matyas 1982), is a widely researched block cipher. Since its introduction to the public at 1977, the DES has been extensively analysed in order to establish its cryptographic strength. In the attempts to cryptanalyse it, many paradigms have been developed which contribute to block cipher design.

Although the actual design criteria for the DES have never been publicised (Schneier 1996), four years prior to the introduction of the DES, the following design criteria a cryptographic algorithm may fulfil were specified by the Federal Register (Schneier 1996):

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

However, the design criteria of the DES were not actually reported, until 1994. Probably this triggered the interest of various researchers to examine extensively the DES and try to identify its design principles. The design criteria of the DES involved the substitution and permutation boxes (S-boxes and P-boxes) (Section 2.5).

More specific, the criteria were as follows (Coppersmith 1994):

- Each S-box has a 6−bit input and a 4−bit output.
- No output bit of an S-box should be too close to a linear function of the input bits.
- If the two left-most and two right-most bits of an S-box are constant and the middle bits vary, each possible output should be attained only once.
- If the Hamming distance[7] of two inputs to an S-box is one, the Hamming distance of their outputs should be at least two.
- If two inputs to an S-box differ in the two middle bits, the output must differ in at least two bits.
- If two inputs to an S-box differ in the first two bits and are identical in the last two bits, the two outputs must not be the same.
- For any non-zero 6-bit difference between inputs, no more than 8 of the 32 pairs of inputs exhibiting that difference may result in the same output difference.
- Same as above, but for three S-boxes.

It is interesting to note that the criteria have a hint of making the S-boxes resistant to differential and linear cryptanalysis (Sections 2.11.1 & 2.11.2). However, these types of attacks where developed in the 90s, whereas the DES was designed in the 70s. Nevertheless, in today's state-of-the-art, more consistent criteria have been developed for S-box design. These are presented at Section 2.4.

The DES algorithm is shown in Figure 2.1. It operates on $\mathcal{V}^{64} = GF(2)^{64}$ producing ciphertext words in $\mathcal{V}^{64}$, using a 64−bit secret key. The actual size of the key is 56 bits, because every $8th$ bit is used for parity check. After being applied with an initial permutation P, the block is divided into to sub-blocks of equal length, L_0 and R_0. The round function $f : GF(2)^{32} \rightarrow GF(2)^{32}$ operates on the right sub-block and the output

[7]The Hamming distance between two binary strings is the number of positions of bits these strings differ.

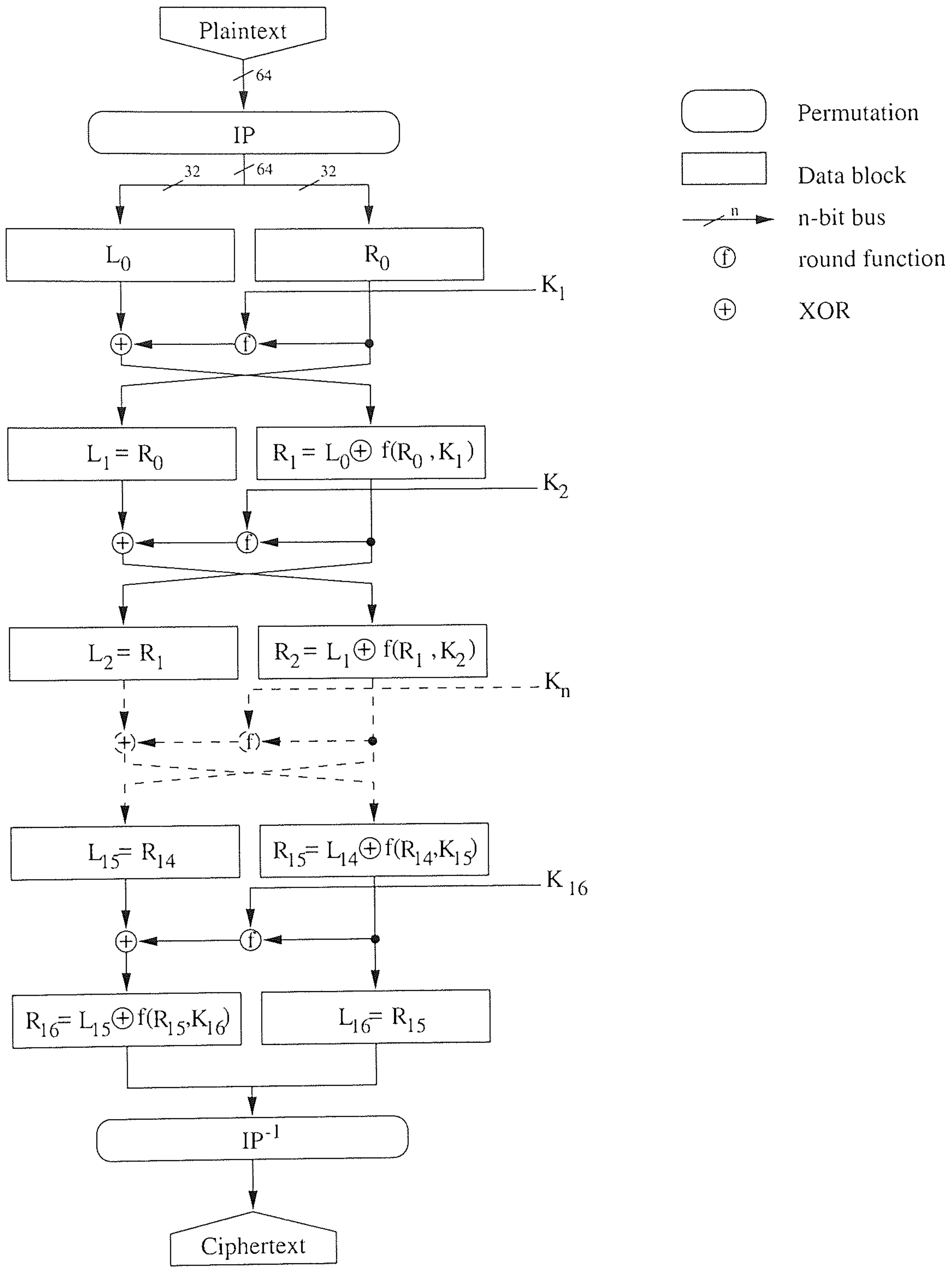


Figure 2.1: The Data Encryption Standard, DES.

is **XOR**ed with the left part. The round is complete by swapping the two sub-blocks. This process is repeated 16 times and finally the concatenated sub-blocks are applied with the inverse initial permutation, P^{-1}, to produce the ciphertext.

The round function f includes eight substitutions $s_i : \mathcal{V}^6 \to \mathcal{V}^4$. The 48−bit input results from a 48−bit subkey **XOR**ed with the- *expanded* from 32 to 48 bits- sub-block R_{i-1}, where i denotes the round of the algorithm. The expansion is simply some specific input bits being repeated. The sub-keys $K_1, K_2, \ldots, K_{16}$ are derived from the 56−bit secret key. For decryption the same algorithm is applied, but the subkeys are used in reverse order.

Modes of operation. The are four standard modes of operation specified for the DES, but they could be used for any block cipher:

Electronic Codebook mode, ECB. This is a straight-forward implementation of the DES. Each 64-bit plaintext word is processed separately, producing a 1-to-1 mapping between plaintext and ciphertext blocks. In ECB mode, an error in one bit of a ciphertext message, affects only the specific block. The disadvantage of the ECB mode is that it does not conceal repeated structures. If a plaintext block is repeated more than once, the resulting ciphertext blocks will be identical. This is a very common scenario in computer communications; packets constructed by the TCP/IP protocols have repeated patterns. Several forms, such as bank statements, use the same field names. It is relatively easy for an eavesdropper to identify in a bank statement the fixed fields, such as "Surname", "Initials", etc. and isolate them from the remaining blocks which contain more valuable information. In networks, encryption in ECB mode is not suggested for encrypting packets, since they have the "stereotyped beginning", such as "source", "destination", etc. (Davies & Price 1984).

Cipher Block Chaining mode, CBC. The CBC mode, as shown in Figure 2.2, includes the previous cryptogram to encrypt the proceeding plaintext: $c_i = DES(c_{i-1} \oplus p_i)$, and $c_0 =$ IV, an *initialisation vector*, agreed between

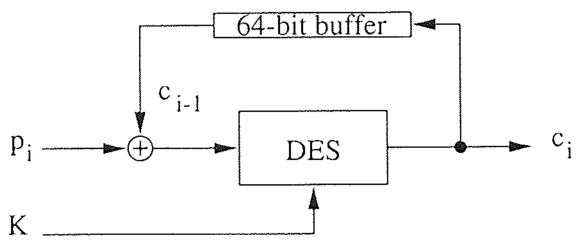


Figure 2.2: Cipher Block Chaining, CBC.

the communicating parties. The decryption would then be: $p_i = DES(c_i) \oplus c_{i-1})$. The feedback in the encryption is replaced by a feedforward in the decryption. The CBC mode has the property of being self-healing. If an error occurs in one or more bits in the ciphertext c_i, only the plaintext p_i and the corresponding bit in p_{i+1} would be wrongly decrypted. The existence of the **XOR** function cancels the errors, and p_{i+2} would be decrypted correctly. The CBC mode conceals the plaintext patterns, thus overcoming the weakness of the ECB mode.

Cipher Feedback, CFB. In some applications it may be unacceptable to wait for a whole 64–bit block to be received in order to perform encryption; encryption should be performed at a byte level. The CFB mode performs such encryption (Figure 2.3). The CFB mode is similar to a self-synchronising stream cipher. Any block less than 64–bits could be selected for CFB encryption, but in computers bytes (and more specifically octets) are most likely to be used. Encryption in CFB would be: $e_i = b_i \oplus B(DES(c_i))$, where B selects the leftmost byte of its input and discards the rest. In every encryption, the input to the DES is $c_i = (c_{i-1} << 8) \vee e_{i-1}$, with $c_0 = \text{IV}$. Similarly, decryption is defined as: $b_i = e_i \oplus B(DES(c_i))$, with the same c_i. It should be noted that in CFB mode both encryption and decryption apply the DES for encryption. As in CBC mode, in CFB patterns in the plaintext are concealed. Concerning error propagation, if there is an error in a byte, then the following seven bytes would be affected, until the wrong byte is discarded by the shift register.

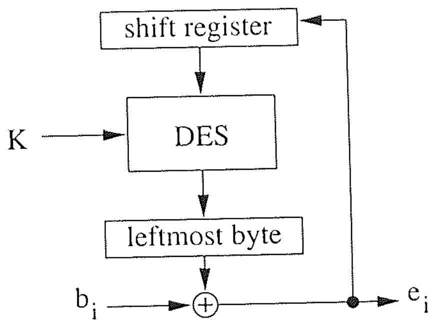


Figure 2.3: Cipher Feedback, CFB.

Output Feedback, OFB. In the OFB mode, the encryption is basically a stream cipher. From a publicly or privately agreed initialisation vector IV, a bytestream (or bitstream) is generated and **XOR**ed with the plaintext bytes (or bits). This mode has all advantages and disadvantages of a stream cipher, i.e. low propagation of errors, but also low diffusion. Patterns are also concealed in OFB, since every byte is **XOR**ed with a different pseudorandom output value.

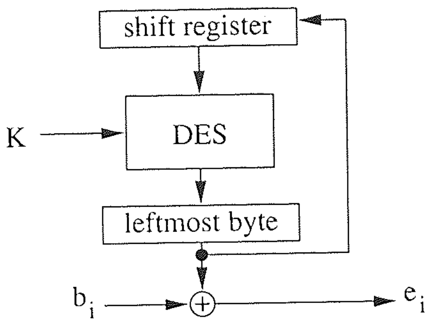


Figure 2.4: Output Feedback, OFB.

From the four standard[8] modes of operation described, only CFB can recover from synchronisation errors. Additionally, in ECB the least and most increment in ciphertext length is zero and 63 bits respectively, whereas in CBC these values are 64 and 127, because of the initialisation vector. Assuming that the plaintext arrives in a rate which is larger than the encryption rate of the ECB, then the fastest encryption of the four is ECB, but it is equal to the speed of CBC after a large number of encryptions. CFB and OFB are eight times slower, because to encrypt the same amount of plaintext eight times more encryptions should be performed. However, in OFB there could be preprocessing, making it the fastest mode of all four.

Non-standard modes of operation. There are numerous configurations for different modes of operations of DES-like cryptosystems found in the literature (Davies & Price 1984; Jansen & Boekee 1988). Some are derived from the four standard modes of operation, by interchanging the roles of the plaintext and ciphertext, or by combining the modes. Similar to CBC and CFB, plaintext block chaining, PBC and plaintext feedback, PFB, could be constructed. In the former, encryption would be defined as $c_i = DES(m_i) \oplus m_{i-1}$, with $m_0 =$ IV, whereas decryption would be $p_i = DES(c_i \oplus m_{i-1})$. In

[8]It should be highlighted that 'standard' refers to the modes of operation.

PFB, the encryption is specified from the equation $e_n = b_n \oplus B(DES((c_i << 8) \vee b_{i-1}))$ and the decryption would be $b_n = e_n \oplus B(DES((c_i << 8) \vee b_{i-1})$.

Although the general conclusions and properties of the standard operating modes could be applied in non-standard modes, in some cases an apparently insignificant change in the structure of the mode in a strong cryptosystem - a modification of the placement of a hot wire which connects a certain output with a certain input for instance - may lead to a very weak cryptosystem. Consequently, non-standard modes should be used with caution, since they have not been analysed extensively.

Security of the DES. A lot of research has been carried out concerning the security of the DES. Starting from the most obvious attack, the exhaustive search is likely to be economically feasible in the coming years. Garon and Outerbridge (1991) extrapolated that by the year 2000 the DES could be broken in a day for an investment cost of \$ 3,580,000.

The main concern throughout this thesis is the cryptographic strength rather the economics of exhaustive search. Generally, a cipher is considered to be secure, if there are no alternatives other than exhaustive search. If exhaustive search is feasible, this implies bad selection of key length. However, securing a cipher from alternative attacks is not straightforward, because it is not possible to consider all potential attacks - i.e. both published and new state of the art cryptanalytic techniques. Confidence in security against standard types of attacks, such as frequency distribution, differential and linear cryptanalysis, requires that the cipher has been analysed extensively. Yet, a slight modification of an attack scenario, may break the cipher. If the cipher is publicised and not broken, it is more likely that it is secure. This is stated in Kerchoff's principle (Schneier 1996), which suggests that a cipher can be considered to be cryptographically strong only if it is publicised and still not broken.

Attempts to analyse the DES have resulted in more general conclusions concerning block ciphers. The tools of mathematics were used to asymptotically analyse the DES. Luby and Rackoff (1986) adopted this theoretic approach, and concluded that the security of DES-like cryptosystems is related to the existence of pseudorandom bit generators. The DES did not fulfil this property, therefore it was considered insecure.

Concerning its algebraic structure, after a series of attempts by different researchers it was finally proven than DES does not form a group (Campbell & Wiener 1993). That is, that given two keys k_1 and k_2, there is no key k_3 such that $E_{k_2}(E_{k_1}(p)) = E_{k_3}(p)$. If this relation held, then triple encryption would not strengthen the security of the cryptosystem.

The key schedule was also found to have weaknesses. If the key is all 1s, all 0s, or half of it 1s and the other half 0s, then the generated subkeys are all identical. These are the **weak keys**. There are also six pairs of **semi-weak** keys that select the same DES permutation. However, the number of these problematic keys is trivially small, and does not have any significant impact on the security of the cryptosystem.

The round function of the DES introduces the subkey by **XOR**ing it with the (expanded) right sub-block, before performing the non-linear substitution. This configurations results to a complementation property of the cipher: if $E_k(p) = c$, then $E_{\neg k}(\neg p) = \neg c$, where $\neg k, \neg p$ and $\neg c$ denote the complements of k, p and c respectively. This suggests that one has to test half the possible keys, $2^{56}/2 = 2^{55}$ (Pfleeger 1989). It has not yet been determined whether the complementation property is a weakness (Schneier 1996).

The success of Biham and Shamir's (1991) differential cryptanalysis was found to be related to the number of rounds of the DES. More specific, it was shown that differential cryptanalysis is more efficient than exhaustive search, if the number of rounds of the DES is less than 16. Before this result, DES with reduced number of rounds was successfully cryptanalysed; Andelman and Reeds (1982) cryptanalysed a three and four-round DES; Chaum and Evertse (1986) broke a six-round DES.

Matsui (1994) who developed the linear cryptanalysis, described in the same paper how to linearly approximate the S-boxes. With this technique, an 8-round DES could be broken in 40 seconds, a 12-round DES in 50 hours, and a full 16-round DES is breakable with an effort smaller than exhaustive search.

The International Data Encryption Algorithm, IDEA

The IDEA[9] cipher by Lai (1992), was an improvement of the Proposed Encryption Standard, PES, (Lai & Massey 1991) which was broken with differential cryptanalysis (Lai *et al.* 1991). In the same paper a minor modification was suggested, and the cipher was then the Improved PES, but finally it was named IDEA.

As in DES, IDEA operates on $GF(2)^{64}$ plaintext blocks, and maps them to $GF(2)^{64}$ ciphertext blocks, but the key length is 128 bits, much larger than the 56–bit DES key. The blocks are divided into 16–bit sub-blocks, and encryption is completed in eight rounds, plus a final output transformation (Figure 2.5).

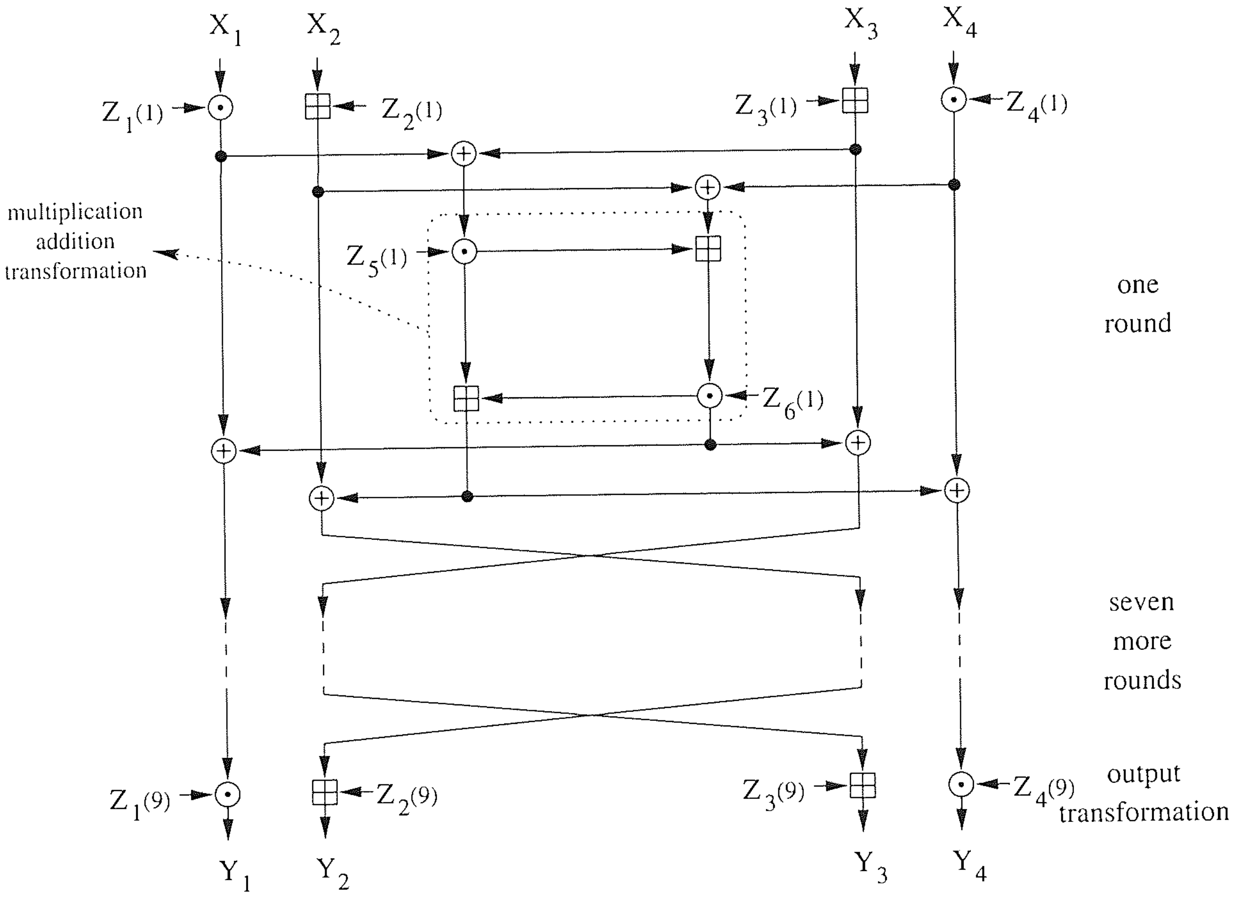


Figure 2.5: The International Data Encryption Algorithm, IDEA

As shown in Figure 2.5, IDEA consists only of operations; there are no substitution

[9]IDEA is a registered trade mark

boxes. More specifically, three algebraic groups are being mixed, namely:

- bitwise **XOR**, denoted as $\oplus$;
- integer addition modulo 2^{16}, where the 16-bit sub-block is treated as the radix-two representation of an integer; the operation is denoted as $\boxplus$;
- integer multiplication modulo $2^{16}+1$, where the 16-bit sub-block is is treated as the radix-two representation of an integer, except that the zero is represented by $2^{1}6$; the operation is denoted as $\odot$.

The multiplication operation could be viewed as a substitution block. The same algorithm is used both for encryption and decryption. The subkeys are generated as follows. For the encryption, the key is divided in the eight 16-bit subkeys. The first six are used for round one, and the remaining two are used in round two, as keys $Z_1(2)$ and $Z_2(2)$. Then the key is rotated to the left by 25 bits and is divided again into eight subkeys and four of them are used in round two as keys $Z_n(2), n = 3,4,5,6$. The remaining four are used in round three, etc. The decryption uses the additive or multiplicative inverses of the keys in reversed order. More specifically, subkeys one, four five and six in round i of the decryption, are the multiplicative inverses of the respective keys in round $(9-i)$ of the encryption. The remaining two, are the additive inverses.

Security of IDEA. The IDEA is of particular interest, because its design was based on theoretical foundations. Although that it was designed to withstand differential cryptanalysis, this was not proven finally, but the designer gave evidence that the cipher is immune to such attack, after four rounds (Lai 1992).

On its analysis, Lai considers diffusion and confusion and describes how these are offered by arithmetical operations. The first requirement for confusion was that the output of one type of operation should never be used as an input to an operation of the same type. The notion of *incompatibility* between operations was defined. Three operations $\oplus, \odot, \boxplus$ are **incompatible** if:

- no pair of the three operations satisfies a distributive law. For instance there exist $a, b, c \in GF(2)^{64}$ such that $a \boxplus (b \odot c) \neq (a \boxplus b) \odot (a \boxplus c)$;
- no pair of three operations satisfies a generalised associative law. For instance, there exist $a, b, c \in GF(2)^{64}$ such that $a \boxplus (b \oplus c) \neq (a \boxplus b) \oplus c$;
- no group of $(GF(2)^{16}, \oplus)$, $(GF(2)^{16}, \boxplus)$, $(GF(2)^{16}, \odot)$ is an isotopism with any of the other two under the direct bijective mapping. For instance, the groups $(GF(2)^{16}, \oplus)$ and $(GF(2)^{16}, \odot)$, are not isotopic for the direct mapping δ which is used, i.e. $\delta(x) \odot \delta(y) \neq \delta(x \oplus y)$, $\forall x, y \in GF(2)^{16}$;
- under the mapping δ, a linear- or generally an affine- function corresponds to a non-linear function;

Lai (1992) demonstrated through a series of theorems that the proposed algorithm satisfied all above conditions of incompatibility. Concerning the diffusion, it is provided by the multiplication addition structure, as shown in Figure 2.5. The multiplication addition transformation has the following characteristics (Lai 1992):

- the transformation is reversible for any choice of subkeys or input sub-blocks;
- it offers complete diffusion, i.e. every output bit is related to ever input bit;
- the structure uses the minimum number of operations required to offer complete diffusion;

Again these characteristics where proven to exist in this structure, through a series of theorems.

The impressive theoretical foundation behind the design of this cipher provides a formal indication of security. Additionally, any known attempt to cryptanalyze it has failed (Meier 1994). A class of weak keys were found, but this was a minuscule defect, since the probability on using one of these keys is one in 2^{96} (Daemen *et al.* 1994). However, IDEA is a relatively new algorithm and it has not been analysed extensively;

it has not been determined for example, whether IDEA forms a group. But generally it is deemed as one of the strongest algorithms (Schneier 1996).

2.3.2 Other symmetric algorithms

There is a plethora of symmetric algorithms found in the literature. The design criteria are not always public, because on the one hand such action was considered to jeopardise the strength of the cipher and on the other hand the designer may have included a trapdoor, so the algorithm is penetrated relatively easily. The DES took such an approach, although a number of criteria were published about a decade after it was available to the public.

Generally, developing a cipher was an art rather a science. One of the reasons was that there where no actual theoretical foundations concerning the security properties of certain classes of functions. However, a very complex cryptosystem may appear to be secure, but too much complexity could be illusionary (Bauer 1997). This was a common mistake many cryptographers made; numerous examples in the literature prove it (Kahn 1976). Yet, today much has been done on the theoretical side and a number of lemmas could assist in determining the security level of a cryptosystem. Such an approach was adopted for the design of IDEA as shown at the previous section.

The DES and IDEA have been presented particularly, because the DES, being the most popular algorithm, was the main drive in cryptanalytic research for the last 20 years and the IDEA is one of the best and effective instances of theoretic background applied to cipher design.

Some other of the symmetric algorithms found in the literature are RC4 (Schneier 1996), RC5 (Rivest 1994), FEAL (Shimizu & Miyaguchi, 1988; den Boer 1988; Gilbert & Chase 1991; Tardy & Gilbert, 1992), BEAR and LION (Anderson & Biham, 1996b), NewDES (Scott 1985), REDOC II (Cusick & Wood, 1991; Biham & Shamir, 1992).

2.4 Cryptographic primitives

In this section, the cryptographic primitives which are used mainly in block ciphers are presented. There are different views in the literature concerning what a cryptographic primitive is. In this thesis, a cryptographic primitive is considered any component that may perform a cryptographic transformation.

2.5 Substitution Boxes

Usually the substitution boxes (S-boxes) are the only non-linear step in an algorithm, and are the main contributors in a cipher's cryptographic strength. An S-box is a mapping of n input bits to m output bits. This is implemented efficiently in most of the programming languages with a one-dimensional array of n elements of m-bits each.

The relative sizes of m and n are very important to the cryptographic strength of the S-box. If $m \geq 2^n - n$, then there would be a linear relation of the input and output bits, and the S-box would not be resistant to linear cryptanalysis (Section 2.11.2). However, if $n \geq 2^m$, then there is a linear relation between the output bits (Biham 1995).

The orthodox method of S-box design (Adams & Tavares 1990) is to first determine the evaluation criteria. This ensures that an S-box will satisfy predetermined properties. The main advantage is that S-boxes generated in this way are resistant to known attacks, such as differential and linear cryptanalysis. However, such an approach does not ensure that the S-boxes produced are invulnerable to other attacks.

2.5.1 Evaluation criteria

An S-box generally is likely to be found in a one-way function, when used in a cryptographic algorithm. Adams and Tavares (1990), have specified the criteria a S-box must fulfil in order to be "cryptographically desirable":

- bijection

- nonlinearity
- strict avalanche
- independence of output bits

Bijection implies that the S-box is a permutation of the integers $2^n - 1$. This property is desirable in structures where the inverse of the S-box is also required. In cases such as one-way functions, bijection is not required.

Nonlinearity is a very desirable characteristic, because it is directly linked with the overall security of the structure. If the non-linearity property does not hold, the entire cryptosystem would be easily breakable.

The *strict avalanche criterion, SAC* defined by Webster and Tavares (1986) relates to *confusion* and *diffusion*. The SAC property requires that for every input bit i, inverting bit i causes output bit j to be inverted with a probability of 50%, for all j.

Independence of output bits ensures that no relation between two or more output bits could be determined, suggesting a strong cipher. The presence of correlation between two bits reduces the search space.

The design of an S-box usually involves the study of boolean functions. These functions could be constructed to satisfy certain properties. The most important property they should have is *non-linearity*. The perfect nonlinear boolean functions are called **bent functions** (Seberry *et al.* 1995).

Finding bent functions is a complicated task and they seem to be rare (Carlet 1993). Carlet (1993) defined a class of functions which are partially bent.

In general there is a trend towards systematic design of S-boxes (Pieprzyk 1996), rather than generating random ones. O'Connor (1994a,b) studied a class of random S-boxes and concluded that for relatively small sizes, such an approach is not effective. According to O'Connor (1994b), if the S-Boxes are large enough, the prediction probabilities of the differential characteristics are expected to be low, even if the S-Boxes have random values and not derived by some method to produce high non-linearity.

The relevant investigation of Biham and Shamir (1992), where the "good" S-Boxes of the DES were replaced by random ones concluded that the resulting cryptosystem was far weaker than the original one and the conclusion was that random S-Boxes are not a good practice. However O'Connor demonstrated that Biham's and Shamir's conclusion is only true for small S-Boxes. On the other hand, *strong* S-boxes could be useful for constructing families of cryptographic algorithms, and the characteristics of the S-boxes may determine security parameters of instantiations of these algorithms (Pieprzyk 1996).

2.5.2 Substitution-Permutation Networks

SPNs are based on Shannon's principles of a *mixing transformation* which consists of a number of rounds, where each round contributes a small amount of confusion and diffusion, but the combination results to higher values of confusion and diffusion (Shannon 1949). Based on Shannon's ideas, the SPNs were introduced by Feistel *et al.* (1975), with a network structure defined by a number of rounds. Each round has a series of small substitutions, followed by a permutation. The nonlinear substitutions offer confusion, whereas the permutations with their linear mixing, contribute to diffusion.

The parameters in a SPN are the number of input (output) bits N, the number of rounds R, the size of the S-boxes $n \times n$, and the number of S-boxes per round which would be $M = N/n$. Figure 2.6 presents an SPN for $N = 16$, $R = 4$ and $n = 4$ ($M = 4$).

Heys and Tavares (1996) set the evaluation criteria for an SPN to be the strength against linear and differential cryptanalysis. Their design criteria focused on diffusion and nonlinearity. The upper bounds on the probability of a differential characteristic and the deviation of the probability of a linear approximation from the ideal value of $\frac{1}{2}$ where specified. The differential and linear characteristics are used in cryptanalysis and in general the higher their values, the more efficient the cryptanalytic attack, as presented in more detail in Sections 2.11.1 and 2.11.2.

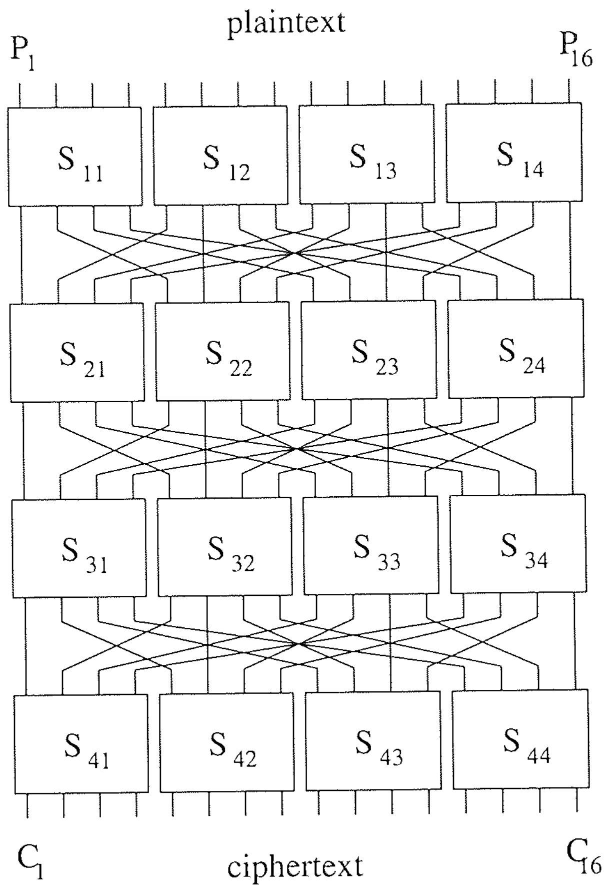


Figure 2.6: SPN with $N = 16$, $R = 4$ and $n = 4$ (source: Heys & Tavares 1996).

2.6 One-way functions

2.6.1 Background

A *one-way function* $f(x) = y$ is a function where it is relatively easy to compute in one direction, but significantly harder to inverse, i.e. for a given x one can easily compute $f(x)$, but for a given $f(x)$, it is hard to find x. The term "hard" though, adds an arbitrary description to the definition above. The term *"computationally infeasible"* is used formally in complexity theory; a problem is regarded as computationally infeasible, when the fastest known algorithm for finding the solution using the best available computer technology would require a significantly large time (comparable to the age of the Universe, say).

However, it has not been proved that one-way functions do or do not exist (Brassard 1979; Garey & Johnson 1979; Schneier 1996). There is no guarantee that a function which is perceived to have one-way properties, will still be perceived so after a month, a year, a decade, or a hundred years. Advances in technology affect the one-way status of a function. That is, in order for a function to exhibit useful one-way properties for a forseable time ahead, larger values must be used.

Advances in technology are broadly predictable: numerous extrapolations have been published regarding the computing power versus cost for the following years (Garon & Outerbridge 1991; Schneier 1995). The involvement of theoretical and applied mathematics has however introduced an extra uncertainty. A breakthrough in mathematics may result in development of new and highly efficient algorithms, to solve classes of problems in a "computationally feasible" manner. The conclusion is that since the existence of one-way functions has not been proved, a cryptographer must always be alert and constantly assess the one-way functions which are used in their cryptosystem.

Yet, most of the scenarios, particularly those in public key cryptography, rely on the existence of one-way functions. Goldwasser *et al.* (1988) demonstrated that digital signatures are equivalent to the existence of one-way functions. Furthermore, zero-knowledge protocols (Section 2.17.2) rely on the properties of one-way functions (Ostrovsky & Wigderson 1993). Blum & Micali (1982) have shown that pseudo-random generators are equivalent to the existence of one-way functions. In fact, the existence of pseudo-random generators - and more specifically of pseudo-random bit generators - is essential for provably secure symmetric cryptosystems (Luby & Rackoff 1986).

2.6.2 Trapdoor and hash one-way functions

There are two categories of one-way functions which are widely used in cryptography, namely the *trapdoor one-way functions* and the *one-way hash functions.*

A **trapdoor one-way function** $f_k(M)$ is a one-way function which can be inverted, given some extra information, k. This extra information is called the *trapdoor* (Menezes 1993). A **one-way hash function** $h(M) = H$ is a one-way function which operates on an arbitrary-length input M, to produce a fixed-length output h (Schneier 1996).

It should be highlighted that a hash function is not necessarily a one-way function. Merkle (1979) lists the additional characteristics a hash function must have, in order to be a one-way:

- Given M, it is easy to compute H.
- Given H, it is hard to compute M such that $h(M) = H$.
- Given M, it is hard to find another message M', such that $H(M) = H(M')$.

The first two characteristics ensure the one-wayness of the function. However, in some applications the last characteristic is required. There are cases where different messages hash to the same value, because the space of input messages could have an arbitrary size, whereas the space of the hashed values has always a fixed length. When two different messages hash to the same value, we have a *collision*. Therefore, the third characteristic ensures the *collision-resistance* of the function.

Collision resistance one-way hash functions are suitable for digital signatures; if an intruder could find collisions, they could forge digital signatures. Some functions have been shown not to be collision resistant. The FFT-Hash for instance, was proposed by Schnorr (1992) and it was shown that collisions could be found by performing only 2^{24} computations of the main function of FFT (Baritaud *et al.* 1993). The next version, FFT-Hash-II was proposed after a year (Schnorr 1993), but Vaudenay (1992) proved still other collisions. Vaudenay (1996) describes also a method to find collisions on the Digital Signature Standard (DSS), by forging the public parameter.

However, there are some additional characteristics for a one-way hash function:

- The one-way hash function must be multiplication-free (Anderson & Biham 1996a).
- It is not feasible to find X and Y such that the Hamming weight of $h(X) \oplus h(Y)$ is less than one would expect to get from random chance if we calculated $h(M)$ for a large number of M (Anderson 1995a).

If the function has the multiplicative homomorphism property, it could be exploited by an attacker (Anderson 1995a). This characteristic is not required in all functions, but should be present in cryptosystems such as RSA, since it involves modular multiplication. The last characteristic implies *correlation freedom.* Okamoto (1993), argues that correlation freedom is a stronger property than collision-freedom and Anderson proved this conjecture (1995a).

There is also a special category of one-way hash functions which are present in many security protocols, the *keyed one-way hash functions.* A **keyed one-way hash function** is obtained when a message M is hashed together with a secret key. Such functions are used for message authentication (Johnson *et al.*1991; Tsudik 1992; Rivest 1991, 1992a, 1992b; RACE 1992).

Other known one-way hash functions include SHA which is used in the Digital Signature Standard, DSS (Federal Register, 1992), Tiger (Anderson & Biham, 1996a), Haval (Zheng *et al.*, 1993) and Snerfu (Merkle, 1990). Furthermore, one-way hash functions based on non-standard modes of operation of block ciphers were also proposed, such as MDC-2 and MDC-4 (Meyer & Schilling, 1988), LOKI (Brown *et al.*, 1990, Lai, 1992) and AR Hash (ISO 1992, Damgard & Knudsen, 1994).

2.7 Feedback Shift Registers

Shift registers have been studied extensively for the last five decades. The areas of application range from cryptography and error correcting codes, to prescribed property generators and mathematical modelling (Golomb 1967). Shift registers are *finite state machines.* By definition, a **finite state machine** is a device which consists of a finite number of states, and accepts sequentially inputs from a finite set, and produces a sequence of outputs from a finite set. Such devices can efficiently be implemented in hardware.

2.7.1 Linear Feedback Shift Registers

A **Feedback Shift Register, FSR**, is represented at Figure 2.7. As it can be seen, it consists of a shift register and a *feedback function*. Each of the boxes labelled x_i is a binary storage element. These are n in total and are called *stages* of the shift register (Golomb 1967). At periodic intervals, each of the components x_i, passes its contents into x_{i-1}. The feedback function is some boolean function $f(x_1, x_2, ..., x_n)$ of the terms x_i and specifies the new value for x_n.

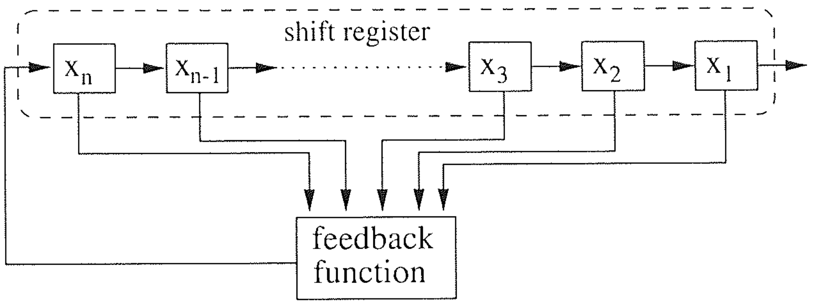


Figure 2.7: A Feedback Shift Register

If a feedback function can be expressed in the form

$$f(x_1, x_2, ..., x_n) = c_1x_1 \oplus c_2x_2 \oplus ... \oplus c_nx_n$$

where each of the constants c_i are either 0 or 1, then this function is called a *linear*, and the underlying FSR is called a **Linear Feedback Shift Register, LFSR**. LFSR are the most common type of shift registers used in cryptography, mainly because they have been consistently analysed and they could have certain prescribed properties, such as a specified period. On the other hand, non-linear functions are generally avoided, because non-linearity may lead to very short periods, i.e. the lenghts of the bit sequences (Bauer 1997). In fact, Siegenthaler (1986) showed that non-linearity not only is insufficient to prevent cryptanalysis, but may actually aid it.

It is mostly desired in cryptography that a LFSR had a maximum period. That is, given any initialisation values to x_i's - except the vector $(0, 0, ..., 0)$ - the LFSR would again obtain those values after visiting all $2^n - 1$ states. Maximum period is very important in cases where LFSRs are used as stream ciphers and they produce key bits which are **XOR**ed with the plaintext, to produce the ciphertext (Figure 2.8). In those cases, the secret key would be the initial state of the x_i buffers (i.e. an n-bit key). The "maximum period" property can ensure that the produced key stream length is comparable with that of the plaintext, so that certain cryptanalytic attacks will not succeed.

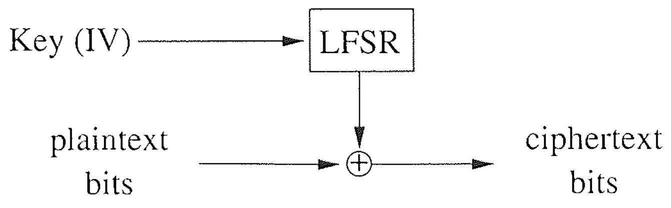


Figure 2.8: A Feedback Shift Register used in a stream cipher

Fortunately, there is a way to determine whether a LFSR has a maximum period. It is related with the feedback function and the constant coefficients c_i. These select which register bits are to be **XOR**ed in order to produce the boolean output. A list of these c_i is a *tap sequence* or a *Fibonacci configuration* (Schneier 1996). Every tap sequence could be represented as a *characteristic* polynomial *mod* 2. For instance the sequence:

$$1100101$$

would form the polynomial:

$$x^7 + x^6 + x^3 + x + 1$$

The *degree* of the polynomial is its highest power of x with non-zero coefficient, and in the case of an LFSR, it is the length of the shift register. In order for a particular LFSR to have a maximum period, the tap sequence must form an *primitive polynomial.* A **primitive** polynomial of degree n, is a polynomial which is an irreducible factor of $x^{2^n-1} + 1$ over $GF(2)$ (Berlekamp 1984).

Constructions which combine several LFSRs in a variety of ways in a nonlinear manner include the Geffe generator (Geffe, 1973), the Jennings generator (Jennings, 1980), then MSR-generator (Mund *et al.*, 1988) the stop-and-go generator (Beth & Piper, 1984), the alternating step generator (Günther, 1987) and Rueppel's self-decimated generator (Rueppel, 1988).

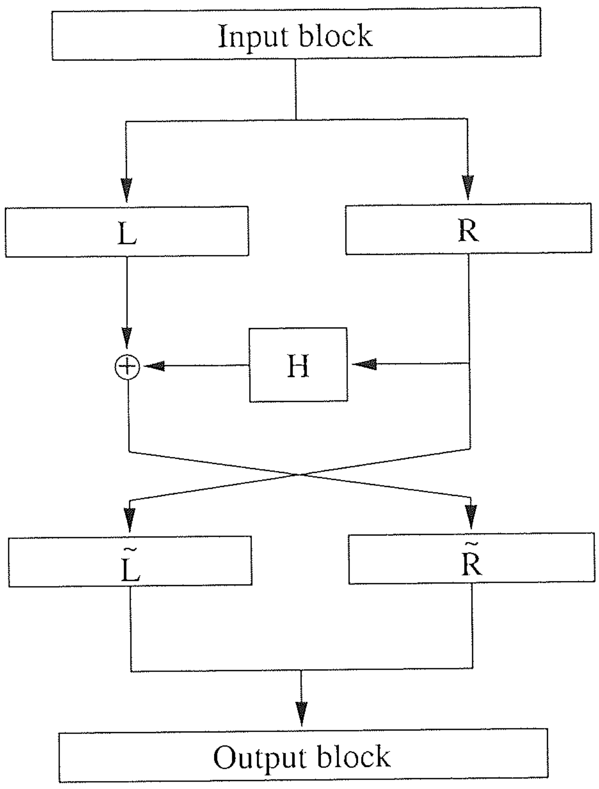


Figure 2.9: A Feistel block

2.8 Feistel Networks

Feistel networks (Feistel 1974) are the most common constructions used in block algorithms. They have great value in cryptography, because they enable a cryptosystem which employs it to be *provably* secure, as will be demonstrated at section 2.8.1. A Feistel block is presented at Figure 2.9. The input message block of length n is broken into two sub-blocks L and R, with length n_L and n_R respectively. If $n_L = n_R$, which is normally the case, then the encryption block is called *balanced*. The function H is a strong keyed one-way hash function, with input n_R bits, and output n_L bits. The output of the function is **XOR**ed with block L, and the final two blocks are swapped:

$$\tilde{L} = R$$

$$\tilde{R} = L \oplus H_k(R)$$

A **Feistel network** is a construction which consists of a number of rounds of a Feistel block. In each round, a different key is used in the function H. In a symmetric block cipher, the key for each round is generated by the secret key. Such process is called *key scheduling* (Davies & Price 1984). Most of the block ciphers use key scheduling, including the DES. In particular, the DES is a 16-round balanced Feistel network, with an initial permutation at the beginning, and its inverse permutation at the end of the 16*th* round. The 56-bit key is scheduled to generate sixteen 48-bit subkeys. Consequently, the block cipher could be described as follows:

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

$$R_i = L_{i-1} \oplus H_{k_i}(R_{i-1})$$

where i denotes the ith round of the block cipher.

The Feistel network described is a *Balanced Feistel Network, BFN* since the left and right blocks are of equal size. An *Unbalanced Feistel Network, UFN* is a Feistel network where the two blocks do not have equal sizes. Most standard block ciphers which are based on Feistel networks are BFNs. Furthermore, if the same round function is used in every round with different sub-key, the Feistel network is called **homogeneous**. If different round functions are used, the Feistel network would be **heterogeneous**.

In terms of cryptographic strength, Schneier & Kelsey (1996) conjecture that heterogeneous UFNs should be more resistant to some cryptanalytic attacks such as linear and differential cryptanalysis (Section 2.11.1).

Unfortunately there is not much systematic research in UFNs, because there were not many standard ciphers based on UFNs. Furthermore, more attention was drawn to the round function rather than the topology of the Feistel network. Schneier & Kelsey (1996) described an approach for a systematic description of all Feistel network-based block ciphers. Their paper is presented in Chapter 3 (description of the proposed method) because the definitions are used throughout the proposed method.

The Feistel networks are interesting in symmetric block cipher design mainly for three reasons:

- The same configuration is used both for encryption and decryption. This means that there is no need to implement two different algorithms. The only difference is that the keys should be scheduled in reverse in the case of decryption to that of encryption.

- There are virtually no restrictions for the function H. A one-way hash function is preferable, but any hash function could be used. This does not affect the invertibility of the block cipher, but its cryptographic strength. That is because the structure of a Feistel block always passes the same inputs to the function at a given round at the encryption to that of the corresponding round at the decryption.

- Under certain conditions, the symmetric block cipher can be provably secure, as it is presented at the next section.

2.8.1 Luby & Rackoff

Luby and Rackoff (1986) in an attempt to analyse the DES and determine its security by quantitative means, have drawn several conclusions which initiated a burst of research activities based on their results (Schnorr 1988; Pieprzyk 1991; Anderson & Biham 1996b). Initially, Luby and Rackoff demonstrated that the existence of pseudorandom function generators (Goldreich *et al.* 1984), yields the existence of pseudorandom permutation generators. Then, they used these conclusions to construct a provably secure block cipher and derived the properties where generally a block cipher can be secure (Luby & Rackoff 1988). The relevant papers involve complexity-theoretic and information-theoretic (Maurer 1993) analysis and are beyond the scope of this thesis, but the main steps and conclusions will be outlined, since these conclusions justify the cryptographic method proposed in this thesis.

Let $\{0,1\}^n$ be the set of all 2^n strings of length n. Let $\mathcal{F}^n$ denote the set of all 2^{n2^n} functions mapping $\{0,1\}^n$ into $\{0,1\}^n$. The subset of $\mathcal{F}^n$ which are invertible and one-to-one, are the permutations and let $\mathcal{P}^n$ denote this set. The cardinality of $\mathcal{P}^n$ would be $(2^n!)$, substantially smaller than that of $\mathcal{F}^n$. Let $h_k^{2^n}$ be a function in $\mathcal{P}^n$

indexed by a key k of a given length. Apparently, to address all elements of $\mathcal{P}^n$, the key must have length $\log_2(2^n!)$ bits.

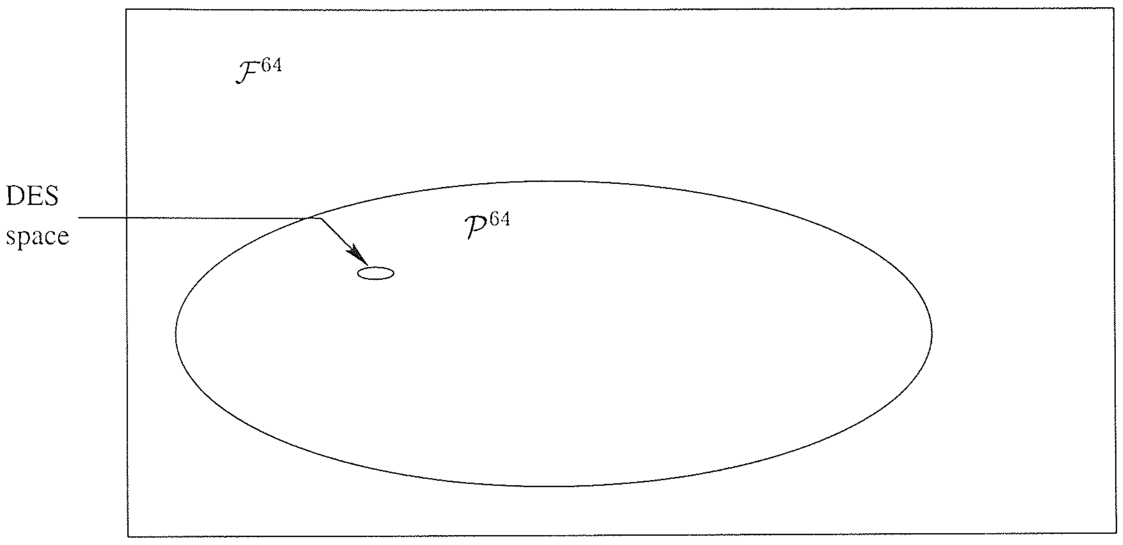


Figure 2.10: Relation of $\mathcal{F}^{64}$, $\mathcal{P}^{64}$ and DES space

If the above notations are applied in the case of DES, then $n = 64$, $k = 56$. With a smaller key, not all permutations could be selected, but only the 2^{56}, a number substantially smaller compared to the size of $\mathcal{P}^n$, $(2^{64}!)$ (Figure 2.10). The question is whether the permutations that can be selected are cryptographically strong - or in other words, whether the permutations which yield structure and non-randomness are excluded. Informally, this statement is the *black box test*, which was suggested by Turing (Hodges 1985) and for the case of DES is as follows (Luby & Rackoff 1988):

> "Say that we have two black boxes, one of which computes a fixed randomly chosen function from $\mathcal{F}^{64}$ and the other computes h_k^{64} for a fixed randomly chosen k. Then no algorithm which examines the boxes by feeding inputs to them and looking at the outputs can obtain, in a "reasonable" time, any "significant" idea about which box is which."

The terms "reasonable" and "significant" introduce ambiguities in the black box test. Luby and Rackoff (1988) used the tools of mathematics and computer science to asymptotically analyse the DES and introduced the notion of *asymptotic security*. By doing this, "reasonable" was replaced with "computationally infeasible" and "significant" was related to the probability of distinguishing the functions of the two black boxes.

Let $L \bullet R$ denote the concatenation of the two n-bit binary strings, L and R. For any permutation $f \in \mathcal{F}^n$, an invertible permutation $\bar{f} \in \mathcal{P}^{2n}$ can be defined as:

$$\bar{f}(L \bullet R) = R \bullet [L \oplus f(R)]$$

i.e. the left and right parts are swapped and the right part is unchanged, whereas the left part is **XOR**ed with $f(R)$. This corresponds to one DES step. More particular, the DES could be defined as a permutation which is the composition of 16 such steps:

$$\psi(f_1, f_2, \ldots, f_{16}) = \bar{f}_1 \circ \bar{f}_2 \circ \cdots \circ \bar{f}_{16}$$

Luby and Rackoff (1988) motivated by the structure of the DES, defined a mapping: $H : \mathcal{F}^n \times \mathcal{F}^n \times \mathcal{F}^n \to \mathcal{P}^{2n}$, which is a three round DES:

$$H(f_1, f_2, f_3) = \psi(f_1, f_2, f_3)$$

An *oracle circuit*[10] C_{2n} was considered for distinguishing a function randomly chosen from $\mathcal{F}^{2n}$ from a function chosen from a much smaller set $\psi(\mathcal{F}^n, \mathcal{F}^n, \mathcal{F}^n)$. Let

$$p_1 = P[C_{2n}(f) = 1 : f \in_R \psi(\mathcal{F}^n, \mathcal{F}^n, \mathcal{F}^n)]$$

denote the probability that C_{2n} outputs 1 if the oracle gates are evaluated for a function chosen randomly from $\psi(\mathcal{F}^n, \mathcal{F}^n, \mathcal{F}^n)$, and

$$p_2 = P[C_{2n}(f) = 1 : f \in_R \mathcal{F}^{2n}]$$

the probability that C_{2n} outputs 1 if the oracle gates are evaluated for a function chosen randomly from $\mathcal{F}^{2n}$. Luby and Rackoff demonstrated the following lemma (Maurer 1993).

Lemma: *Let C_{2n} be an oracle circuit with k oracle gates such that no input value is repeated to an oracle gate. Then*

$$|p_1 - p_2| \leq k^2/2^n$$

Luby and Rackoff proved that composition of slightly weak functions may result to very strong ones, but the condition is true only if pseudorandom function generators exist, since they can construct pseudorandom permutation generators, as presented above. In turn, pseudorandom function generators can be constructed from pseudorandom bit generators. Consequently, the validity of the above lemma by Luby and

[10] An oracle circuit C_{2n} is a circuit with oracle gates, i.e. gates with $2n$-bit input and $2n$-bit output and all oracle gates evaluate the same fixed function in $\mathcal{F}^{2n}$.

Rackoff, rests on the existence of pseudorandom bit generators, as described by Goldreich *et al.* (1984).

In practice however, it is infeasible to construct $\psi(f_1, f_2, f_3)$, due to memory limitations. For $n = 32$ for instance, the number of required bits would be $3 \cdot 32 \cdot 2^{32} \approx 4 \cdot 10^{11}$. In general, provable security is a trade off between generalisation and efficiency.

The work of Luby and Rackoff triggered the interest of many researchers concerning the provable security of ciphers and more specifically, of Feistel transformations . Zheng *et al.* (1990) provided a practical alternative modifying the Feistel transformation by introducing several types of transformations which would result to provably secure ciphers but would also be possible to implement them with the current technology. The same researchers described a way to construct a distinguisher for any transformation of the form $\psi(f^i, f^j, f^i)$ where $f^k = f \circ f \circ \ldots (k\text{times}) \ldots \circ f$, proving that it is impossible to construct a three round Feistel transformation with only one function. Pieprzyk (1991) proved that a four round Feistel $\psi(f, f, f, f^i)$ with $i \geq 2$ is pseudorandom, if f is a pseudorandom function.

Yet, in a recent paper by Aiello and Venkatesan (1996), a distinguisher was suggested, which could distinguish a four-round Feistel from a random permutation on $2n$-bit messages, given $O(2^{n/2})$ chosen plaintexts. Furthermore, Coppersmith (1996) motivated by this work, developed an attack to deduce the actual contents of the round functions in a Feistel configuration, with $O(n2^n)$ chosen plaintexts.

In conclusion, four rounds of a Feistel cipher is not enough, if a chosen plaintext attack is mounted, because the required information for distinguishing and breaking the cipher is leaked, in a relatively low number of computational steps (Coppersmith 1996). In practice most ciphers have more than eight rounds, but this is due to the fact that the round functions used are far from pseudorandom bit generators.

2.9 Cryptanalysis

Cryptanalysis is the study and development of methods to deduce plaintext from ciphertext without prior knowledge of the decryption key and in some cases, without knowledge of the cipher. In some cryptanalytic methods this is achieved by recovering the decryption key, whereas in other methods there is no need to do so.

The most obvious attack on a given cryptosystem is the **brute force** or the **exhaustive search**, where different keys are tested until the correct key is found. This attack applies to all ciphers. The question is whether such attack is feasible. If the cost to break the cipher is greater than the gain one has from breaking it, then exhaustive search is not worthwhile. In this case, the task of cryptanalysis is to develop a method which requires less effort to break the cipher.

2.10 Classes of attacks

Depending the information available, a cryptanalyst may mount one of the following attacks:

- **Ciphertext only attack**. In this type of attack, the cryptanalyst has only the ciphertext and the publicly available knowledge, such as frequency distribution of letters in a language. The cryptanalyst's task would be to deduce the plaintext and/or the key used to encrypt the messages.
- **Known plaintext attack**. In this case, the cryptanalyst has a number of plaintext and ciphertext pairs. The task would be to identify the encryption or decryption algorithm and deduce the key, to use it for decrypting future messages.
- **Chosen plaintext attack**. In this case, the cryptanalyst has the ability to chose the plaintexts that the system would encrypt and has also access to the corresponding ciphertexts. The task would be to deduce the secret key.

- **Chosen ciphertext attack.** In a chosen ciphertext attack the cryptanalyst has access to the decryption algorithm and may test large amounts of ciphertext and by observing the plaintext, the task is to deduce the decryption key.

The attacks which in general attract the interest of cryptographers are the chosen plaintext and chosen ciphertext attack because they are the strongest type of attacks, especially the latter. Many cryptographic algorithms fall for chosen ciphertext attacks. The chosen plaintext and ciphertext attacks are *oracle-like* attacks, because the cryptanalyst queries the cryptosystem (oracle) which in turn responds with the ciphertext or plaintext (answer).

2.11 Cryptanalysis of product ciphers

Ciphers which result from composition of classes of encryption steps are more secure because they can hide any characteristics such as patterns or redundancy of a language. Therefore, the cryptanalytic methods described so far are not applicable to product ciphers. Two other cryptanalytic attacks overcome the weaknesses of the attacks presented so far. These are the differential and the linear cryptanalysis.

2.11.1 Differential cryptanalysis

Differential cryptanalysis, introduced by Biham and Shamir (1991) is a very powerful cryptanalytic tool for breaking block ciphers and more precisely DES-like cryptosystems, i.e. cryptosystems which consist of an iteration of a weak function in a Feistel network. Their research was initiated by an attempt to cryptanalyze the DES but it was discovered that many published block ciphers fall to this attack.

The method was based on the fact that the possible output values of the substitution boxes were not uniformly distributed. This is done by examining the resulting differences of ciphertext pairs, caused by respective input plaintext pair difference. More analytically, differential cryptanalysis operates as follows (Biham & Shamir 1993a):

First, for the round function of the DES, two inputs **X** and **X′** produce the outputs **Y** and **Y′** respectively. Since all these values are known, their **XORs** are also known, $X \oplus X' = \Delta X$ and $Y \oplus Y' = \Delta Y$. Additionally, ΔA and ΔC (Figure 2.11) are also known.

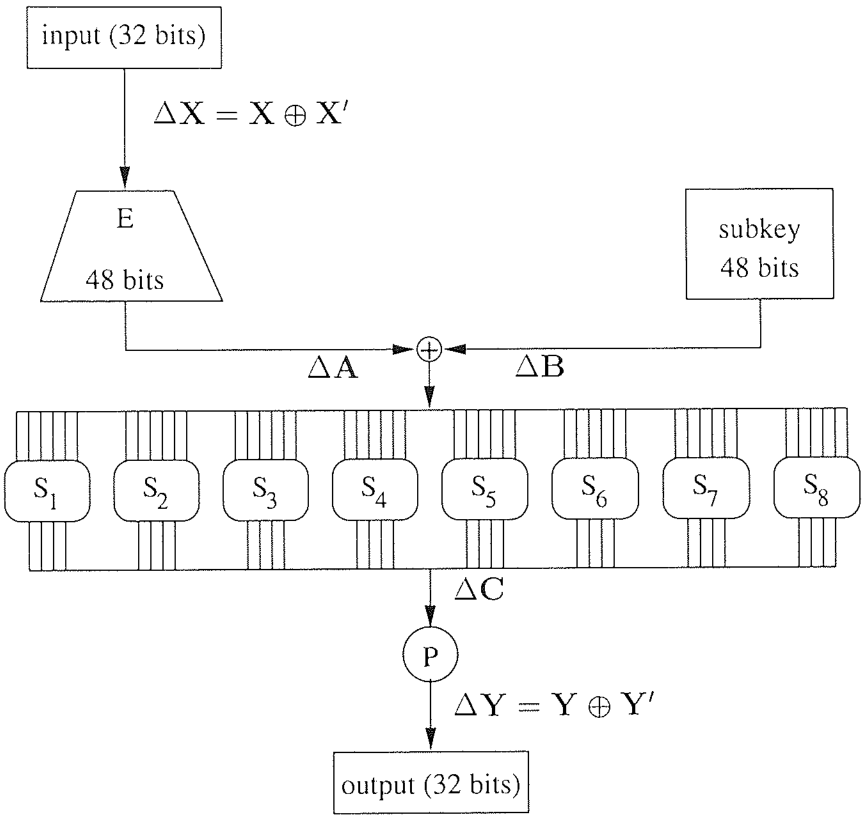


Figure 2.11: The round function of the DES.

The lack of uniformal distribution of the S-Boxes, results in having some pairs of ΔA and ΔC appearing more times than others. This leaks information about the key bits; since ΔA may generate only a limited number of distinct ΔCs, the possible keys would be found by the set determined by **A XOR K$_i$** and **A′ XOR K$_i$**, where K_i denotes the *ith* candidate key.

In differential cryptanalysis the notion of a characteristic is introduced. A **n-round characteristic** is a pair of given ciphertext difference produced by a plaintext difference by a n-round algorithm, associated by the probability of occurrence of these two differences. The characteristic is represented as Ω. The interesting property of characteristics, is that in many rounds the overall characteristic would be the product of all one-round characteristics, under the assumption that the rounds are independent. This assumption implies that the subkeys are independent; in the case of the DES this is not true, but the assumption was needed to simplify the analysis.

A plaintext pair which satisfies a last round characteristic is a *right pair*, whereas in the opposite case it is a *wrong pair*. When a right pair is found, the suggested key is the correct one. On the contrast, a wrong pair would suggest a random value for the key. Since the subkey is 48 bits long for the DES, the remaining 8 bits could be recovered by exhaustive search.

From a practical point of view, a large number of runs is needed in order to find many right pairs. A sufficient amount of information must be accumulated until the correct key can be separated from the noise. Biham and Shamir (1991) successfully applied the method to a reduced number of rounds, but the full 16 round DES would need a large effort, but smaller than a brute force.

2.11.2 Linear cryptanalysis

Linear cryptanalysis introduced by Matsui (1994) is a known plaintext attack in which the cryptanalyst searches for linear relations between certain plaintext, ciphertext and key bits, requiring that the probabilities of these relations being satisfied, are other than 50%. Such relations are *linear approximations* and could be constructed for various number of rounds for a block cipher. If a block cipher with a block input $[p_1p_2\ldots p_n]$, block output $[c_1c_2\ldots c_n]$ and an $m-$bit key $[k_1k_2\ldots k_m]$ is considered, in linear cryptanalysis the attacker would derive an expression such as:

$$p_{i_1} \oplus p_{i_2} \oplus \ldots p_{i_a} \oplus c_{j_1} \oplus c_{j_2} \oplus \ldots c_{j_b} \oplus = k_{l_1} \oplus k_{l_2} \oplus \ldots k_{l_c}$$

which holds for probability $p_L \neq \frac{1}{2}$ over all keys. It is required that $|p_L - \frac{1}{2}|$ is maximal, since this will require the lowest complexity for a successful attack.

Obviously, its easier to find one-round linear approximations. As with differential cryptanalysis, the weakness of the round function for the case of the DES rests on the

S-Boxes. The best linearly approximated S-Box is S_5, where the four bit outputs have the same parity of the second input bit for 12 of the 64 inputs. This property could be utilised to construct the best linear approximation in the DES which holds with a probability of $12/64 = 0.19$

Linear approximations for different rounds could be composed in a similar manner as the round characteristics are composed in differential cryptanalysis. For a reduced three round DES, the best relation could be applied on the first and third round, and the combination would give the linear approximation:

$$p_7 \oplus p_{18} \oplus p_{24} \oplus p_{29} \oplus p_{47} \oplus c_7 \oplus c_{18} \oplus c_{24} \oplus c_{29} \oplus c_{47} = k_{22}^{(1)} \oplus k_{22}^{(3)}$$

which would hold with a probability of $(12/64)^2 + (1 - 12/64)^2 = 0.70$. In the above equation, $k_i^{(n)}$ represents the i*th* bit of the n*th* subkey.

The basic method of linear cryptanalysis can recover only one bit of the key, but further processing and additional techniques could be applied in order to recover more key bits. These techniques reduce the number of rounds of the linear approximations by eliminating the first and last rounds, and counting all the key bits which affect the data (Biham 1995).

Biham (1995) studied linear cryptanalysis in contrast to differential cryptanalysis and showed that although these two methods differ, they are similar at the structural level. In both methods round characteristics are defined but the concatenation rule for each method is different. The DES is broken with linear cryptanalysis with 2^{47} known plaintexts (Matsui 1994) and with differential cryptanalysis by mounting a 2^{47} chosen plaintexts (Biham & Shamir 1993b). The latter attack was a refined version of the original method. Since linear cryptanalysis can break the DES with a known plaintext attack, it could be conjectured that the cipher is weaker in linear than in differential cryptanalysis. The design criteria of the DES (Coppersmith 1994) do not provide evidence that the cipher was designed to withstand this type of attack.

2.11.3 Differential Fault Analysis

Differential fault analysis, DFA (Biham & Shamir 1997) is a relatively new and completely different approach in cryptanalysis. This method was inspired by a similar approach of Boneh *et al.* (1997) for public key cryptosystems, where a computational error could leak information by using certain algebraic properties of modular arithmetic.

In their paper, Biham and Shamir first described a potential attack on the DES. The attack consisted of two parts. On the first part the round where the error occurred was identified. A software implementation of the DES was used in order permit insertion of errors at desired instances. The actual place of the error was not necessarily known, but could be determined. One-bit errors were also assumed. The method encrypted a plaintext two times and the two corresponding ciphertexts where compared. If there was a difference, then one of the encryptions had an error. Assuming that the error was always on the left input sub-block to the round, if the difference was one bit, then the error was at the last round. If the difference was more bits, then depending on the number of differences, the round where the error occurred could be determined. On the second part, the positions of the differences are examined and compared with the distributions of the S-Boxes - the same tables used in differential cryptanalysis. The process could reveal the key bits. It is interesting to note that if the attacker can cause faults to appear in the last two, three or four rounds, this analysis requires only about 10 ciphertexts (Biham & Shamir 1997).

However, such attack requires that the attacker is able to pose some kind of strain on the device and also requires that the strain produces faults in a controllable way. In practice and especially in computer networks this is an unrealistic scenario.

However, DFA could be particularly useful when the cryptographic algorithm is unknown, and it is contained in a tamper resistant device. The SKIPJACK algorithm of the Clipper chip (Brickell *et al.* 1993) is one example of such device. Smart cards are also tamper resistant devices, with applications ranging from phone cards to electronic wallets. Assume that a tamper resistant device contains a secret key stored in an EPROM. Assume that the device allows only writing to the EPROM. Even if the cipher is unknown, the key could be deduced as follows. The EPROM is gradually exposed in ultraviolet light. Eventually the key bits which have a logical '1' value should be erased, i.e. set to logical '0'. By controlling the intensity of the UV light, one can cause the resetting of the key bits one by one. During the process of erasure, a constant plaintext is encrypted and the ciphertexts are recorded. Eventually, the encryption will produce the same ciphertexts. All the key bits would be zeroes. The number of different ciphertexts would be equal to the number of ones in the key. By working backwards, at most n bits should be tested for every '1', where n is the length of the key. On average the correct key would be determined after $O(n^2)$ encryptions.

2.12 Cryptanalysis of modes of operation

The modes of operation are usually used for hiding patterns, protection against chosen plaintext attacks and are designed in a way that the error propagation is low and data synchronisation would be feasible (Biham 1994). Therefore, it was assumed that a cryptosystem would be more secure if the underlying cipher is used in a mode of operation. Furthermore, a combination of multiple modes of operation should increase considerably the security of the cryptosystem. Yet Biham (1993, 1994, 1996) demonstrated that certain combinations of multiple modes of operation are not as strong as they ought to be.

The approach used the layout of the modes as shown in Figure 2.12. By doing this, one can easily trace the paths which the data follow and draw conclusions about the strength of the cryptosystem. It is important to note that the cryptosystem would not be less secure than its strongest single mode component, only if the subkeys are independent and only if the attacker has knowledge of the plaintexts. The same result was proven by Even and Goldreich (1985) but only for cascade ciphers.

In general, Biham (1994, 1996) described several cryptanalytic techniques for the

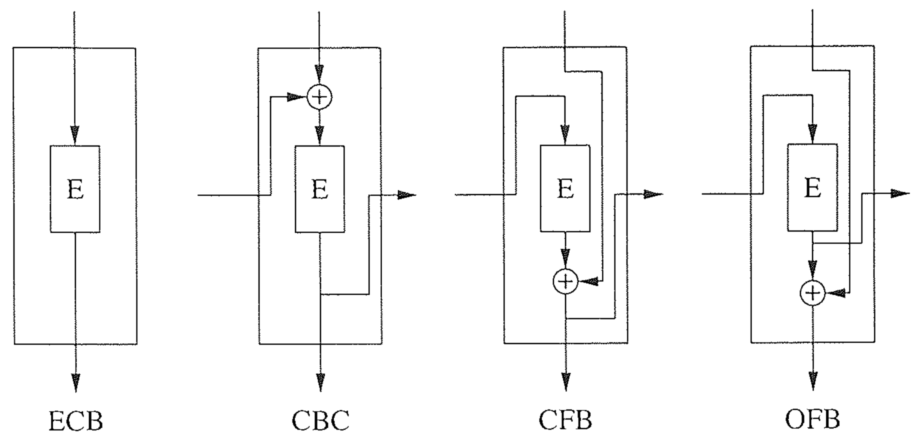


Figure 2.12: Standard modes of operation.

various standard modes of operation and for the combination of modes by cascading the different configurations. The outcome was that the cascade of standard modes of operation of the DES were more secure than the strongest of the participating ciphers only if the subkeys to every cipher were independent. However, the whole construction was not significantly more secure than the strongest cipher. Additionally, if the subkeys were not independent, the security of the underlying cryptosystem, would be equal to the security of the least secure component. For instance, if a configuration of three cascaded ciphers would be used and the first cipher had the same key as the third cipher, a cryptanalyst would attempt to obtain the key from the weaker of the two, and automatically he would have the secret key of the strongest cipher.

The attacks require knowledge of the configuration of the cascade and modes of operation of each step. If the configuration is known, then the general objective would be to select plaintext and ciphertext values, so that the required data for differential cryptanalysis is fed to the desired block for cryptanalysis. Biham conjectured that if the mode is built around the whole configuration instead of cascading the modes together, the cryptosystem would be more secure, since it would be more difficult to control and access the intermediate values.

The triple standard modes of operation were not much more secure than single modes, so Biham (1996) suggested non-standard modes, which they are conjectured to be more secure. More specifically, the configuration '$M1 \to M2$' was defined, where

$M1$ is any stream mode generating a plaintext-independent stream and encrypting it under any mode $M2$ and finally **XOR**ing the result with the plaintext. Also, another configuration '$M1[M2]$' was defined as a stream mode $M1$ where the output is **XOR**ed before and after any mode $M2$. An extension would be to use cascade modes and **XOR**ing the same value of $M1$ to all intermediate points. This extension is denoted by '$M1[M2M3 \ldots Mn]$'. Under these notations, the secure modes were conjectured to be OFB→CBC→CBC and OFB→CFB→CFB similar to an OFB mode and the modes OFB[CBC,CBC^{-1}], OFB[CFB,CFB^{-1}], OFB[CBC,CBC] and OFB[CFB,CFB] as secure chaining/feedback modes (Biham 1996). CBC^{-1} and CFB^{-1} denote the decryption in CBC and CFB modes respectively.

2.13 Tests for randomness

As mentioned in Section 2.2.3, a "good" cipher is one which maintains high levels of diffusion and confusion. Statistical tests for randomness examine ciphertext with its respective plaintext and key. The purpose for the series of these tests is to determine whether a cipher is capable of causing unpredictable changes to a ciphertext, when known changes occur either in the plaintext or the key. If the cipher fails to pass the tests, then its is generally considered weak, because it leaks information about the key or the plaintext.

Unfortunately, the opposite is not necessarily true. That is, if the cipher passes the tests, this does not guarantee a secure cipher. It may be a candidate for a strong cipher, but it should be made public so it may be extensively analysed. Statistical tests may show that a given sequence is random, but this does not suggest anything about the unpredictability of the sequence- a highly desirable characteristic in cryptography.

In general, tests for randomness apply to the output of the stream ciphers, before they are mixed (usually **XOR**ed) with the plaintext. Tests of randomness also apply to block ciphers to measure the confusion and diffusion.

In the following sections the tests which are used in the evaluation of the instantiations of the classes of ciphers proposed in this thesis are presented.

2.13.1 The frequency test

The most common test is the frequency or equidistribution test (Knuth 1981). It would be reasonable to expect that given a stream of bits, half of them would be zeroes and half of them would be ones. For a sequence of bits of length n, let n_0 and n_1 be the number of zeroes and ones respectively. Then, the chi-square is computed to test the hypothesis that $n_0 = n_1$ for one degree of freedom (Beker & Piper 1982):

$$\chi^2 = \tfrac{(n_0-n_1)^2}{n}$$

2.13.2 The serial test

The serial test checks the transitional probabilities, i.e. from 1 to 1, from 1 to 0 from 0 to 1 and from 0 to 0. If 11 occurs n_{11} times, 10 n_{10} times, 01 n_{01} times and 00 n_{00} times, it would be desirable that $n_{11} = n_{10} = n_{01} = n_{00} \approx \frac{n-1}{4}$ (Beker & Piper 1982). It has been shown that (Good 1957):

$$\frac{4}{n-1}\sum_{i=0}^{1}\sum_{j=0}^{1} n_{ij}^2 - \frac{2}{n}\sum_{i=0}^{1} n_i^2 + 1$$

approximates the χ^2 distribution for two degrees of freedom. Consequently, the above formula could be used for performing the serial test.

2.13.3 The autocorrelation test

The autocorrelation test examines whether the number of zeros and ones are randomly distributed within the sequence. If the sequence is $\alpha_1\alpha_2\ldots\alpha_n$ and $A(d)$ is defined as (Beker & Piper 1982):

$$A(d) = \sum_{i=1}^{n-d} \alpha_i\alpha_{i+d}$$

then if the sequence has n_0 zeros and n_1 ones randomly distributed, the expected value of $A(d)$ for $d \neq 0$ would be:

$$\mu = \frac{n_1^2(n-d)}{n^2}$$

Again, hypothesis testing should be used to test whether $A(d) = \mu$.

2.13.4 The block cipher test

Consider a block cipher operating on a $n-$bit plaintext with a $m-$bit key to produce an $n-$bit ciphertext (Piper 1998). It is not necessary that the plaintext and ciphertext blocks have the same length, but this is usually the case.

Let $w(x)$ denote the weight of the binary string x. A random plaintext is selected p_0 and encrypted using a randomly selected key k_0. Let $p_1, p_2, \ldots, p_n$ denote the plaintexts which $w(p_0 \oplus p_i) = 1,\ i = 1, \ldots, n$ and for each i, $i-1$ would denote the position where the difference occurred. Let $c_0, c_1, \ldots, c_n$ be the corresponding ciphertexts. We would expect the distribution of $w(c_0 \oplus c_i)$ to be as in Figure 2.13, for all $i = 1, 2, \ldots, n$.

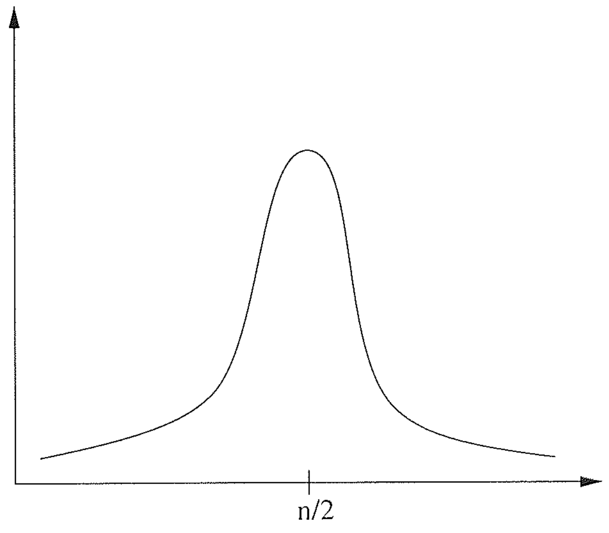


Figure 2.13: Distribution of $w(c_0 \oplus c_i)$.

The cipher would pass the test if the variance is between certain values. It would be unrealistic to expect zero variance and larger values imply un-balancedness.

With the same data the confusion could also be measured. Simply, if:

$$e(i) = \sum_{j=1}^{r} (c_0 \oplus c_j)_{[i]}$$

where r denotes the number of encryptions ($r = cn$) for all $i = 1, 2, \ldots, n$ and $(b)_{[i]}$

denotes selection of the $i-$th bit of a string b, it would be desirable for a cipher to have a distribution of $e(i)$ as in Figure 2.14.

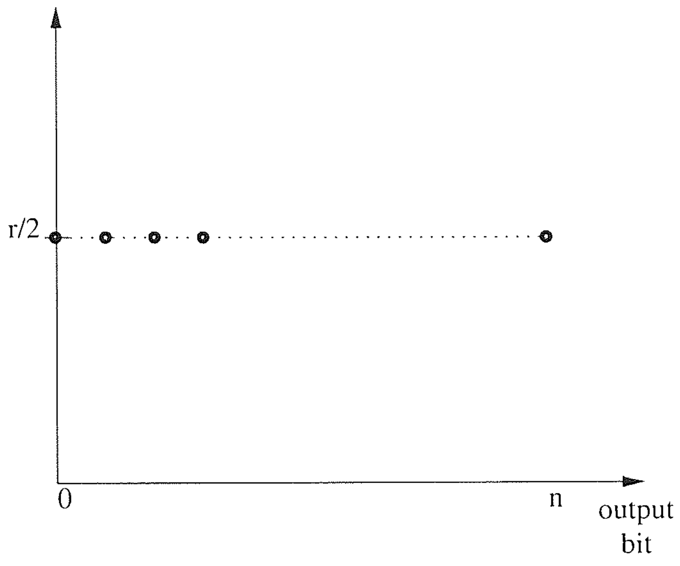


Figure 2.14: Distribution of $e()$.

The same tests should be performed considering the key as the variable and changing with the same manner as the plaintext.

2.14 Computer network security

Existing major operating systems and networking protocols were not developed with high levels of communications security in mind. Widespread growth of networking encouraged a wide range of users, many of whom required high level of security in their communications. High levels of interconnection allow many people to access systems from anywhere, which increases likelihood of malicious or accidental damage. In addition, the trend towards client/server architecture, especially in open systems, makes security vital.

Client-server networks can be attacked by service requests, where no password or other security protocol is necessary. The client-server approach enables users to make requests such as a database query and use such request to initiate unauthorised processes. Users may start processes in different systems and each process may initiate a new one on behalf of the user (Kaufman 1993). Control messages are exchanged between users and processes, so the reliability of the system relies on the integrity of these control messages (Janson & Tsudik 1993). Thus, the validity (integrity and authenticity) of information in a communication channel relies on the secure and certified exchange of sensitive information. The secure exchange of information in a network is the foundation of network security.

2.15 Security services

Although different networking environments require different approaches to security implementations, the following *primary* security services are desirable at most cases:

- **confidentiality**, i.e. information is not disclosed to unauthorised individuals, parties or processes,
- **integrity**, i.e. data have not been destroyed or generally modified in an unauthorised manner,
- **authentication** (of host/user/data), which is a service that provides a proof that the received data is from the source it claims to be (data origin authentication) or - in a case of peer association - a peer entity is the one that claims to be, and
- **non-repudiation**, which offers either a proof of origin, or a proof of delivery.

The primary services are desired in most networking environments, whereas the secondary security services such as access control, selective field confidentiality, integrity with or without recovery, traffic flow control, etc., should be described by the policy. It should be emphasised that authentication and integrity are mutually dependent, and lack of one of the two, would result to failure of providing the other. For instance, if the integrity of the authentication data is not offered, an intruder may modify the data without being noticed, and authentication would fail. Similarly, one should not expect to receive data which pass - say an integrity checksum test - but the origin is not confirmed, since the messages may be transmitted from an unauthorised origin.

2.16 Classes of attacks in communications

There are two main categories of attacks in communications, namely passive and active eavesdropping. Passive eavesdropping- or interception- is when the eavesdropper is situated at some position in a communication channel and reads data as it passes through this channel. In most cases it is impossible to detect a passive eavesdropper. Today, a lot of sensitive information such as passwords flows unencrypted on the Internet. Passive eavesdropping programmes called *sniffers* could easily log the first messages of initialisation of communication sessions, which contain information about user names and passwords. With the use of encryption, the passive eavesdropper is forced to decrypt the captured ciphertext.

Active eavesdropping though could be categorised into four different classes:

- Interruption. An attacker may cause interruption of communications in a network by flooding the channel with a large number of packets. This would result to a communications failure. A serious problem caused by interruption is the denial of service of a legitimate user (Needham 1994).

- Modification. If there is no authentication and integrity, the eavesdropper may capture the packets, modify some of the contents to suit his personal benefit and transmit them.

- Fabrication. An eavesdropper may also send his own packets but also modify the headers, so all the packets may appear to be sent by another (legitimate) source.

- Replay. An eavesdropper may retransmit packets he captured some time in the past.

Active eavesdropping is the so called **man-in-the-middle** attack (Schneier 1996). According to this attack, an eavesdropper may be placed between the communicating parties and impersonate each party to the other. The legitimate parties may believe that they are communicating directly, but in practice, the attacker would control the data. Straightforward implementations of some public key cryptosystems, such as the RSA, are vulnerable to this attack.

2.17 Security protocols

A security protocol is an orderly sequence of steps taken by two or more parties in order to enjoy the security services. Pfleeger (1989) lists the characteristics a protocol must have:

- Established in advance. The protocol is completely designed before it is used.
- Mutually subscribed. All parties to the protocol agree to follow its steps, in order.
- Unambiguous. No party can fail to follow a step properly because the party misunderstood the step.
- Complete. For every situation that can occur, there is a prescribed action to be taken.

Specifications of security protocols are considered also for the different layers, such as the network layer (Atkinson 1995a, 1995b, 1995c; Kumar & Crowcroft 1993; Ioannindis & Blaze 1994; Cheng *et al.*1995), the transport layer (Mirhakkak 1993; O'Higgins & Schnider 1990; Katsavanos & Varadaharajan 1993) and the application layer (Winfield & Wolman 1993), as well as certain networking protocols such as the file transfer protocol (Postel & Reynolds 1985) and the network time protocol (Mills 1992; Bishop 1992).

2.17.1 Representations

It is a common practice to use names for the participating parties for convenience, when describing a security protocol. The same approach is adopted in this thesis. The parties are as follows:

Alice	Alice is the sender of a message, or initiates a communication.
Bob	Bob is the intended recipient Alice wishes to communicate with.
Eve	Eve is the eavesdropper.
Trent	Trent is the trusted third party.
Victor	Victor is a third party verifier.

The format:

Alice→Bob: X

implies that Alice has sent Bob the message X.

2.17.2 Key distribution - negotiation

The key distribution is probably the most critical part for secure communications. Public key cryptography has boosted the use of cryptography in open systems, and its contribution to key distribution is remarkable. There are many approaches for key agreement in practice, but because of the different ways Internet users can connect and interact, the use of public key cryptography is the most flexible and efficient way for the users to obtain the needed keys (Maugham *et al.* 1995). Yet, there are issues such as key management; a public key directory is helpful for avoiding the man-in-the-middle attack, for instance.

Another desirable characteristic a protocol must have, is that it must provide **perfect forward secrecy**. That is, if a long-term master key is disclosed, an attacker would not be able to deduce the session short-term keys which were derived from the master key (Günther 1990). Krawczyk (1996) proposed a key distribution scheme for the Internet, focusing on perfect forward secrecy.

No matter how mathematically complicated a key agreement protocol is, it is based on one of the following scenarios:

- Alice generates a random key and encrypts it using a public key algorithm, such as RSA, with Bob's public key. The encrypted random key is sent to Bob.
- Alice and Bob use a public key algorithm to exchange public information. Once the exchange has taken place, they use each other's public information along with their own secret keys to produce a common value and produce a secret key.

The complexity of key distribution- and generally security protocols- cannot be arbitrarily high. The protocols are subject to environmental restrictions (e.g. small packet headers, limited processing power, etc.). Under this consideration, a family of light weight protocols was proposed by Bird *et al.* (1995). The first stage of these protocols is strong two-way authentication performed before the key distribution or exchange.

Third party key distribution

Key distribution may also involve a trusted third party. A paper by Davis and Swick (1990) describes **private key certificates** which are administered and used as public key certificates; the protocol offers the advantage that users can communicate securely while sharing neither an encryption key or a network connection. Briefly the protocol is as follows. Suppose that Alice and Bob wish to communicate securely. Instead of sharing a private key to exchange messages, they could use Trent as an intermediary. Trent shares a key between Alice and Bob separately. Bob sends a message to Alice, encrypted with the common key he shares with Trent. If Alice wants to read the message, she sends it to Trent, who translates it into her key. Such approach is scalable, allowing multiple users to communicate.

Scalability is mostly needed in open systems. The work of Davis and Swick agrees with the work by Chikazawa *et al.* (1990), where a key-sharing system for global telecommunications is proposed for multiple user participation and a trusted third party. The latter approach has some advantage over the former, since the proposed scheme prevents the conspiracy among users and employs one-way communications, which are more desirable in high rate networks. According to Chikazawa *et al.*, a distribution centre generates a n-dimensional vector, where n is the number of participating users, and a common key is based on this vector; each user has its "view" of the key, where it is combined with his secret information.

A protocol for electronic cash on the Internet was proposed by Brands (1994). The system involved also tamper-resistant devices (smart cards) where the user's secret key, balance and certificate of the public key were stored. The certificate of the public key is a value which results from a keyed hash function computed by an issuing authority (e.g. a bank). For performing payments, the user (buyer) issues an electronic cheque which would consist of his/her public key certificate and an electronic signature of the user to the amount and identity of the service provider. The service provider sends the payment transcript of the electronic cheque to the bank, and its amount is credited.

The third party key distribution protocols, have addressed the man-in-the-middle problem described above (Section 2.16), but are practically more difficult to be included in open architectures. A typical example is the X.500 authentication system (Kille 1991), which was designed for the Internet but is not widely used, because the Internet was developed in an ad-hoc fashion and no specified authority was responsible for this open system.

A protocol based on Diffie-Hellman key exchange but involving a trusted third party, was proposed by Diffie *et al.* (1992) and is called the *station-to-station, STS* protocol. According to the STS protocol, Alice and Bob have public key certificates which consist of their name and public key, sealed electronically with Trent's signature. The protocol performs key exchange with authentication in three rounds.

Secret sharing

Secret sharing involves the scenario where a group of people are shared a piece of a secret and a certain number of them is required to collaborate, in order to construct this secret. If m people are involved and at least n are needed to construct the secret, then they are involved in a (m, n)-**threshold scheme** (Shamir 1979).

The main principle behind a threshold scheme is that a trusted third party which has knowledge of a secret, can compute **shadows** of it and distribute it to the members. More analytically, a message in a (m, n)-threshold scheme is one of the variables in a n vector solution of an equation in a finite field. The shadows are instances of this equation. Therefore to solve an equation of n unknowns, n points would be needed (Blakley 1979). It should be highlighted that even if $(n - 1)$ shadows are put together, the respective people wouldn't have any more information compared to what they knew before they exchanged the shadows.

There are several variations and types of secret sharing schemes found in the literature. These may involve secret sharing with the ability to prevent cheaters, i.e. when one or more users disclose a fake shadow (Brickell & Stinson 1990, Benaloh 1987). Other schemes may allow construction of the secret without revealing the shares (Desmedt & Frankel 1990, 1992), without a trusted third party (Ingemasson & Simmons 1991), or with the ability to disenrol a member (Martin 1993).

Zero knowledge protocols

Zero knowledge protocols, which appear to have a great theoretical interest, are authentication protocols, but could also be used for key exchange. The term *zero knowledge* is used because no secret information is revealed during the conversation. Compared to public key protocols, zero knowledge systems require smaller computational resources (Aronsson 1995). Thus, zero knowledge protocols are attractive in smart card applications (Schnorr 1991).

According to zero knowledge theory, Alice can authenticate herself to Victor by proving that she knows a solution to a *difficult* problem. More formally, a problem must be NP-complete (Naor *et al.* 1992) and Alice must prove that she can solve it in a polynomial time. A problem could be a proof of knowledge of factoring a large number (Rabin 1979; Feige *et al.* 1987), or finding a Hamiltonian cycle in a graph. The zero knowledge protocols involve interaction between Alice and Victor, where Victor may request as many rounds of proofs he wants, in order to be convinced.

Message authentication

Authentication of messages could be performed with **digital signatures**. A digital signature involves the sender's private key over the message. If Alice encrypts a message with her private key, Bob could use Alice's public key to decrypt it. Confidentiality would not be offered in this approach, since the decryption process could be performed by any user who knows Alice's public key.

However, this method has the problem that Eve may record two or more messages and concatenate them together; the resulting signature would also be valid. In some public key cryptosystems this is feasible. One solution would be to sign the hashed value of the message.

Message authentication could also be performed by symmetric encryption. Since only Alice and Bob share the secret key, one would know if the incoming messages originated from its peer. However, with symmetric encryption it is not possible to prove to a third party the ownership to a message.

2.17.3 The Secure Sockets Layer protocol

The Secure Sockets Layer, SSL protocol (Freier *et al.* 1996) was developed by Netscape Communications Corporation and aims to provide communications privacy over the Internet. The protocol is designed for client/server applications in a way to prevent eavesdropping, tampering and message forgery. The protocol consists of two layers, namely the **SSL Record Protocol** and the **SSL Handshake Protocol**. The SSL

Record Protocol, is layered on the top of the network or TCP layer and is used for encapsulation of higher level protocols, such as the SSL Handshake Protocol. The latter allows the client and the server to authenticate to each other and exchange security relevant information, such as an encryption algorithm negotiation and cryptographic keys. The encryption algorithms supported by the SSL protocol are the DES, RSA, RC4 and DSS. Formally, the goals of the SSL were the specific four:

1. **cryptographic security**, i.e. to establish a secure connection between two parties,
2. **interoperability**, i.e. the SSL should describe the basic cryptographic requirements via specifications so that independent programmers could develop applications compatible with th SSL,
3. **extensibility**, i.e. allow import of new cryptographic techniques in the existing protocol, and
4. **relative efficiency**, which includes a session caching scheme in order to avoid repeating cryptographic operations of a previous session.

The SSL is application protocol independent, i.e. a higher level protocol enjoys transparency of the SSL layers. As a result, the SSL is proposed to co-operate with several applications including ftp, telnet, http. From an efficiency perspective, little priority has been allocated to performance. In contrast, the protocol's main goal is cryptographic security; cryptographic operations are highly CPU intensive and the network activity increases substantially due to the handshake orientation of the protocol.

Wagner and Schneier (1996) presented an analysis of the security of the latest version (3.0) of the SSL protocol. Version 3.0 of the SSL has considerably increased the security by correcting major problems which existed at its predecessor, SSL 2.0. According to Wagner and Schneier, the SSL provides a lot of known plaintext to an eavesdropper, but as long as the ciphers are secure, confidentiality would be offered. However, the SSL does not prevent traffic analysis. Since the SSL is mainly used in the Web, one would expect to have confidentiality when downloading web pages. This is not the case. Since the source and destination information is not encrypted, an eavesdropper may record the volume of data being passed between the client and the server. It is easy then to compare the amount of data transferred with the web pages on a server. The length of the URL request is also useful for revealing which particular pages a client would visit. In general, an attacker could determine which parties are communicating and what type of services are being used. If block ciphers are used, then it is expected that the encrypted information would be slightly larger than the plaintext, due to padding. In stream ciphers though, padding is not performed.

Concerning active eavesdropping, the SSL provides security, since it is resistant to certain attacks, such as the cut-and-paste and the short-block attack (Bellovin 1996). In addition, replay attacks are prevented, since the message authentication includes also an implicit sequence number. Yet, the security could be improved if a stronger MAC is used.

The messages for changing cipher specification are not protected by authentication. Consequently, an attacker may control and send forged messages.

The major security flaw though is in the key exchange, and more precisely, in the specification of the *server key exchange* message. The problem is that the key algorithm is not signed. Consequently, an attacker may choose the key exchange algorithm, and the security of the protocol would be equal to the security of the weakest algorithm used for the exchange of keys.

Finally, the SSL is criticised because it uses keys with short lengths. This is unavoidable and it is caused due to export restrictions.

2.17.4 S/MIME: Cryptographic security services for MIME

MIME (Multipurpose Internet Mail Extensions), is a popular Internet standard, used for exchanging various file formats by electronic mail (Freed & Borenstein 1996). The Secure MIME (S/MIME) specification provides the following cryptographic security services to applications supporting MIME (Dusse *et al.* 1998): authentication, message integrity, non-repudiation of origin and privacy and data security.

MIME is based on *mail user agents*, which are servers running on hosts and implement the mail related protocols to handle the mail messages. In this section the term *agent* is used instead of *party* which is used throughout the thesis.

This thesis is mainly interested in the process in the decision procedure on which encryption method is to be used between a sending and receiving agent. Each agent constructs a *capabilities list* which consists of the encryption ciphers it supports. This information is "announced", among other information, to agents who wish to communicate. Therefore, the decision process involves selection of a cipher from the capabilities list of the receiving agent.

The sending agent has to decide on which encryption is to be used depending on the secrecy of the data and the capabilities of the receiving agent. However, it is possible that the sending agent may not have the capabilities list of the receiving agent. In such case, the sending agent may use either a weak encryption (RC2 with 40-bit key) which its implementation is mandatory for all agents, or use a strong encryption such as triple DES, but is also willing to risk that the recipient may not be able to decrypt the message.

Consequently, although S/MIME may be algorithm independent and support very strong encryption, it cannot be applied in all instances and the sending agent has to take decisions on the tradeoff between level of security and risk of failed decryption.

2.18 Conclusions

The theoretical background has focused on symmetric cryptography and more specifically on symmetric product block ciphers, considering their design principles, measures of cryptographic strength, their vulnerabilities against cryptanalytic attacks, and their use in cryptographic protocols for secure communication.

To provide a vehicle for investigation of the construction and security of ciphers, two well known ciphers, DES and IDEA, were discussed in some detail. The DES was described for its historical value and contribution to the research community. Many cryptanalytic methods as well as theoretic findings were developed in an attempt to analyse the DES. Most of these methods have general applications to DES-like cryptosystems, i.e. ciphers which are based on Feistel networks. The IDEA cipher was presented because confusion and diffusion were considered in its analysis. Furthermore, the IDEA cipher was a cipher based on theoretical foundations demonstrating that high confusion and diffusion could be offered by combining operations which belong to different algebraic groups.

The following aspects of product block ciphers were reviewed:

- the cryptographic primitives and more specifically the Feistel transformations,
- the key schedule,
- the properties related to cryptographic composition, such as the avalanche effect and the group property,
- the cryptographic modes of operation,
- their strength against cryptanalytic attacks.

Feistel transformations are widely used in symmetric block ciphers and are the main components in product ciphers. Since the proposed project focuses on product ciphers, theoretical papers concerning Feistel Networks were reviewed and the results included in the survey. An influential paper published by Luby and Rackoff (1986) focused on analysing the DES in terms of provable security, clarifying the idea of a distinguisher in the context of cryptography.

The first attempts to provide a systematic approach to heterogeneous and unbalanced Feistel networks were by Schneier & Kelsey (1996), resulting in some useful definitions of heterogeneous and unbalanced Feistel networks. Schneier and Kelsey concluded explicitly there was need for further research in heterogeneous and unbalanced Feistel networks, and part of the work described in this thesis addresses this.

A product cipher consists of the round function(s) and the key schedule. The key schedule is the process which generates the subkeys from a master key. Every subkey is assigned to a round of the product cipher. Knudsen (1994b) defined the strength of a key schedule and proposed an effective method using one-way functions to generate an arbitrary number of subkeys from a strong key schedule.

Composition of ciphers can strengthen security, but can also introduce weaknesses. The avalanche effect (Pfleeger 1989) is a characteristic which is linked directly to the cryptographic strength of a product cipher. The avalanche effect states that the cryptographic composition of two ciphers (or two rounds) may result in a cipher which is much stronger than the two individual ciphers. On the other hand, the group property (Beker & Piper 1984) is an undesirable characteristic which can limit the effectiveness of the avalanche. It became apparent from the literature that establishing the existence of a group property in a cipher is not always a simple process (Campbell & Wiener 1993).

Concerning cryptographic modes of operation, two classes were distinguished in the literature, the standard modes of operation and the non-standard modes of operation. The standard cryptographic modes of operation were described for the DES, but are also used for any block cipher. The relevant papers (Biham 1993, 1996) revealed that the combination of standard modes of operation in general (referred as *multiple modes of operation*) may reduce the cryptographic strength of the construction: the overall strength of the cryptosystem would be equal to the strength of the least secure cipher used. This is because the weakest component can be attacked by the cryptanalyst planting differential values in some points in the structure of the cipher in order to mount a chosen ciphertext attack (Biham 1996). A chosen ciphertext is the strongest attack on a cryptosystem supporting multiple modes of operation (Schneier, 1996). Therefore when designing a system there is a need to identify whether an attacker can mount such an attack. In the case where a chosen ciphertext cannot be mounted, then multiple modes of operations are preferred because of its advantages when used for secret communication. Thus, the attack scenarios influence the design objectives of a cryptosystem.

The question of cryptographic strength is one of the central issues in the proposed cryptosystem. Knowledge of the attack methods and the requirements of each attack, enabled the identification of measures of cryptographic strength which are most appropriate to the work of this thesis.

Measures of strength described in the literature include confusion, diffusion, differential (Biham & Shamir 1991) and linear characteristics (Matsui 1994), statistical tests for randomness (Beker & Piper 1982), information theoretic and complexity measures (Dai & Yang 1991). The well known concepts of confusion and diffusion were selected to be measures of cryptographic strength in this thesis, as they could be applied to block ciphers. Differential and linear characteristics were not applicable, as the design aims set in the thesis require practical security to be offered to a system in which oracle-like attacks are not permitted, making differential and linear cryptanalysis infeasible.

The literature survey concludes with a brief review on security protocols which are used to offer security in computer networks, as the proposed cryptosystem is intended to be used in communication over computer networks.

To establish secure communication, a security protocol is required. A security protocol usually consists of a setup session in which security related information is exchanged, for example keys, their lifetime, authentication information, and a session for the information communication. Conventional protocols were briefly reviewed (RSA, Diffie-Hellman key exchange, Station to station). Key exchange protocols were not part of this research but were required for the prototype.

The literature has extensive description of the use of a single publicly known cipher in security protocols. More recently alternatives have been proposed, such as to include more than one cipher in a protocol, with the communicating parties selecting privately one cipher from the list of available ones (Kaliski 1998). Furthermore, a class of cipher independent protocols are developed and are used in the Internet for Web or e-mail transactions (such as the SSL protocol and S/MIME respectively); the advantages are described in the literature.

The proposed project is based on the idea of a security protocol supporting multiple ciphers. The proposed project uses the RSA protocol (Rivest *et al.*1978) as part of the setup session, but with the addition of the symmetric composed cipher as extra secret information.

Part III

Proposed Method

Chapter 3

Description and Analysis of the Proposed Method

3.1 Introduction

This project describes a method to enable two communicating parties to negotiate a private symmetric block cipher to be used for a private communication session. The method requires evaluation of the strength of the cipher, and that it should be completed in an acceptable time for setting up communication.

This chapter is dedicated to the description and analysis of the proposed method for secure cryptographic communication in a client/server oriented environment. The proposed method is concerned with communications between a client and a server, through an untrusted network. The host and operating system are assumed secure in this project. Although not part of the scope of the work of this thesis, the security framework is outlined and the requirements of the underlying system are described for completeness. Assuming that the specific requirements are met, the cryptosystem should perform as expected.

The design objectives of the cryptosystem are specified, followed by a proposal for implementing a cryptosystem which fulfils these objectives.

Finally, some conclusions regarding the security of the cryptosystem are given.

3.2 Security framework

The proposed cryptosystem was designed to operate in environments where two trusted hosts communicate via an untrusted network[1], to perform a secure session. Two alternatives could be accepted:

- only the host is trusted, i.e. communications within the LAN are susceptible to eavesdropping,
- the LAN is trusted.

Both scenarios are acceptable in the cryptosystem's context. For the first scenario, it should be assumed that an attacker - which would more likely be another unauthorised but legitimate user - has access to the host, but would not be able to eavesdrop the communication sessions of other users in the same host. This also raises more considerations if a certain graphic user interface is included, since windows environments cause the user to be more remote from the operating system, resulting to a smaller degree of control. It should be assumed that security is offered in the windows environment, in a sense that the key strokes or windows could not be scanned.

3.3 Design objectives of the proposed cryptosystem

The proposed cryptosystem aims to develop an effective method for constructing a number of ciphers, where every cipher instantiation may be evaluated in terms of its cryptographic strength. To meet this aim, the design objectives of the proposed cryptosystem are as follows:

- practical security, which consists of providing evidence that the ciphers are secure against known plaintext attacks
- ability to integrate into cryptographic algorithm independent security protocols

[1]However, according to the *DoD Trusted Computer System Evaluation Criteria* (DoD 1985), a computer could not be considered as a trusted one, if it is connected to a network.

- permit evolution of cryptographic technology
- allow standardisation
- effective control of complexity over speed

The first design objective aims to make the cryptosystem suitable for different types of communications requirements. For example, in video-conferencing applications, large time overheads are not accepted, but usually video does not require a high degree of security. Often the accompanying audio contains more information than the video, and so may require a more secure cipher, so higher complexity could be used in encryption. For sensitive data, it is preferable to use ciphers with components of standard algorithms, which are known to be more likely to be secure.

Practical security is desirable for all ciphers. Depending on the type of application, the opportunities and the computational requirements of the attacker are considered in assessing the strength of the cipher. In the specific context, it is assumed that the attacker has complete access to the communicated ciphertext and to some plaintext/ciphertext pairs. The ciphertext space is public and therefore the attacker has knowledge of all cipher instances, but does not know which specific instance is used in each communication session.

In the context of this thesis, a non standard cryptographic algorithm is an algorithm which is well known and widely accepted. Since non standard cryptographic algorithms are used for the cryptosystem, strength against chosen plaintext or ciphertext attack would be a difficult requirement and is the first item proposed for further work. Although a number of principles and conditions for provably secure cryptosystems have been described in the literature, it is generally accepted by authors in the field that in most cases it is hard to apply these principles and produce practical secure ciphers. In terms of practical security, the proposed method aims to distinguish potentially secure from proven insecure ciphers, provided that an attacker is not in a position to mount a chosen ciphertext attack, although more stringent rules may lead to ciphers which maintain their strength against such attack. However this proposition should be taken with a high degree of caution and it is strongly suggested in this thesis that the cryptosystem should be extensively analysed. Because the ciphers used are non-standard ciphers (although built from standard ciphers), they should not be embedded in protocols which allow unlimited-queries-oracle-like attack scenarios.

History has proven that totally secret algorithms were not a good approach, because they had not been extensively analysed; instead, they were broken by reverse engineering. Furthermore, there is an international interest in establishing security standards, such as quality standards (eg. ISO 900X). Security standards are inherent requirement for organisations who wish to communicate in an internetworking environment. Companies can be confident in secure communication with other companies who conform to accepted security standards. In contrast, a company's private algorithm would be of unknown strength, and furthermore it may have trapdoors or Trojan horses for espionage purposes. The proposed cryptosystem allows evolution of ciphers, since it is open for enhancing or updating the set of encryption steps, and also any party could assess the strength of another party's ciphers.

In order to meet the fourth objective, standardisation is the main leverage in modern communications since all communication devices need standards to operate with success. In cryptography though there is a debate on the need for, and feasibility of a cryptographic standard. Although Devargas (1993) suggests that an encryption algorithm need not be standardised, because of the uncertainty regarding its strength in the future, standardisation is unavoidable in cryptography used in communications. It also provides assurance to all parties.

According to Kaliski (1998) most protocols defined recently are algorithm independent supporting a variety of encryption algorithms, signature algorithms, and key management techniques. The S/MIME secure mail protocol being developed by the Internet Engineering Task Force (IETF) is likely to take this form. As another example, RSA Laboratories' revision of PKCS #7 (RSA 1993), currently in progress, will be algorithm-independent. Algorithm independence is helpful because algorithms have a variety of properties, including speed, security, implementation complexity, and patent coverage. A protocol that gives flexibility in algorithm choice will simplify overall system design[2] and accommodate a wide range of applications. Of course, interoperability is a problem if people choose different algorithms, so in many cases a "mandatory" algorithm choice is specified which all implementations must support.

3.4 Description of the proposed cryptosystem

3.4.1 Outline of the cryptosystem

A conventional symmetric algorithm could be represented as in Figure 3.1. The algorithm is considered to consist of the encryption steps, the structure, and the key. For a conventional algorithm, the encryption steps are public and so is the structure of the algorithm, whereas the key is secret. It is public knowledge in the case of the DES for example, that a one-way keyed function which involves substitutions and permutations, is applied 16 times to form the structure of the DES algorithm.

public		secret
encryption steps	structure	key

Figure 3.1: A conventional symmetric cryptographic algorithm

It is possible for the entire cryptographic algorithm to be private, in the sense that it is known in a limited domain - for example within an organisation. This is illustrated in Figure 3.2.

For wider communication the "fully" private algorithm is inefficient and vulnerable, especially between parties who have not communicated previously. Moreover, there is no plausible reason to accept and employ someone else's code, with unknown properties.

[2]Kaliski is referring to the systems analysis level rather than implementation.

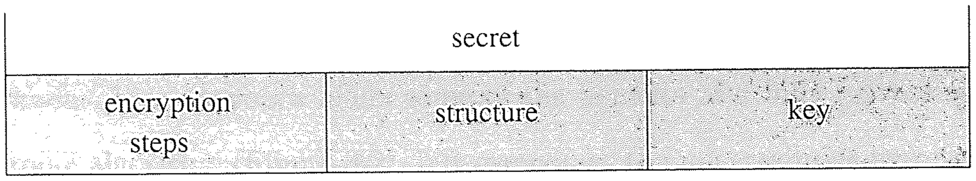


Figure 3.2: A private symmetric algorithm

A novel mid-way approach is to use well known encryption steps, but to keep the structure of their combination secret, as shown in Figure 3.3.

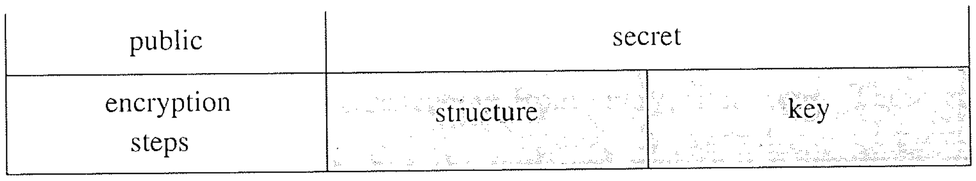


Figure 3.3: The secret structure and key symmetric algorithm

Encryption and decryption speeds depend on the algorithm complexity. In order to enhance the security of a system, larger keys may be introduced, which in most cases result in increased complexity and therefore increased computational costs. For example, use of the triple DES scheme, employing two keys, results in an algorithm which consists of 16x3=48 rounds of the DES primitive. However, in applications where computational time is non-negligible, such an approach is not feasible in terms of computational time. Cryptographic algorithms are CPU intensive (Koblitz 1987), the complexity of these algorithms can add large time overheads, reducing the overall throughput of a communication channel. In many applications there is need for higher speed with lower security, for example in the case of video-conferencing mentioned earlier. Current approaches in using algorithms favour a fixed composition and consequently a fixed complexity.

To provide higher security than is available using a conventional algorithm (Figure 3.1) there is a growing practice in the design of security protocols, of selecting one algorithm from a number of available algorithms for each session (Freirer *et al.* 1996; Atkinson 1995a). Conventionally the choice is one from four algorithms. This linearly increases the search space for an intruder, who must first guess the algorithm used, then mount a conventional attack. There is a 25% probability that the correct algorithm will be chosen. This approach offers some of the benefits of a fully private symmetric cryptographic algorithm (Figure 3.2), but overcomes the uncertainties by retaining use of publicly known algorithm.

This project proposes to use the secret structure and key symmetric algorithm introduced in Figure 3.3 to significantly reduce the probability of an intruder guessing the correct algorithm.

The proposed cryptosystem selects from a potentially large number of different algorithms. The algorithms are constructed from encryption steps. These encryption steps may be complex or simple, but the preferred underlying primitives are simple for fast computation. The encryption steps should be public knowledge, whilst the structure is secret.

3.4.2 Description of the cryptosystem

The cryptosystem consists of the following parts:

- the cryptographic algorithm generator (CAG), and
- the cryptographic algorithm negotiation (CAN)

Structure of the cryptographic algorithm

In the proposed method, a number of encryption steps and other classes of functions - simple modular additions, multiplications, hash functions, etc. - are interconnected to produce a final encryption function. The construction is negotiated between the two parties to establish the cryptographic algorithm to be used in one session. The knowledge of the encryption configuration together with the key allows the decryption of the message. The two parties who wish to communicate securely could publicly agree on the set of the encryption steps, but the way these steps are structured to form the encryption (and decryption) algorithm is secret.

The cryptographic algorithm is implemented by the following components:

- Blocks,

- Summing units.

Blocks and summing units can be interconnected in series/parallel combinations.

Blocks and summing units are linked by data flows. Summing units are placed whenever the encryption block is supported by one or more feedback blocks. The feedback blocks operate on the output of the underlying encryption block and their outputs are summed to be fed to the input of the encryption block. It should be noted that the summing units perform a bit-wise **XOR** operation.

The data flows are made up of (message) data, and initialisation values and are 64–bit blocks.

The block diagrams of the encryption and decryption functions are presented in Figure 3.4 and Figure 3.5 respectively. It should be noted that the difference between the encryption and decryption functions is the bottom row, where the encryption functions, E_x are replaced by the decryption functions, D_x. The rest of the functions, F_x could also include encryption functions, but they should not be replaced by their decryption functions at the decryption algorithm. Another difference is that the feedback structure of the encrypting algorithm is replaced by the feedforward in the case of the decryption algorithm.

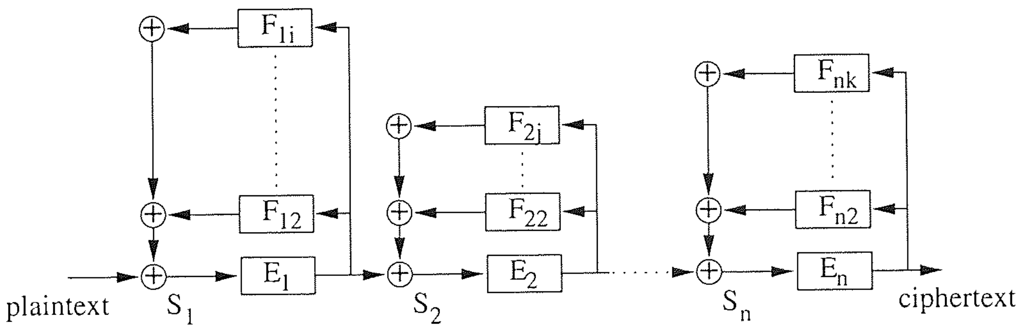


Figure 3.4: Block diagram of the encryption algorithm

The encryption steps may consist of one-way functions, substitutions, permutations, random number generators or one round Feistel ciphers - such as a DES step. One-way functions play a important role to the algorithm's cryptographic strength. From a

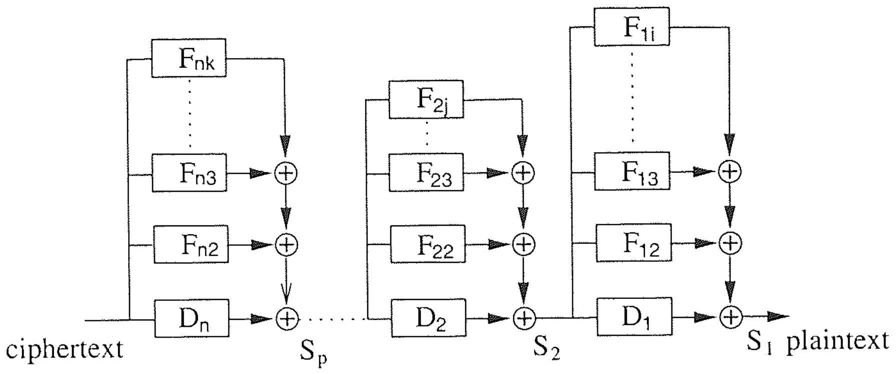


Figure 3.5: Block diagram of the decryption algorithm

cryptographer's perspective, one way functions have been shown to be necessary and sufficient for many cryptographic primitives (Chapter 2.4 of the literature review).

A set of substitution boxes (S-boxes) will also be included to. As highlighted by Seberry *et al.* (1994), S-boxes are likely to be vulnerable to differential cryptanalysis. However, current state-of-the art has shown that S-boxes could be designed to be resistant to such attacks (Heys & Tavares 1996).

The Feistel ciphers (Luby & Rackoff 1998) can be found in several encryption algorithms and are the most popular configurations in block product ciphers, as described in Chapter 2.4 of the literature review.

All of the above encryption are finally cryptographically composed in a random manner, in order to produce the final encryption algorithm. The latter could be considered as a series of composed algorithms, where each one of them is operating in a non-standard chaining mode (Davis & Price 1984). The difference is that more than one feedback operations may exist.

Generating the encryption/ decryption algorithm

The purpose for the following description of the generation of the encryption and decryption algorithm is to develop an approach in which the cipher instance is communicated between two parties.

As stated above, the cryptographic algorithm is a random combination of encryption and feedback blocks. The combined algorithm is generated as follows:

1. A random number of columns (layers) is selected between a lower and an upper value. The lower value ensures that feedback blocks are included if required so, whereas the upper value is a rough indication of the maximum costs.

2. For each layer, a random height is selected, similar to (1), i.e. between a lower and an upper value. The height specifies the number of feedback blocks(functions) at each layer.

3. Given $\mathcal{F}_m = \{f_1, f_2, ..., f_m\}$ a set of transformations, and $\mathcal{E}_n = \{e_1, e_2, ..., e_n\}$ a set of encryption steps, the blocks can be assigned with the elements of these sets. More specifically, the blocks of the bottom row of the cryptographic algorithm (Figure 3.4) are assigned with the cryptographic functions from $\mathcal{E}_n$ which are selected randomly, whereas the rest of the blocks could include randomly selected elements from both sets $\mathcal{F}_m$ and $\mathcal{E}_n$.

This encryption structure allows the reverse transformation of the data. The decryption algorithm is the encryption algorithm mirrored with the encryption functions on the bottom row replaced with their respective decryption functions. More formally:

1. The encryption algorithm's block structure is mirrored, so all feedbacks change to feedforwards.

2. If $\mathcal{D}_n = \{d_1, d_2, ..., d_n\}$ is the set of decryption functions with respect to $\mathcal{E}_n$, then each e_i block at the bottom row becomes d_i.

The input to the encryption and decryption algorithms (Figures 3.4 and 3.5) consists of the plaintext or ciphertext data and initialisation values. The F blocks, could be a transformation of the data, or a pseudorandom number generator (for operation as a stream cipher). Transformation of the data may involve hash functions, logical shifts, permutations. The E (and D) blocks may range from permutations and substitutions, to single rounds of Feistel ciphers. The F functions may also include the E and D

functions and could be one-way. However, the E functions must be reversible, i.e. for each E_r function, a D_r must exist.

Once the algorithm has been constructed, it would be handed over to the security protocol, which is responsible for its transmission.

Formal description of the algorithm

Let $K \epsilon GF(2)^{64}$ a 64-bit key which is used to encrypt a $(64 \times t)$-length plaintext $p_1p_2\dots p_t$ to its corresponding ciphertext $c_1c_2\dots c_t$. The initial values of the sums are derived from Figure 3.3:

$$S_{10} = \sum_{m=1}^{i} f_{1m}(E_1(K))$$

$$S_{20} = \sum_{m=1}^{j} f_{2m}(E_2(E_1(K)))$$

$$\vdots$$

$$S_{n0} = \sum_{m=1}^{k} f_{nm}(E_n(E_{n-1}(\dots(E_1(K))\dots)))$$

where $\sum$ denotes the bit-wise modulo 2 addition (**XOR**).

Some of the functions f may also have as input a portion of the key K. In the actual implementation this infers that the key is a global variable where all functions may access it. Some functions may access the key only during the initialisation round, whereas some others may access it at each round.

The second round involves encryption of the first block of the message:

$$c_1 = E_n(S_{n0} \oplus E_{n-1}(\dots \oplus E_2(S_{20} \oplus E_1(S_{10} \oplus p_1))\dots))$$

and the sums of the second round are computed:

$$S_{11} = \sum_{m=1}^{i} f_{1m}(E_1(S_{10} \oplus p_1))$$

$$S_{21} = \sum_{m=1}^{j} f_{2m}(E_2(S_{20} \oplus E_1(S_{10} \oplus p_1)))$$

$$\vdots$$

$$S_{n1} = \sum_{m=1}^{k} f_{nm}(E_n(S_{n0} \oplus E_{n-1}(\ldots(E_1(S_{10} \oplus p_1))\ldots)))$$

In general, the encryption of the j-th block of the plaintext is obtained by:

$$C_j = E_n(S_{n(j-1)} \oplus E_{n-1}(\cdots \oplus E_2(S_{2(j-1)} \oplus E_1(S_{1(j-1)} \oplus p_j))\ldots))$$

where:

$$S_{1(j-1)} = \sum_{m=1}^{i} f_{1m}(E_1(S_{1(j-2)} \oplus p_{j-1}))$$

$$S_{2(j-1)} = \sum_{m=1}^{j} f_{2m}(E_2(S_{2(j-2)} \oplus E_1(S_{10} \oplus p_{j-1})))$$

$$\vdots$$

$$S_{n(j-1)} = \sum_{m=1}^{k} f_{nm}(E_n(S_{n0} \oplus E_{n-1}(\ldots(E_1(S_{10} \oplus p_{j-1}))\ldots)))$$

Decryption is performed in a similar way.

Representation of the composed ciphers

The encryption and decryption algorithms were described by two matrices, where all elements apart from the bottom row are identical. At the bottom row in particular, the elements of the one matrix are the cryptographic complements of the other. Due to the various sizes of the columns, a $n \times m$ size matrix was selected, where n is the maximum of the columns, and the elements which are not used were set to zero. The representation of the algorithms with these two matrices was completed by indexing each transform and storing the index value in the respective element in the matrix. It was the cryptosystem's task to interpret the indexes and construct the encryption and decryption algorithms from such information. This is described in Chapter 4.

3.5 Analysis and definition of measures

The construction of the cryptographic algorithms is a twofold problem. First, the combination of the encryption steps alone without feedback must offer practical security. Second, the feedback blocks must not cancel any cryptographic transformation of the primitives they are supporting.

There has been extensive theoretic work in product ciphers and especially in iterative Feistel transformations. However, there has not been much work in combination of different cryptographic primitives and in addition there are several open problems in unbalanced Feistel networks (Schneier & Kelsey 1996). Moreover there is no formal description for combination of primitives. In this section some theoretic requirements which support the proposed method will be presented. These requirements together with experimental results in Chapter 5 lead to the novel concept of the **Cryptographic Block Profile, CBP** which describes the properties of the underlying cryptographic block, such as topology, potential cryptographic strength and so on. The theoretic and experimental rules for filtering out subsets of weak instantiations are described in terms of the CBPs of the blocks.

3.5.1 The total search space

The calculation of the search space (secondary Aim B.) gives an insight to the order of magnitude of the search space.

The total search space is the product of the key space and the algorithm space:

$S_{total} = <k> * <a>$.

Since a 64-bit key is used, the key space is: $<k> = 2^{64}$. Let:

n the number of encryption steps,

m the number of feedback functions.

Since the encryption steps are allowed to be placed in the feedback blocks, the total number of feedback functions would be $n + m$.

Assume an algorithm which consists of one stage. Each encryption step - which becomes an encryption block once placed in the structure of the composed cipher - is combined with any $0, 1, ..., n + m$ feedback blocks. The latter yields a space of:

$$\binom{n+m}{0} + \binom{n+m}{1} + \ldots + \binom{n+m}{n+m}$$

An equivalent form can be derived by using Newton's binomial equation:

$$(a+b)^v = \binom{v}{0}a^v + \binom{v}{1}a^{v-1}b + \ldots + \binom{v}{v-1}ab^{v-1} + \binom{v}{v}b^v$$

where a, b, v are integers. For $a = b = 1$ we have:

$$2^v = \binom{v}{0} + \binom{v}{1} + \ldots + \binom{v}{v}$$

Therefore the algorithm space for all one-stage algorithms would be:

$$S_{A1} = n\left[\binom{n+m}{0} + \binom{n+m}{1} + \ldots + \binom{n+m}{n+m}\right] = n2^{n+m}$$

and its total search space (i.e. combined with the key) is:

$$S_1 = 2^{64} n 2^{n+m}$$

For an algorithm of l stages, the algorithm space is:

$$S_{Al} = [n2^{n+m}]^l$$

The total space is the sum of all possible algorithms from 1 to l stages:

$$\begin{aligned} S_{total} &= S_1 + S_2 + \ldots + S_l = <k>\ n2^{n+m} + <k>\ [n2^{n+m}]^2 + \ldots + <k>\ [n2^{n+m}]^l \\ &= <k>\ \underbrace{([n2^{n+m}] + [n2^{n+m}]^2 + \ldots + [n2^{n+m}]^l)}_{geometric\ series} \\ &= <k>\ (n2^{n+m}\frac{1-[n2^{n+m}]^l}{1-n2^{m+n}}) \end{aligned}$$

In our particular case we have:

$$1 \ll n2^{m+n}$$

Therefore:

$$S_{total} =<k> (n2^{m+n})^l$$

However, this search space includes weak instantiations, such as ciphers which consist of transpositions and/or a small number of Feistel rounds, not offering full diffusion and maximum confusion. In the rest of this Chapter there is some analysis in order to:

- identify weak structures so they can be excluded, or
- establish rules where only potentially strong ciphers could by generated.

3.5.2 Feistel Networks

As presented at Chapter 2.4, Luby and Rackoff (1988) demonstrated that for a balanced Feistel network three rounds are needed so that the distinguishing probability of the Feistel transformation from a random one is low.

With the introduction of differential cryptanalysis, the security of a cryptosystem was described by the probability of its differential characteristic, i.e. certain plaintext/ciphertext differences which occur with non-negligible probabilities. The higher the probability of a differential characteristic, the higher the probability of breaking the cryptosystem. In general though finding a differential characteristic with high probability requires extensive analysis of the round function of the cipher.

Definitions and terminology

The definitions and terminology mainly focus on Feistel transformations. The definitions and terminology from the literature are presented, extended by the definitions and notions proposed in this thesis. Some definitions are not directly used in the proposed method, but aid on deriving both proposed and referenced ones.

The definitions and lemmas in this section are adopted from Schneier and Kelsey (1996). Let n be the length of the input and output block (in bits) of a Feistel network. The round function F of a conventional Feistel network would then be expressed as:

$$F : \{0,1\}^{n/2} \times \{0,1\}^{k} \rightarrow \{0,1\}^{n/2}$$

where k is the length of the subkey in bits.

Definition 1. A Feistel transformation would be defined as:

$$X_{i+1} = (F_{k_i}(msb_{n/2}(X_i) \oplus lsb_{n/2}(X_i)) \bullet msb_{n/2}(X_i)$$

where X_i is the input to round, X_{i+1} is the output of the round, k_i is the subkey used in the round and $lsb_u(x)$ and $msb_u(x)$ select the least significant and most significant u bits of x respectively.

Definition 2. A j-round balanced Feistel network is defined as:

$$X_{i+1} = (F_{k_i}(msb_{n/2}(X_i) \oplus lsb_{n/2}(X_i)) \bullet msb_{n/2}(X_i)$$

for $0 \leq i \leq j-1$. X_0 would be the plaintext, and X_j would be the corresponding ciphertext.

An *Unbalanced Feistel Network, UFN* is a Feistel network where the "left half" and "right half" are not of equal size.

Definition 3. A one round of an $s : t$ UFN is defined as:

$$X_{i+1} = (F_{k_i}(msb_s(X_i) \oplus lsb_t(X_i)) \bullet msb_s(X_i)$$

where $msb_s(X_i)$ is the *source block* and $lsb_t(X_i)$ is the *target block*. If $s > t$, the UFN is called *source heavy*, whereas in the opposite case it is called *target heavy*.

Definition 4. A UFN is *homogeneous* if the round function is identical in each round, except for the round keys. A UFN is *heterogeneous* if the round function is not always identical for different rounds, except the round keys.

Definition 5. A UFN is *complete* when in each round every bit is part either of the source block or the target block, i.e. $s+t=n$. If $s+t<n$, then the UFN is *incomplete* and the unparticipating block $z=n-s-t$ is the *null block*.

Definition 6. A UFN is *consistent* when s,t and z remain constant for the entire cipher. In the opposite case, the UFN is *inconsistent*.

Definition 7. A *cycle* is the number of rounds for each bit in a block to appear in both source and target at least once.

Lemma 8. A cycle C of an $s:t$ UFN is

$$C=\lceil\frac{n}{\min(s,t)}\rceil \tag{3.1}$$

Definition 9. A *rotation* is the number of rounds needed for a bit to return to its starting position.

Lemma 10. A rotation G of an $s:t$ UFN is

$$G=\frac{n}{\gcd(s,t)} \tag{3.2}$$

Definition 11. A UFN is *even* if $C=G$, *odd* if $C\neq G$ and *prime* if $G=n$.

Definition 12. The *rate of confusion,* R_c of a consistent UFN is the minimum number of times per cycle that any bit can occur in the target block.

Lemma 13. For an $s:t$ UFN, the rate of confusion is:

$$R_c\leq\frac{t}{n}$$

Definition 14. The *rate of diffusion,* R_d is the minimum number of times per cycle that a given bit can have the chance to affect other bits.

Lemma 15. For an $s : t$ UFN, the rate of diffusion is:

$$R_d \leq \frac{s}{n}$$

Schneier and Kelsey (1996) argue that the effectiveness of linear and differential cryptanalysis is related to the rate of confusion and diffusion respectively. More specific, if p the bias of the best possible linear approximation in a target block in one round, it was demonstrated that the output bias of any nontrivial characteristic after C rounds would be at most $2^{CR_c-1}p^{CR_c}$. This relation holds for even complete UFNs with cycle C.

Concerning differential cryptanalysis, an increase in the diffusion rate may result in higher resistance to differential cryptanalysis. Similarly, in the case of even UFNs, if p is a nontrivial one round differential characteristic, the probability of its propagation through C rounds, would be at most p^{CR_d}.

In practice though, it would be useful to have indication of the actual values of the confusion and diffusion rate. The *confusion and diffusion matrices*, presented at sections 3.5.4 and 3.5.5 respectively, could be used to determine the rates, as well as the resulting rates of the overall ciphers.

Number of rounds

Most of the work in the literature concerning the required number of rounds in Feistel networks is directed to balanced Feistel networks. That is because the most popular ciphers - such as the DES - are balanced, so the questions regarding cryptographic strength addressed those ciphers.

Luby and Rackoff (1988) suggested that the minimal number of rounds for a balanced Feistel network is three. However, in the case of the DES this is not true. The reason is that three rounds would be enough given that the round function has certain properties. The minimal requirement would be that after three rounds, every input but should have the chance to affect every output bit. In the case of the DES this is not true.

Yet, Aiello and Venkatesan (1996) developed a chosen plaintext attack with only

$O(2^{n/2})$ chosen plaintexts to distinguish between a four round Feistel network and a random permutation on messages in $\{0,1\}^{2n}$. Furthermore, Coppersmith (1996) developed a similar attack with $O(2^n)$ chosen plaintexts which may recover the actual contents of the round function with high probability. Unless an adversary is not in a position to mount a chosen plaintext attack, four rounds of a Feistel Network is not enough.

One should expect that for UFN, more rounds would be needed. If the UFN is target heavy (Def. 3), this implies that there are linear relations between some input bits and some output bits on each round. If the UFN is source heavy (Def. 3), more rounds would be needed, so that every bit will appear in the target block.

3.5.3 The key schedule

Knudsen (1994a) pointed out the importance on the design of the key schedule and demonstrated how in some ciphers a strong key schedule may improve the security considerably. The main result is that a key schedule is considered strong if given a subkey, one could not obtain information about the master key in a polynomial time. Such a property suggests that the implementation of a key schedule should involve one way functions (Damgård & Knudsen 1996). More precisely, since in a key schedule a master key is involved, the underlying one way functions should be keyed, or alternatively a standard block cipher could be used. It is important for the latter to be a standard publicised cipher, in order to be more assured that the only attack is exhaustive search.

The following key schedule procedure is adopted by Damgård and Knudsen (1996). Let $H_k()$ denote a one way keyed hash function, or a standard block cipher, operating with the master key k. The key schedule $K_s = \{k_1, k_2, \ldots, k_l\}$ for an l-round cipher is defined as:

$$k_i = H_k(C + i),\ 1 \leq i \leq l$$

where C is some constant.

3.5.4 The confusion matrix

The confusion matrix proposed in this thesis is a matrix which summarises the probabilistic relation of every input with every output bit. The confusion matrix is used to calculate the confusion of a cipher.

The confusion matrix is constructed by the method described in section 2.13.4 in Chapter 2.9. Given a random plaintext $p_0 \in_U GF(2)^n$, $n-1$ plaintexts are generated, such that the Hamming distance between p_i and p_0 for $1 \leq i \leq n$ is one and $p_i \neq p_j$ for all $i \neq j$. Furthermore, $i-1$ denotes the position where the difference occurs.

For example, if $n = 4$ and $p_0 = 1101$, p_i, $0 \leq i \leq n$ would be:

i	p_i	$p_0 \oplus p_i$
0	1101	0000
1	0101	1000
2	1001	0100
3	1111	0010
4	1100	0001

Next, the corresponding ciphertexts are obtained:

$$c_i = E_k(p_i),\ 0 \leq i \leq n$$

and the **XOR**s $\psi_j = c_j \oplus c_0$, $1 \leq j \leq n$ are also computed. If $a[k]$ denotes the k-th bit of the binary string a, then matrix Ψ is defined as:

$$\Psi = \begin{bmatrix} \psi_1[0] & \psi_1[1] & \ldots & \psi_1[n-1] \\ \psi_2[0] & \psi_2[1] & \ldots & \psi_2[n-1] \\ \vdots & \vdots & \vdots & \vdots \\ \psi_n[0] & \psi_n[1] & \ldots & \psi_n[n-1] \end{bmatrix}$$

Definition 16. If the above process is run L times for distinct random p_0's, and the corresponding Ψ matrices are $\Psi_1, \Psi_2, \ldots, \Psi_L$, the *confusion matrix* C is defined as:

$$C = \frac{1}{L} \times (\Psi_1 + \Psi_2 + \ldots + \Psi_L)$$

where all operations are performed in the domain of rational numbers.

The confusion matrix describes the probability of an output bit changing, given that an input bit changes, for all pairs of input and output bits. The actual confusion matrix should require that all possible input values are tested, which is equal to $n2^{n-1}$ distinct pairs of Hamming distance one. For 64-bit messages, the number of distinct pairs is $64 \times 2^{63} = 2^{69}$. Hence, in practice the computation of the actual C is infeasible, so it is assumed that the confusion matrix approaches the actual values while L increases.

It would be desirable for a secure cipher to have a confusion matrix with all entries equal to 0.5. This would mean that a change of any one bit in the input, would have a 50% change of the output bits.

Definition 17. A confusion matrix where all entries are equal to 0.5, is a *perfect confusion matrix.*

However, it is also required that not only the output bits have a 50% change probability, but the changes between the output bits should not be related, i.e.

$$Pr(\psi_j[i] | \psi_j[k], k \neq i, 1 \leq k \leq n) = 0.5,\ 1 \leq j \leq n$$

This property is not examined by the confusion matrix. For instance, in a perfect confusion matrix, output bit $c[i]$ and output bit $c[j]$ have a 0.5 probability of change for any one-bit change in the input, but one of the following could also be true:

- $c[i] = c[j]$
- $c[i] = \overline{c[j]}$

The Ψ matrices though maintain enough information for examining the distribution of the changes. One test could be constructed as follows. As more Ψ matrices contribute to the confusion matrix, some or all entries should start converging to 0.5. If two or more entries $\psi_a[i], \psi_b[j],\ a \neq b, i \neq j$ exist such that $\psi_a[i] = \psi_b[j]$ or $\psi_a[i] = 1 - \psi_b[j]$ for $L = 2, 3, \ldots$, then there is a linear relation between these two entries. This is the proposed Ψ matrix depth test which is described later on (section 3.5.7).

The random distribution of ones within every Ψ matrix should also be assessed. More precisely, autocorrelation tests on the Ψ matrices investigate whether the changes are distributed randomly between the input and the output. The process is described later on.

Following the definition of the confusion matrix, we have the following lemma:

Lemma 18. There exists no Feistel network with less than three rounds and a perfect confusion matrix.

The proof follows from the main lemma of Luby & Rackoff (section 2.8.1).

It would be useful to investigate the conditions under the combination of the cryptographic blocks may result in - or approach - perfect confusion matrices. The diffusion matrix presented below, can provide such information.

3.5.5 The diffusion matrix

In this section the diffusion matrix is proposed. The diffusion matrix is used for calculating the diffusion of a cipher.

The $\beta(\cdot) : N \to \{0,1\}$ operator is defined as:

$$\beta(n) = \begin{cases} 1, & \text{if } n \neq 0; \\ 0, & \text{if } n = 0. \end{cases}$$

Definition 19. The *diffusion matrix* is defined as:

$$\mathcal{D} = \beta(\mathcal{C})$$

The diffusion matrix shows if a pairwise relation exists between input and output bits. That is, if a change of a particular input bit has the chance to affect a particular output bit. The diffusion matrix is very helpful, because it has the following property:

Lemma 20. Let C a product cipher with j encryption steps. The diffusion matrix of the cryptosystem is equal to:

$$\mathcal{D}_C = \beta(\mathcal{D}_1 \cdot \mathcal{D}_2 \cdot \ldots \cdot \mathcal{D}_j)$$

where $\mathcal{D}_i$ is the diffusion matrix of the ith encryption step.

Proof. The case for a two round product cipher is shown, that is $\mathcal{D} = \beta(\mathcal{D}_1 \cdot \mathcal{D}_2)$. Let $[\cdot]$ be a boolean evaluation, which evaluates the expression within the brackets to one if it is true and to zero is it is false, such as [p is prime]. The elements of $\mathcal{D}$, $\mathcal{D}_1$ and $\mathcal{D}_2$ are denoted by δ_{ij}, δ'_{ij} and δ''_{ij} respectively. Note that the output of round one is equal to the input of round two. For the first leftmost input bit it is:

$$[\text{input bit 1 is related with round-1 output bit } j] = \delta'_{1j},\ 1 \leq j \leq n \tag{3.3}$$

from the definition of the diffusion matrix. Similarly, for the first leftmost output bit:

$$[\text{output bit 1 is related with round-2 input bit } j] = \delta''_{j1},\ 1 \leq j \leq n. \tag{3.4}$$

Combining (3.3) and (3.4) we obtain:

$$[\text{input bit 1 is related with output bit 1}] = \delta'_{11} \cdot \delta''_{11} + \delta'_{12} \cdot \delta''_{21} + \ldots + \delta'_{1n} \cdot \delta''_{n1}$$

where the right-hand-side is a boolean expression, i.e. $. + .$ denotes the boolean **OR** and $. \cdot .$ denotes the boolean **AND**. If this is repeated for all input and output bits it gives:

$$[\text{input } i \text{ is related with output } j] = \delta_{ij} = \delta'_{i1} \cdot \delta''_{1j} + \delta'_{i2} \cdot \delta''_{2j} + \ldots + \delta'_{in} \cdot \delta''_{nj},\ 1 \leq i,j \leq n$$

or equivalently,

$$\mathcal{D} = \beta(\mathcal{D}_1 \cdot \mathcal{D}_2).$$

Q.E.D.

Application in cryptographic blocks

Balanced Feistel networks. Let $\mathbf{0_t}$ denote the $t \times t$ zero matrix, $\mathbf{I_t}$ the $t \times t$ identity matrix and $\mathbf{A_t}(\rho)$ a $t \times t$ matrix with $\rho \times 100\%$ ones and $100 - \rho \times 100\%$ zeros. A diffusion matrix for a one round balanced Feistel network where the round function operates on the left input sub-block would then be:

$$\mathcal{D} = \begin{bmatrix} \mathbf{0}_{n/2} & \mathbf{I}_{n/2} \\ \mathbf{I}_{n/2} & \mathbf{A}_{n/2}(\rho) \end{bmatrix}$$

The larger the ρ in $\mathbf{A}_{n/2}(\rho)$, the greater diffusion the round function offers. If the round function can relate all of its input bits, to all of its output bits, then it offers complete diffusion for the round, and the diffusion matrix would be:

$$\mathcal{D} = \begin{bmatrix} \mathbf{0}_{n/2} & \mathbf{I}_{n/2} \\ \mathbf{I}_{n/2} & \mathbf{A}_{n/2}(1) \end{bmatrix}$$

Note that the diffusion of the round function is different from the diffusion of the cipher and it concerns the diffusion between the input block to the round function and the output block of that function, which are respectively the source and target sub-blocks. The diffusion of the round function is the submatrix $\mathbf{A}_{n/2}(\rho)$. It is desirable to have round functions with complete diffusion, because in practice less rounds would be needed to achieve complete diffusion of the cipher.

The diffusion matrix for a two round homogeneous balanced Feistel network would be:

$$\mathcal{D} = \begin{bmatrix} \mathbf{0}_{n/2} & \mathbf{I}_{n/2} \\ \mathbf{I}_{n/2} & \mathbf{A}_{n/2}(\rho) \end{bmatrix} \times \begin{bmatrix} \mathbf{0}_{n/2} & \mathbf{I}_{n/2} \\ \mathbf{I}_{n/2} & \mathbf{A}_{n/2}(\rho) \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{n/2} & \mathbf{A}_{n/2}(\rho) \\ \mathbf{A}_{n/2}(\rho) & \mathbf{A}_{n/2}(\rho') \end{bmatrix}$$

where $\rho' \leq \rho$. For a two round heterogeneous balanced Feistel network, the resulting diffusion matrix would be:

$$\mathcal{D} = \begin{bmatrix} \mathbf{0}_{n/2} & \mathbf{I}_{n/2} \\ \mathbf{I}_{n/2} & \mathbf{A}_{n/2}(\rho_1) \end{bmatrix} \times \begin{bmatrix} \mathbf{0}_{n/2} & \mathbf{I}_{n/2} \\ \mathbf{I}_{n/2} & \mathbf{A}_{n/2}(\rho_2) \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{n/2} & \mathbf{A}_{n/2}(\rho_2) \\ \mathbf{A}_{n/2}(\rho_1) & \mathbf{A}_{n/2}(\rho') \end{bmatrix}$$

For instance, if a two round balanced Feistel network with $n = 4$ is considered, the diffusion of the cipher would be the product of the diffusions of each round:

$$\mathcal{D} = \beta(\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}) = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

For a one round balanced Feistel operating on the right input sub-block, the diffusion matrix would be:

$$\mathcal{D} = \begin{bmatrix} \mathrm{A}_{\mathrm{n}/2}(\rho) & \mathrm{I}_{\mathrm{n}/2} \\ \mathrm{I}_{\mathrm{n}/2} & 0_{\mathrm{n}/2} \end{bmatrix}$$

Consequently, if a left-oriented Feistel is combined with a right-oriented Feistel (that means that the swapping between the left and the right part are cancelled), the resulting diffusion would be:

$$\mathcal{D} = \begin{bmatrix} 0_{\mathrm{n}/2} & \mathrm{I}_{\mathrm{n}/2} \\ \mathrm{I}_{\mathrm{n}/2} & \mathrm{A}_{\mathrm{n}/2}(\rho_1) \end{bmatrix} \times \begin{bmatrix} \mathrm{A}_{\mathrm{n}/2}(\rho_2) & \mathrm{I}_{\mathrm{n}/2} \\ \mathrm{I}_{\mathrm{n}/2} & 0_{\mathrm{n}/2} \end{bmatrix} = \begin{bmatrix} \mathrm{I}_{\mathrm{n}/2} & 0_{\mathrm{n}/2} \\ \mathrm{A}_{\mathrm{n}/2}(\rho_3) & \mathrm{I}_{\mathrm{n}/2} \end{bmatrix}$$

i.e. no increase in the diffusion.

Unbalanced Feistel networks. The diffusion matrix is very helpful for describing the diffusion of a cryptosystem which could consist of any kind of UFNs, such as consistent, inconsistent, homogeneous, heterogeneous and so on.

A diffusion matrix for an $s:t$ UFN would be:

$$\mathcal{D} = \begin{bmatrix} 0_{(\mathrm{n-s})\times(\mathrm{n-t})} & \mathrm{I}_{\mathrm{t}} \\ \mathrm{I}_{\mathrm{s}} & A_{(s)\times(t)}(\rho) \end{bmatrix}$$

where $0_{(n-s)\times(n-t)}$ is a zero $(n-s)\times(n-t)$ matrix and $A_{(s)\times(t)}(\rho)$ is a matrix with $\rho \times 100\%$ ones.

It is expected that for an incomplete UFN, ρ is smaller than that from a complete one. In fact, in the case of an incomplete UFN, a zero matrix $0_{(s)\times(z)}$ would be a submatrix of $A_{(s)\times(t)}$.

It could be realised from the diffusion matrices that for $s:t$ UFNs with varying values of s and t, different number of rounds are required in order to obtain complete diffusion. The lowest number of rounds is that of a balanced Feistel network which is equal to 3.

Based on the properties of matrix multiplication, it could be observed that the least number of rounds for a homogeneous UFN for complete diffusion is equal to $C + 1$, where C is the cycle of the UFN. The constraint though is that in order to assure complete diffusion after the required rounds, the submatrix $A_{(s)\times(t)}(\rho)$ which is merely determined by the round function should have $\rho = 1$. For an incomplete UFN it is always $\rho < 1$ and more rounds are required. The DES for example, although being a balanced Feistel network, needs five rounds to obtain complete diffusion, as will be shown in Chapter 5.

3.5.6 The autocorrelation test

The autocorrelation test (Beker & Piper, 1982) checks whether the ones and zeros have a random distribution in a binary sequence. Autocorrelation is performed on the rows and vectors of the Ψ matrices.

More analytically, a row of a Ψ matrix indicates the relation between one input bit with every output bit. For a potentially strong block cipher a row should have 50% ones and 50% zeros. The same is required for the columns, which represents the relation of an output bit with every input bit. The autocorrelation test should run for each of the Ψ matrices and all matrices should pass the test.

However, this test has some limitations. First, it is not able to detect the similarities of the matrices i.e. if certain or all entries are the same or complement. Second, in most instances it could not run for single encryption steps, because it would fail in all runs. In one round balanced Feistel networks for instance, autocorrelation tests would fail, since randomness appears only in one quadrant, due to the effect of its round function. If it is accepted though that there is an inherent 100% failure rate for one round ciphers, for a product cipher the decrease in the failure rate could be examined, as the product cipher starts with one round and gradually the rounds increase until the specified limit.

3.5.7 The depth test

The depth test proposed in this thesis deals with the matrix similarity problem, as presented in section 3.5.6. If the sequence $\Psi_1, \Psi_2, \ldots, \Psi_L$ of the Ψ matrices used for the evaluation of the confusion matrix is considered, the intermediate values of the confusion matrix are evaluated:

$$\begin{aligned}
\mathrm{C_1} &= \Psi_1 \\
\mathrm{C_2} &= \frac{1}{2}(\Psi_1 + \Psi_2) \\
&\vdots \\
\mathrm{C_L} &= \mathcal{C} = \frac{1}{L}(\Psi_1 + \Psi_2 + \ldots + \Psi_L)
\end{aligned}$$

Eventually all entries should converge to 0.5. If the elements of $\mathrm{C_k}$ are denoted by c_{ij}^k, the depth matrix $\mathcal{H}$ is constructed as:

$$\begin{aligned}
\mathcal{H}_{n \times n} = \{h_{ij} = \max_{1 \leq l,m \leq n} (k | ([c_{ij}^k = c_{lm}^k] \vee [1 - c_{ij}^k = c_{lm}^k]) \wedge \\
([c_{ij}^{k-1} = c_{lm}^{k-1}] \vee [1 - c_{ij}^{k-1} = c_{lm}^{k-1}]) \wedge \ldots \\
([i \neq l] \vee [j \neq m])\}
\end{aligned}$$

where $[\cdot]$ evaluates the expression and returns a boolean value, as introduced in section 3.5.5, and $\wedge$ and $\vee$ denote the boolean **AND** and **OR** operators. The smaller the values of $\mathcal{H}$ the smaller the similarity between the Ψ matrices. In the extreme where $h_{ij} = L$, then input bit i has a linear relation with output bit j.

3.5.8 The diffusion distinguisher

The diffusion distinguisher proposed also in this project deals also with the similarities of the Ψ matrices. It is demonstrated in Appendix A that if two square matrices A

and B have ρ_a and ρ_b densities of zeros respectively ($\rho_a, \rho_b \in [0, 1]$), then if $C = A \times B$, the expected number of zeros would be:

$$\rho_c = (p_a + p_b - p_a p_b)^n$$

where n is the dimension of the matrices. Such relation is derived with the assumption that the zeros are randomly distributed in the matrix. For the products A^2 and B^2, the expected densities would be:

$$\rho_{aa} = \left[\frac{(2n^2 - z_a - 1)z_a}{n^2(n^2 - 1)}\right]^n$$

and

$$\rho_{bb} = \left[\frac{(2n^2 - z_b - 1)z_b}{n^2(n^2 - 1)}\right]^n$$

where z_a and z_b is the actual number of zeros in A and B respectively, i.e. $z_a = \rho_a n^2$ and $z_b = \rho_b n^2$.

By comparing the actual and estimated values, the diffusion distinguisher tests whether a cipher behaves as a random source when generating the Ψ matrices.

3.5.9 The Cryptographic Block Profile

The Cryptographic Block Profile CBP of a cryptographic block, summarises the structure and cryptographic properties of this block. The fields of a CBP are presented at Table 3.1. Once the CBP of all cryptographic blocks of the cryptosystem are determined, their behaviour could be evaluated when used as encryption steps in a product encryption.

The *name* field is the name of the encryption block. It could have any name, but it is useful when the underlying cipher is a round of a standard symmetric block algorithm, such as DES, Blowfish, etc. In this case, standard ciphers could be examined in the proposed context and furthermore, their combinations could also be assessed.

The *type* field indicates the type of the cryptographic block. According to the type of the cryptographic block, very general rules such as *"product of two permutations*

Table 3.1: The Cryptographic Block Profile, CBP.

name
type
source
target
total diffusion rate
marginal diffusion
confusion

has equivalent cryptographic strength of one permutation" could be applied. This field could be one of the following: Permutation, Substitution, Feistel, Benes, Misty.

The *source* field is an integer where when represented in a binary format, the ones specify the length as well as the position of the input to the round function within the input block. The Hamming weight of the source would be the length of the source. This field as well as the target field have no particular meaning in some cipher types such as permutations or simple substitutions and usually are set to their maximum value 2^n.

Similar to the source field, the *target* field represents the portion and the position of the sub-block where the output of the round function is applied. Given the source and target numbers, it is easy to realise that a block of a balanced Feistel for a 16-bit symmetric cipher would have either the pair $(source = 255, target = 255)$ or $(source = 65280, target = 65280)$.

The *total diffusion rate* differs from the diffusion rate and is computed directly from the diffusion matrix.

Definition 21. The *total diffusion rate* of a diffusion matrix (or a Ψ-matrix) is defined as:

$$R'_D(\mathcal{D}) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}\delta_{ij}}{2^n}$$

where δ_{ij} denotes the element of the ith row and the jth column of the diffusion matrix. The difference between the proposed novel quantity of the total diffusion rate and the diffusion rate is that the total diffusion rate of a product cipher could be calculated and is not restricted only to Feistel networks. For a transposition for example, the total diffusion rate would be $n/2^n$, since every input bit would relate to only one output bit. This value is the lowest bound for the total diffusion for a bijective cipher.

It should be noted that for convenience, the total diffusion rate would also be referenced as diffusion.

The *marginal diffusion* is also a novel quantity and demonstrates the capacity of a cipher to achieve its total diffusion rate.

Definition 22. Let $\Psi_1, \Psi_2, \dots, \Psi_L$ the Ψ-matrices which were generated from different inputs. The *marginal diffusion* is then defined as:

$$M_D = \text{mean}\{R'_D(\Psi_i \vee \Psi_j) - R'_D(\Psi_i)\} \; i \neq j$$

The marginal diffusion considers the number of rounds required for a cipher. In general, the higher the marginal diffusion, the less rounds are needed for a cipher. The reason is that if Ψ_i and Ψ_j are less similar, and having in mind that for a "good" cipher half of the entries should be ones and half zero, it should be expected that $R'_D(\Psi_i \vee \Psi_j)$ should be considerably higher than $R'_D(\Psi_i)$ for most of the cases. Therefore, less rounds are needed. On the contrary, small marginal diffusion implies need for increasing the avalanche, which yields more rounds.

The confusion is defined in this thesis as follows. The confusion is determined as the deviation from the perfect confusion (i.e. 1). That is, given the confusion matrix $\mathcal{C}$, The deviation from the ideal 0.5 would be the error $(0.5 - c_{ij})^2$ for all elements of $\mathcal{C}$. The sum of errors then would be subtracted from the expected value, normalised:

$$\text{confusion} = 1 - \sum_{i=1}^{n}\sum_{j=1}^{n}(0.5 - c_{ij})^2/1024$$

3.6 The feedback blocks

The feedback blocks do not increase the security in terms other than exhaustive search. More specifically, if the underlying product cipher - without the feedback - falls to a differential or linear cryptanalytic attack, the security of the whole cryptosystem could be compromised, since the feedbacks do not affect the performance of such attack; in fact, they may support it in some cases.

The feedback blocks are employed for two main properties. First, they increase the search space, so that an exhaustive search could not be feasible, since not only the key should be searched, but also the cipher space. Although this is achieved partly by the secret structure of the encryption blocks, with the use of the feedback blocks this space is increased by many orders of magnitude.

Second, in internetworking environments pattern repetition in a ciphertext may be undesirable. This is the main problem overcome by modes of operation based on feedbacks.

However the feedback blocks are an option; the project mainly focuses on the combination of the encryption. Theoretically a large number of feedback blocks could be used for construction, but in practice such extreme is not necessary, since it results into very high computational costs without significant benefits.

Another problem is that the analysis of a cryptosystem with multiple modes of operation is very difficult. Therefore, the more feedback blocks in a cryptosystem, the more likely cancellations of cryptographic transformations may occur. It is generally accepted that a very high degree of complexity could be *illusionary* and a cryptosystem may not be as strong as it seems. Yet the feedback option should be included in order to enable further research.

The use of confusion and diffusion as measures of cryptographic strength, are focusing on the product cipher, rather than the feedback blocks. Therefore, the calculated values of confusion and diffusion for a cipher are independent to the existence of feedback blocks, since there are differential calculations involved, in which the feedback outputs are regarded as constants and cancelled.

3.7 Concluding remarks

This chapter has presented a method for combining encryption steps and has described a number of evaluation measures, in order to generate and evaluate ciphers which are likely to be practically secure. A cipher instance is described as a number of layers, where at the bottom layer the encryption blocks are placed to form the product cipher. The feedback blocks may exist and are placed above the encryption blocks. The security is offered at the bottom layer of the non-standard cipher where the encryption steps are placed, whereas the feedback blocks increase the total search space. Moreover, the feedback blocks characterise the cryptosystem as a cryptosystem with multiple non-standard cryptographic modes of operation, offering the advantages and disadvantages multiple modes of operation have, which were presented in the literature review.

In order to describe information concerning the cryptographic strength of a product cipher, the cryptographic block profile, CBP was presented. The CBP summarises the properties of each underlying encryption step. The information obtained from a CBP indicates the number of rounds which are needed for an encryption step to obtain complete diffusion, as well as if a combination of two or more encryption steps may result to an actual increase of cryptographic strength. Accordingly, a number of rules for the product of the encryption steps are described.

By using a scheme where a cryptographic algorithm is constructed by publicly agreed encryption steps, the evaluation is feasible and timely. There are no threats due to the alien code, because the encryption steps and feedback transformations could be developed by both parties independently, following the publicised specifications. The evaluation is concluded with the use of the CBPs which although they could be public, they could also be verified.

The definition of the confusion and diffusion matrices also contribute to the determination of the potential security of a cipher. It is generally accepted that for the unpublished algorithms their actual security is not necessarily equal to the security set by their inventors. An experimental analysis may involve a series of statistical tests, which may help in separating a weak cipher from a potentially strong one. However, it is easier to establish whether a cipher is weak - when it fails to pass a test - than determining that the cipher is strong, since it may fail other tests. The confusion matrix reveals some information about the behaviour of a cryptosystem concerning the relation between the ciphertext and the plaintext.

The diffusion matrix is very helpful because it can determine how the encryption steps of a product encryption contribute to the diffusion of the cryptosystem. It was demonstrated that the diffusion matrix of a product cipher is equal to the product of the diffusion matrices of the underlying encryption steps. This has two advantages. First, one knows which types of encryption steps to use in order to obtain a desired result (diffusion). Second, there is no need to test a whole product cipher in order to determine its diffusion; determining the diffusion by computational means is a costly process. The diffusion matrices of the individual blocks could have been already computed so they could be used for different combinations (products).

The next chapter describes the prototype which was used to run the tests in order to determine the experimental part of the CBP and to test several cipher instances, in order to generate information and embed it in a client/server talk utility.

Chapter 4

Prototype of ABSENT (ABSolute ENcrypTion)

4.1 Introduction

One of the aims of this thesis is to provide a framework for effective evaluation of non standard cryptographic algorithms which are product ciphers with complex modes of operations (Aim 2.). The effectiveness of the evaluation is related to:

1. The capability of the underlying evaluation methods to distinguish weak instantiations of the proposed cryptosystem from potentially strong ones (Aims 2.(a),(b)).

2. The speed of the evaluation, so it could be performed on line and during a communication setup session between two communicating parties (Aim 3.), such as a client and a server in an internetworking environment.

4.2 Outline

This chapter describes the development of the prototype which is presented in Figure 4.1.

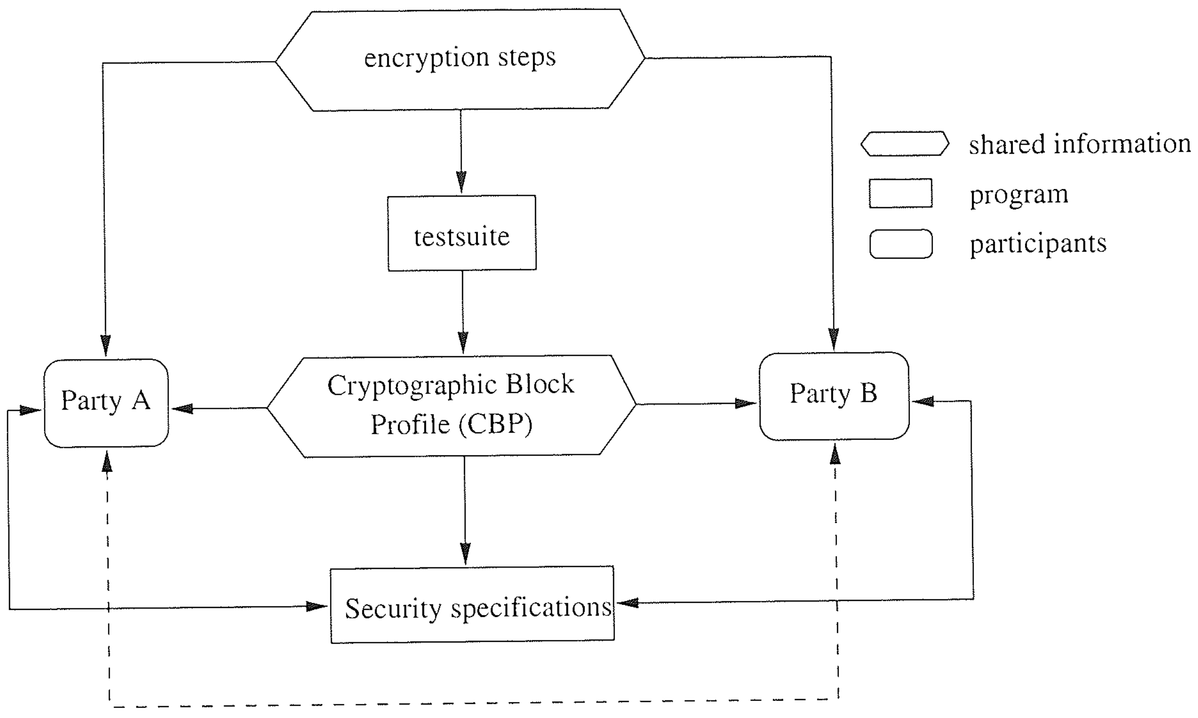


Figure 4.1: The ABSENT framework.

Once a number of encryption steps which form the encryption blocks and the feedback blocks have been selected, they are fed to the test suite in order to investigate their statistical properties not only as stand alone blocks, but as combination of blocks in particular. State of the art methods such as techniques for generating S-boxes resistant to differential cryptanalysis could be employed, or existing publicised encryption steps could be used. The cryptographic block profile is completed by testing each cryptographic primitive separately, and updating the respective fields.

The combination of the cryptographic blocks raises doubts on the security of the resulting ciphers. To provide a measure of security an existing standard cryptosystem with accepted security as a reference, its statistical properties could be assessed with the testsuite and be used for comparison with alternative non standard cryptographic algorithms.

By determining the CBP of the involved encryption steps, the security specifications would be explicitly defined by a set of rules which would use the resources of the CBP fields. That is, the information stored in a CBP should be capable to apply every rule developed or obtained from the literature (which are mainly lemmas) which would contribute in the establishment of the cryptographic strength of a cipher instance. The security specifications should be in such a form, in order to facilitate updates and effective control over the security level.

The client and the server and in general any communicating party should share the cryptographic primitives, the CBPs and the security specifications, providing the standardisation aspect which is required for effective communication. The client could implement independently the cryptographic algorithm generator for developing product ciphers.

Prior to a client's service request, the client should check whether the non standard algorithm reach the negotiated security level. Once this condition is met, the algorithm is sent to the server. The latter would again check the security of the received algorithm and accept or reject the client's service request.

4.3 The cryptographic primitives

The module *cag.c* shown in Appendix D implements the encryption steps which form the encryption and decryption blocks. In the same module the feedback blocks are implemented.

Arrays of pointers to functions were used for calling the cryptographic blocks as well as the feedback blocks, in order to call them dynamically from the description matrix of the cryptographic algorithm which is described later on. Encryption, decryption and feedback blocks would be referred as transformation blocks. The declaration of a transformation block function is as follows:

static unsigned char $*name$(**unsigned char** $*$);

where *name* represents the name of the transformation block. The function requests an 8-byte block and returns the result at the same address.

All transformations χ_i are permutations of the alphabet $\mathcal{V} = \{0,1\}^{64}$, i.e.

$$\chi_i : \mathcal{V} \to \mathcal{V}$$

Three arrays of pointers to functions were defined, in order to call the functions indicated by the description matrix:

unsigned char *(*nfunct[NUM_FUNCT+1])();

unsigned char *(*nefunct[NUM_CFUNCT+1])();

unsigned char *(*ndfunct[NUM_CFUNCT+1])();

where **nfunct** is the array which holds the addresses of the feedback transformation functions, and **nefunct** and **ndfunct** hold the addresses of the encryption and decryption functions. **NUM_FUNCT** and **NUM_CFUNCT** are the number of feedback functions and encryption steps implemented in the prototype. These constants are defined in *fstuff.h*.

4.3.1 Encryption blocks

The test cryptosystem consists of 15 encryption steps. They range from simple monoalphabetic substitutions to Unbalanced Feistel Networks with the round function offering complete diffusion between the respective input and output bits.

The encryption steps which are selected for a product cipher instantiation form the cryptographic blocks which are cascaded and some or all of them may be combined with feedback blocks. More precisely, every encryption step which is included in the cipher instance is called an encryption block, since this term is more appropriate to describe a structure. In terms of standard cryptographic algorithms, the DES round is included, as well as the initial and inverse initial DES permutations. The S-boxes which are used in some encryption steps are those specified for the blowfish block algorithm. These are four S-boxes (**blow0**[ ] – **blow3**[ ])which require a byte and produce a 32–bit output.

Permutations (transpositions)

There are four permutation primitives defined in the prototype. Two of the permutations operate on a byte level, whereas the other two are the DES initial permutation

(IP) and inverse initial permutation (IP^{-1}) which permute the individual input bits. More analytically, the two permutations are the following:

Permute #1: (2 0 7 3 6 4 1 5)

Permute #2: (7 0 3 1 5 6 2 4)

It should be noticed that the above permutations are key independent similar to the DES initial permutation. For the first permutation for instance, if the input *abcdefgh* is applied, the output would be *cahdgebf*.

Vigenère substitution

One polyalphabetic substitution is implemented, by applying 8 Caesar additions to the 8 input bytes respectively. Such substitution is actually a Vigenère encryption step. The subkey is considered as the keyword which is used in such encryption and every *character* (i.e. byte) is added to the respective plaintext byte **mod** 256.

Balanced Feistel Networks

1-round DES, 2-round DES. The most popular balanced Feistel network is the DES and is included in the prototype. The actual code used in the prototype for the DES is due to Eric Young and appears in Appendix D. There are two encryption steps which call the DES round. One encryption step is a one round DES and the other is a two round DES. The use of two round DES reduces the calls to the encryption and decryption functions by a factor of two, so the DES could be implemented with ten blocks (including the initial and final permutation) instead of 18. However, the one round DES step should also be included for analysing the statistical properties of the DES round function as well as the reduced steps.

It should be highlighted that the subkeys which are used in the DES rounds are different from those in the rest of the transformation blocks. That is, there are two key schedules - one is the standard DES key schedule and the second is the strong key schedule which was described in Chapter 3.

Feistel #1. This balanced Feistel network uses iterative transformations which are defined by the RIPE hash function, as a round function. All specified rounds of the 128-bit version of the hash function are used as a key schedule for generation of the round keys. The underlying Feistel uses the sequence of transformations determined by round 1 and parallel round 3 of the RMD hash transformation. Such combination appeared to produce a function where its output provided good random properties. That is, the output sequence which was generated by this function passed the statistical tests for local randomness (serial and frequency tests).

Feistel #1 accepts the right half of the input (least significant bits) to the round function and applies its output to the left half (most significant bits). The round function is defined as:

```
aa = aaa= 0x67452301UL;        /* constants defined for the RMD hash. */
bb = bbb= 0xefcdab89UL;
cc = ccc= 0x98badcfeUL;
dd = ddd= 0x10325476UL;

t1 = right_input_half;         /* copy the 32 least significant input bits.
FF(aa, bb, cc, dd, t1, 11);    /*round 1 of RMD */
FF(dd, aa, bb, cc, t1, 14);
FF(cc, dd, aa, bb, t1, 15);
FF(bb, cc, dd, aa, t1, 12);
FF(aa, bb, cc, dd, t1,  5);
FF(dd, aa, bb, cc, t1,  8);
FF(cc, dd, aa, bb, t1,  7);
FF(bb, cc, dd, aa, t1,  9);
FF(aa, bb, cc, dd, t1, 11);
FF(dd, aa, bb, cc, t1, 13);
FF(cc, dd, aa, bb, t1, 14);
FF(bb, cc, dd, aa, t1, 15);
FF(aa, bb, cc, dd, t1,  6);
FF(dd, aa, bb, cc, t1,  7);
FF(cc, dd, aa, bb, t1,  9);
FF(bb, cc, dd, aa, t1,  8);

GGG(aaa, bbb, ccc, ddd, t1,  9);  /* parallel round 3 of RMD */
GGG(ddd, aaa, bbb, ccc, t1,  7);
GGG(ccc, ddd, aaa, bbb, t1, 15);
GGG(bbb, ccc, ddd, aaa, t1, 11);
GGG(aaa, bbb, ccc, ddd, t1,  8);
GGG(ddd, aaa, bbb, ccc, t1,  6);
GGG(ccc, ddd, aaa, bbb, t1,  6);
GGG(bbb, ccc, ddd, aaa, t1, 14);
GGG(aaa, bbb, ccc, ddd, t1, 12);
GGG(ddd, aaa, bbb, ccc, t1, 13);
GGG(ccc, ddd, aaa, bbb, t1,  5);
GGG(bbb, ccc, ddd, aaa, t1, 14);
GGG(aaa, bbb, ccc, ddd, t1, 13);
GGG(ddd, aaa, bbb, ccc, t1, 13);
GGG(ccc, ddd, aaa, bbb, t1,  7);
GGG(bbb, ccc, ddd, aaa, t1,  5);
```

where the function *FF*() and the operation *GGG*() are transformations defined in *rmd128.h* (Bosselaers 1996).

Feistel #2. Similar to the previous encryption step, *Feistel #2* adopted some rounds from the RIPE hash function. More specifically, the round function consists of round 2 and parallel round 4 of the hash function. Again, the combination of these two rounds were found to produce a good locally random sequence.

The round function of *Feistel #2* operates on the right half of the input and is defined as follows:

```
t1 = left_input_half;         /* copy the 32 most significant input bits. */
II(aa, bb, cc, dd, t1, 11);   /*round 4 of RMD */
II(dd, aa, bb, cc, t1, 12);
II(cc, dd, aa, bb, t1, 14);
II(bb, cc, dd, aa, t1, 15);
II(aa, bb, cc, dd, t1, 14);
II(dd, aa, bb, cc, t1, 15);
II(cc, dd, aa, bb, t1,  9);
II(bb, cc, dd, aa, t1,  8);
II(aa, bb, cc, dd, t1,  9);
II(dd, aa, bb, cc, t1, 14);
II(cc, dd, aa, bb, t1,  5);
II(bb, cc, dd, aa, t1,  6);
II(aa, bb, cc, dd, t1,  8);
II(dd, aa, bb, cc, t1,  6);
II(cc, dd, aa, bb, t1,  5);
II(bb, cc, dd, aa, t1, 12);

HHH(aaa, bbb, ccc, ddd, t1,  9);  /* parallel round 2 of RMD */
HHH(ddd, aaa, bbb, ccc, t1, 13);
HHH(ccc, ddd, aaa, bbb, t1, 15);
HHH(bbb, ccc, ddd, aaa, t1,  7);
HHH(aaa, bbb, ccc, ddd, t1, 12);
HHH(ddd, aaa, bbb, ccc, t1,  8);
HHH(ccc, ddd, aaa, bbb, t1,  9);
HHH(bbb, ccc, ddd, aaa, t1, 11);
HHH(aaa, bbb, ccc, ddd, t1,  7);
HHH(ddd, aaa, bbb, ccc, t1,  7);
HHH(ccc, ddd, aaa, bbb, t1, 12);
HHH(bbb, ccc, ddd, aaa, t1,  7);
HHH(aaa, bbb, ccc, ddd, t1,  6);
HHH(ddd, aaa, bbb, ccc, t1, 15);
HHH(ccc, ddd, aaa, bbb, t1, 13);
HHH(bbb, ccc, ddd, aaa, t1, 11);
```

where $II()$ and $HHH()$ are defined in *rmd128.h* and $aa \ldots ccc$ are initialised as in *Feistel #1*.

Blowfish L. This encryption step is a balanced Feistel cipher with the round function presented at Figure 4.2. The round function is the one used for the Blowfish encryption algorithm. However, the key schedule employed is generated differently from the one described for the original algorithm. Since the subkeys are generated using a strong key schedule, i.e. they are generated by a one way hash function and they provide no significant information about the master key, the strength of the cipher should be at least the same as that of the original algorithm.

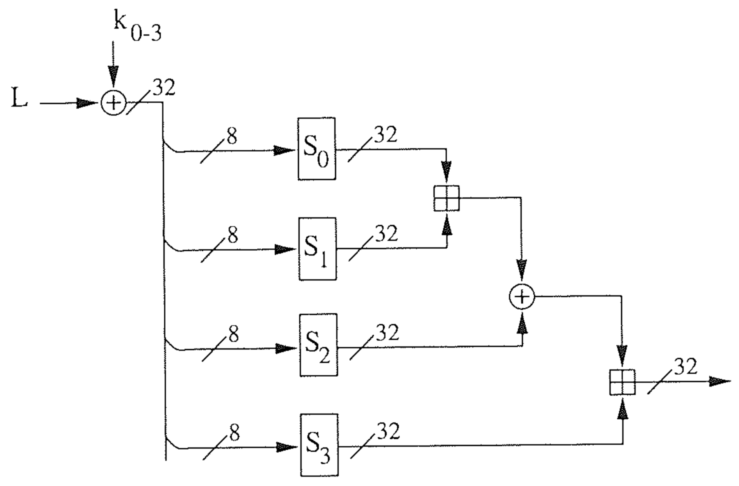


Figure 4.2: The round function of Blowfish.

Blowfish R. The *Blowfish R* encryption step has the same round function as the previous one, but the difference is that the right half of the input is applied to the round function, i.e. the 32 least significant input bits, instead of the 32 most significant input bits. *Blowfish R* and *Blowfish L* (Figure 4.3) allows to study iterative ciphers where there is no swapping between the left and right part of the output, such as the proposal by Koyama and Terada (1993) which was applied on the DES.

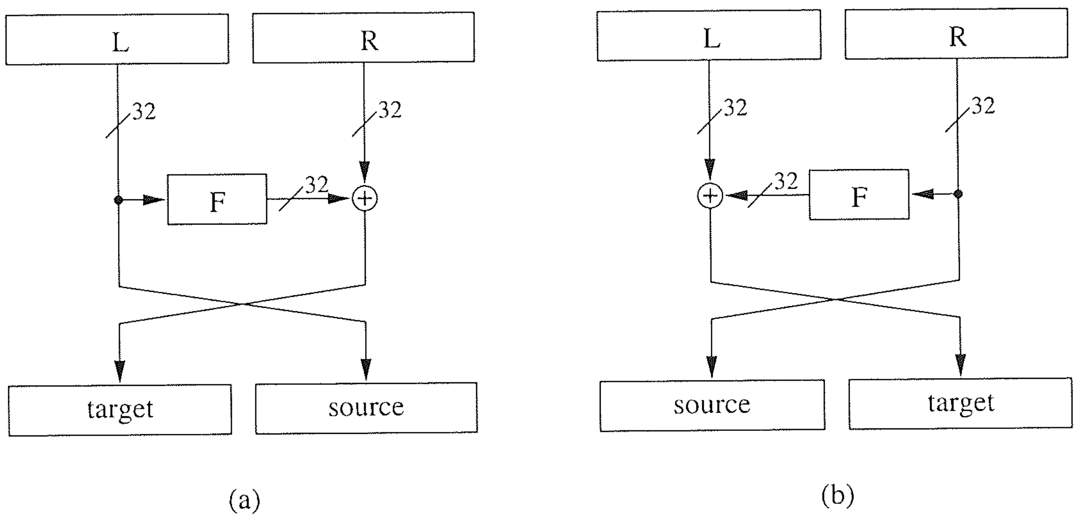


Figure 4.3: The *Blowfish L* (a) and *Blowfish R* (b) encryption steps.

Unbalanced Feistel Networks

The prototype implements two UFNs, one which is target-heavy, and one which is source-heavy. More particularly, the two UFNs are the following:

UFN 16:48. This target heavy UFN is presented at Figure 4.4. The round function accepts the 16 least significant input bits and cyclically rotates them by six and eight bits. The rotation increases the diffusion of the involved input bits. The eight most significant bits from the first rotation are **XOR**ed with the third subkey, whereas the eight most significant bits from the second rotation are **XOR**ed with the fifth subkey. These two blocks of eight bits are fed in the blowfish S-boxes S_0 and S_1 respectively and their outputs are **XOR**ed. The latter output consists of the 32 most significant bits of the output to the round function.

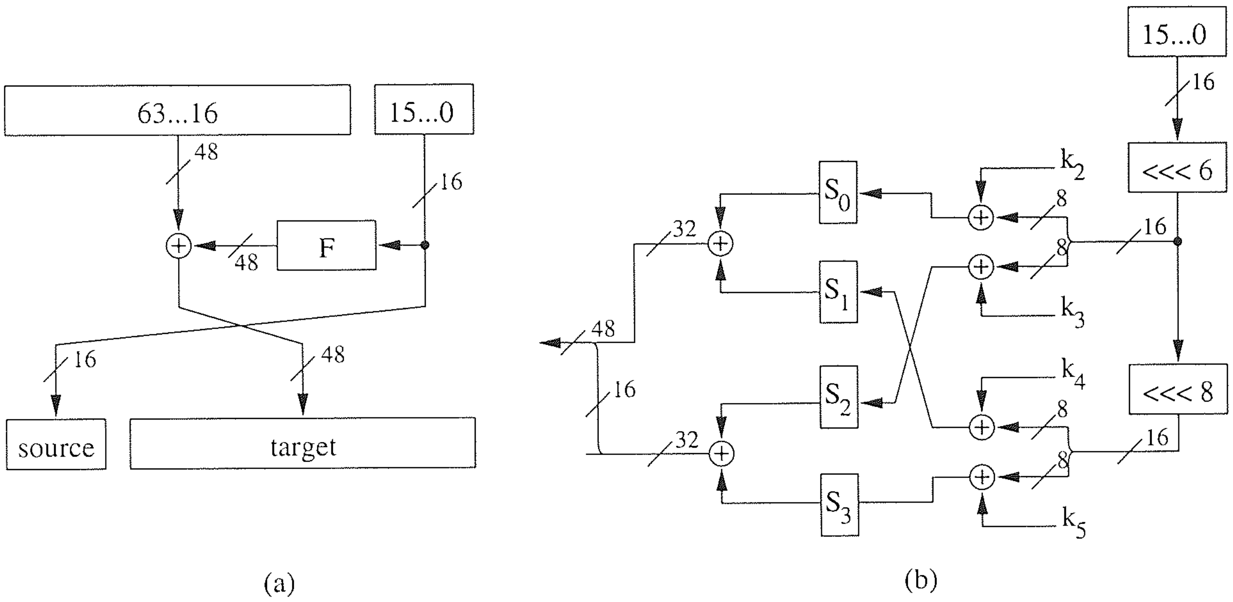


Figure 4.4: The target heavy 16:48 UFN (a) and its round function (b).

Similarly, the eight least significant bits of the first rotation are **XOR**ed with the fourth subkey and the eight least significant bits of the second rotation are **XOR**ed with the sixth subkey. The two results are fed to the S-boxes S_1 and S_3 and the outputs are **XOR**ed. The 16 least significant bits of this operation form the remaining 16 lest significant bits of the round function.

UFN 40:24. The round function of the source heavy UFN is merely a hash function (Figure 4.5). Once again the four S-boxes are used, but their outputs are added in modulo 2^{32} and **XOR**ed as shown in the figure and the 24 least significant bits are selected as the output of the round function.

Since there were 40 (=5×80) input bits and only four S-boxes, S_2 was repeated in order to transform the remaining eight bits. The round function is similar to the round function of the Blowfish. The difference is that the third and fourth S-box are combined by addition modulo 2^{32}, before being **XOR**ed with the first two S-boxes and the last S-box (S_2) is added modulo 2^{32} to the previous result.

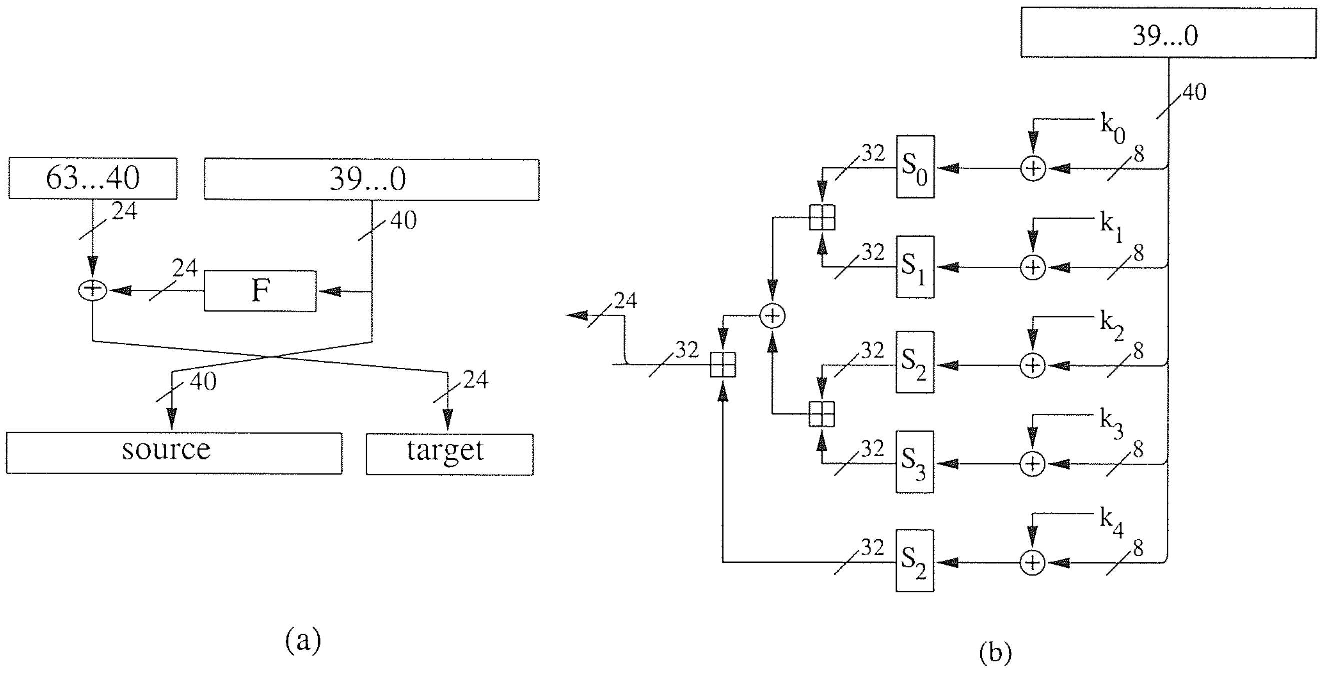


Figure 4.5: The source heavy 40:24 UFN (a) and its round function (b).

Linear Feedback Shift Registers

In general, LFSRs could be effectively cryptanalysed, given a small amount of the generated sequence. Therefore it is suggested that when they are used as a cryptographic primitive, their output should not be combined directly with the ciphertext, but should pass through another transformation, preferably a one way function. Two encryption steps which involve LFSRs are included in the prototype. It should be noted all LFSRs consist of 32−bit shift registers.

LFSR #1. There are actually five 32-bit LFSRs specified and they are selected by some of the bits of the round keys. That is, there is a 32-bit register, and the feedback function is a primitive polynomial which is selected among five polynomials. The number of available primitive polynomials could be easily extended if required.

The initial state of the shift register consists of the key bytes $7, 3, 4, 5$, **XOR**ed with the fourth input byte. The latter is concatenated four times in order to form a 32-bit block, for the bitwise **XOR** operation. The first 32 shifts of the LFSR are discarded, since they form the initial state. The following 56 generated bits are **XOR**ed with all input bits excluding those which form the fourth input bit, because such information is required for decryption. This process is illustrated in Figure 4.6. For simplicity, the clock signal is omitted.

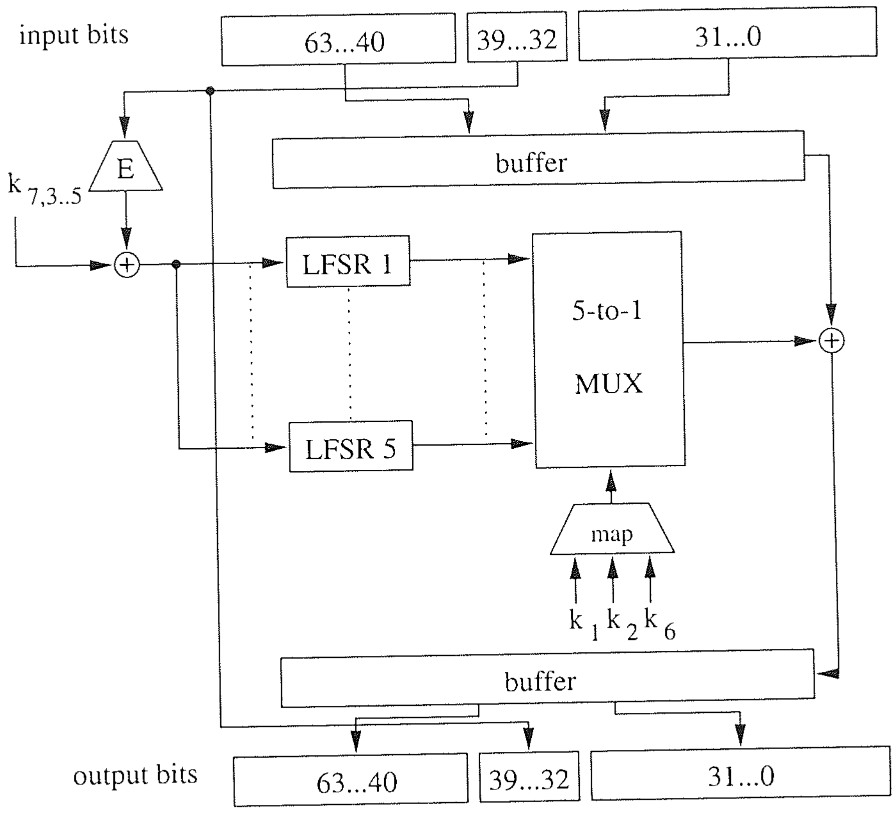


Figure 4.6: The LFSR #1 encryption step.

In general, this encryption step as well as *LFSR #2* are relatively slow, compared to the other encryption steps, since it is required to perform many cycles and the first ones are redundant. The same function is called for the decryption.

LFSR #2. The previous encryption step is very weak since the output of the LFSR is applied directly to the input bits. *LFSR #2* is stronger, because the output sequence is accumulated in a buffer r and the quantity $r^3 \ \mathbf{mod}(2^{32})$ is applied to the input bits instead. Additionally, the initial state of the LFSR is specified by the 16 least significant bits, i.e. the two last bytes of the input block, combined with keys k_2 and k_4 by a bitwise **XOR**. The key bits k_0, k_3 and k_5 are involved in the selection of the LFSR.

4.3.2 Feedback blocks

A very important consideration when using feedback blocks is that direct wirings should not be used, because they construct a short path where the differential characteristics could be back-propagated (Biham 1994). It is conjectured that a feedback block should maintain properties of uniformly differential mappings, different to the cryptographic block they are supporting. This would mean that the output value of the feedback would appear to be pseudorandom.

Theoretically any function could serve as a feedback block. This is because it need not be inverted during decryption. This is due to the fact that when the feedback in encryption is a feedforward in decryption and produces the same value in both configurations.

The proposed model could support stream cipher modes, if the feedback blocks are modified in a way that their input is discarded and the underlying primitive is a random number generator. However, such blocks are not considered in order to avoid synchronisation issues between the sender and the receiver.

In addition to the encryption steps which are also available as feedback blocks, the following were also implemented:

Rotate left and right. These two key dependent rotations operate separately on the two input halves. The left rotation rotates the right half by $k_0 \ \mathbf{mod}(32)$ bits and the left half by $k_1 \ \mathbf{mod}(32)$ bits. The right rotation operates accordingly by applying the subkey k_1 on the left half and the subkey k_2 on the right half of the input block.

It should be noted that rotations are very weak transformations and should be not used alone in a feedback, because they preserve linearity between input and output.

Hash #1. This is a one way hash function based on a Benes network(Aiello & Venkatesan, 1996) as presented in Figure 4.7. The entries to the S-boxes are the product of $k_3 \times R \ \mathbf{mod}(2^8)$ and $k_4 \times L \ \mathbf{mod}(2^8)$, where R and L are the right and left half respectively. The left and right input halves are added with the respective outputs of the S-boxes.

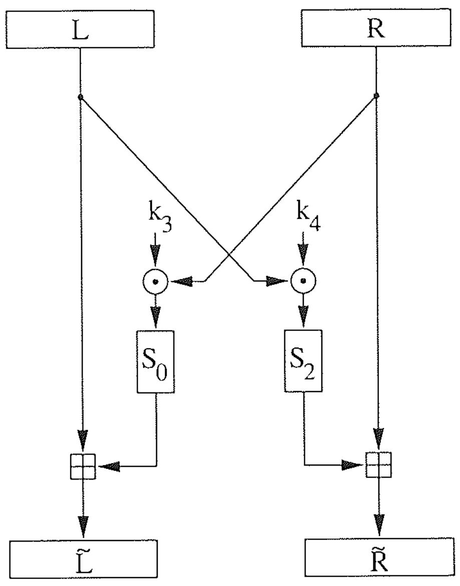


Figure 4.7: The hash function used in the feedback block.

Plus and times mod 2^{32}. These are two weak transformations, where the input is either added or multiplied with key schedule bytes. For the multiplication, the sixth and seventh key bytes are used, whereas for the addition the seventh and eight bytes are used. These operations should not be used alone in a feedback, but supported by other feedback blocks.

4.4 The key schedule

As mentioned in a previous section, two key schedules were implemented, the DES key schedule and a stronger key schedule based on a one way hash function. The DES key schedule was required in order to enable the model to implement the DES encryption algorithm. The DES subkeys were defined in Young's C code (Appendix D), where the following type was declared:

```
typedef struct des_ks_struct
      {
        union {
                  des_cblock _;
                 /* make sure things are correct size on machines with
                 * 8 byte longs */
                unsigned long pad[2];
                } ks;
#define _       ks._
      } des_key_schedule[16];
```

Since the model should be capable of handling cryptographic algorithms of variable lengths, the key schedule was modified to:

des_key_schedule[(**MAX_LAYERS**> 16) ? **MAX_LAYERS** : 16];

where **MAX_LAYERS** is the maximum allowed length of the cryptographic algorithm, defined in *fstuff.h*.

4.5 The testsuite

The *testsuite.c* module which is listed in the Appendix D can be used interactively or it can execute a script. Once testsuite runs, it enters the interactive command mode:

```
Welcome to ABSOLUTE ENCRYPTION test suite

Type ? for help.

ABSENT>_
```

The list of available commands is displayed by typing ?:

```
ABSENT>?

seed        - create an algorithm by giving a seed
random      - create automatically a random algorithm
define      - construct an algorithm manually
edit        - edit the current algorithm
list        - list of cryptographic primitives
display     - display the current algorithm
graph       - draw the current algorithm
show        - demonstrate the transformation of a cryptographic primitive
key[bin]    - display or change the key [in binary format]
encrypt     - encrypt a string
run         - run encryption/decryption sequences
speed       - perform a time trial on the current algorithm
test        - measure confusion/diffusion
ciphertext  - encrypt a file
```

```
plaintext    - decrypt a file
save         - save the current algorithm
load         - load an algorithm
script       - execute an ABSENT script (? script for help).
quit         - quit ABSOLUTE ENCRYPTION

ABSENT>_
```

The description of the commands is at the Appendix C. This chapter focuses on the commands regarding the tests which assist in the evaluation of a block cipher.

4.5.1 Statistical tests

The available statistical tests are specified in the `test()` function. Since most of the tests are computationally intensive, they are all specified in the same function in order to minimise the number of calls to other functions and thus reduce the additional overheads. The only calls to other functions are those to encrypt and decrypt the data, which could not be avoided. Consequently, some code is repeated for different tests. Such an approach is favourable because the variables which are involved in most of the computations are declared as register variables, increasing the overall speed of the tests.

Tests for randomness

There are three tests for randomness which are applied on the ciphertext, namely the frequency test, the serial test and the autocorrelation test (Chapter 2.9). The first two tests consider each output bit as a separate source of bit generator, whereas the autocorrelation test examines the distribution of ones and zeros in the 64-bit output.

The frequency test. The frequency test examines the appearance of ones and zeros of each of the 64 output sequences when the input is one of the following:

1. Linear (whole input)

2. Linear per byte

3. Random
4. Linear byte - constant others
5. Structured input

Input of type (1) starts with a random value and then increases the input with a given increment step. Input of type (2) is similar to (1), but in addition a byte $(0-7)$ should be selected. That byte would increase linearly with a given increment step, but the other bytes would be updated with random values.

Input of type (3) would cause the whole input to have random values in every run. This test is useful to investigate whether the cipher introduces any bias towards ones or zeros. It should be mentioned that the random number generator for the input should produce ones and zeros with a uniformal distribution.

Input of type (4) is as (2) with the difference that once a byte is selected to increase linearly, the remaining bytes have a constant value. The test should run for at most 256 loops, given that the increment step is one, since after that number of loops it would pass from the same values and it would not provide any additional information.

Finally the structured input runs for a fixed number of loops and the input is assigned the values from an array which is defined as a constant. This type examines whether the cipher is able of taking a structured plaintext and producing randomness from it. The structured input consists of the following:

```
const unsigned char struct_in[]={
                      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                      0xff,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                      0x00,0xff,0x00,0x00,0x00,0x00,0x00,0x00,
                      0x00,0x00,0xff,0x00,0x00,0x00,0x00,0x00,
                      0x00,0x00,0x00,0xff,0x00,0x00,0x00,0x00,
                      0x00,0x00,0x00,0x00,0xff,0x00,0x00,0x00,
                      0x00,0x00,0x00,0x00,0x00,0xff,0x00,0x00,
                      0x00,0x00,0x00,0x00,0x00,0x00,0xff,0x00,
                      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff,
                      0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
                      0x00,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
                      0xff,0x00,0xff,0xff,0xff,0xff,0xff,0xff,
                    0xff,0xff,0x00,0xff,0xff,0xff,0xff,0xff,
                    0xff,0xff,0xff,0x00,0xff,0xff,0xff,0xff,
                    0xff,0xff,0xff,0xff,0x00,0xff,0xff,0xff,
                    0xff,0xff,0xff,0xff,0xff,0x00,0xff,0xff,
                    0xff,0xff,0xff,0xff,0xff,0xff,0x00,0xff,
                    0xff,0xff,0xff,0xff,0xff,0xff,0xff,0x00,
                    0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00,
                    0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff,
                    0x00,0xff,0x00,0xff,0x00,0xff,0x00,0xff,
                    0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa};
```

The above array consists of 22 lines. This means that the test would run for 22 loops. The array can be easily extended to consider more inputs. This type is more applicable in the autocorrelation test where the output block is considered a random sequence. Actually, a structured sequence is defined as a sequence which fails to pass the autocorrelation test.

For the frequency test hypothesis testing is used with a significance level of 5%. Figure 4.8 shows a sample of the results of the test. The 'X's under the respective output bit indicate if that bit has passed or not the frequency test.

```
Output bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
---------------------------------------------
   PASS    |   | X |   | X | X | X |   | X |
   FAIL    | X |   | X |   |   |   | X |   |

Output bit | 8 | 9 |10 |11 |12 |13 |14 |15 |
---------------------------------------------
   PASS    | X |   | X | X | X |   | X | X |
   FAIL    |   | X |   |   |   | X |   |   |
```

Figure 4.8: Format of the results for the first 16 output bits for a frequency test.

The serial test. The input conditions for the serial test are those described for the frequency test. The output of the test has the same format as the frequency test.

The autocorrelation test. The autocorrelation test is applied on the output block which is considered a binary sequence and is examined to check whether the ones and zeros are randomly distributed. There is only one input type for the autocorrelation test, namely the structured input, (5). By applying a structured input to the cipher, it is expected that the cipher would destroy any symmetries. The output would be a percentage of the number of success runs to the total number of attempts which in this case is equal to 22. The hypothesis is again on a 5% significance level.

The differences tests

Once the cipher succeeds in passing the required tests for randomness, the tests based on the differences between input and output pairs could be performed. The reason for these following the randomness tests is that most of the difference tests are more computationally intensive than the statistical tests. Consequently, when a cipher fails to pass the tests for randomness, there is no need to continue with the differences tests. These series of tests are the more important than the randomness tests in this project, because the provide more specific information including the computation of confusion and diffusion

However, when evaluating the encryption steps separately in order to update their cryptographic block profiles, the difference tests are only needed, so there is no particular reason to run the tests for randomness, since it is expected that they would fail to pass most of them.

The block cipher test (confusion/diffusion). This test provides information about the state of confusion and diffusion. In the ideal case for a cipher, for each one-bit change in the input or the key, all output bits are able to change with a probability of 0.5.

After completion of every input or key bit test, two graphs are produced. The first graph is the plot of the frequency of the changes and the second graph is the plot of the Hamming weight (i.e. the number of ones in a binary string) of every output difference. There is an option to store the data produced in a file, which can be read from Matlab[1]

[1]MATLAB is an integrated technical computing environment developed by "The MathWorks Inc."

to display the frequencies and the weights as two surface plots. The graphs are plotted by a function call to `plot()` which is called a total of 128 times, 64 for the frequency plots and 64 for the weight plots. Hence, there are no substantial overheads in this case, since all computations other than the plots are much more intensive than those needed for plots.

In every run there are two encryptions. The first is performed on a random input (or key) and for the second input bit (or key bit) b_j is inverted. The two ciphertexts are **XOR**ed and the result is used to update the frequency and Hamming weight tables. The same process is run for all input (or key) bits, $0 \leq j \leq 63$.

The number of runs L is required for the bit tests. The total number of encryptions would then be:

$$\text{(number of encryptions)} = 2 \times 64 \times L$$

Diffusion matrix. The diffusion matrix is produced with procedure similar to the block cipher test. Every difference produced is **OR**ed with the previous difference. The result of these binary operations is to have an increasing number of ones as the number of these tests increase because of the property of the binary **OR** involved.

Confusion matrix/depth analysis. The calculation of the depth matrix is the most computationally expensive test because a pairwise comparison between all elements of the 64×64 matrix. The same information is used for the generation of the confusion matrix. The depth matrix could be saved to a file for Matlab or any other data processing package.

Although the diffusion matrix could be obtained from the confusion matrix, the former is generated by another process as the computational requirements for computing only the diffusion matrix are much lower than the other matrices.

Total diffusion/marginal diffusion. The only input conditions for the total and marginal diffusion are the requests to compute these values based on the plaintext input or the key. Once the variable has been selected, the diffusion matrix starts being constructed and the total diffusion is recorded on every run. The marginal diffusion is the difference between the total diffusion of the current run minus the total diffusion of the previous run.

The process terminates when either complete diffusion has been reached, or the marginal diffusion is zero for more than four consecutive runs. In both cases the number of runs is also recorded. If there is a point where the marginal diffusion is zero, this would mean that the cipher is not capable of increasing the total diffusion towards its maximum, 1.00. However, the test must allow the cipher to run at least four times even when the marginal diffusion is zero, since that value may occur because of a certain random input sequence. After four zeros it would be unlikely that the failure to diffusion increment is due to the poor selection of the random inputs.

The expected marginal diffusion for a good cipher for the first run is 0.5, since half of the inputs have changed, 0.25 for the second run, since half of the remaining zeros have changed, and so on.

The diffusion distinguisher. The diffusion distinguisher test runs on the Ψ matrices. The four quadrants of the matrix are labelled as follows:

$$\Psi = \begin{bmatrix} Q_1 & Q_2 \\ Q_3 & Q_4 \end{bmatrix}$$

and the results are summarised as in Figure 4.9. Once the number of zeroes are calculated from all matrices involved, they are used for the calculation of the expected values. All values refer to the densities of zeroes in the products $Q_i \times Q_j, i, j = 1 \ldots 4$, where Q_i are the row labels and Q_j are the column labels.

```
Zeros in product:0.042236
expected zeros in prod:0.000000
                  Expected
       Q1        Q2        Q3        Q4
 -------------------------------------------
Q1 0.970138 0.595806 0.888356 0.613196
Q2 0.595806 0.000000 0.134826 0.000103
Q3 0.888356 0.134826 0.002534 0.151077
Q4 0.613196 0.000103 0.151077 0.000000

                   Actual
       Q1        Q2        Q3        Q4
 -------------------------------------------
Q1 0.968750 0.486328 0.881836 0.514648
Q2 0.486328 0.000000 0.075195 0.000000
Q3 0.881836 0.090820 0.586914 0.122070
Q4 0.514648 0.000000 0.103516 0.000977
```

Figure 4.9: Format of the results for the diffusion distinguisher test.

4.6 The cryptographic block profile

The CBPs of the encryption steps are summarised in the module *cpb.c*. Some of the values of the fields were obtained after performing the tests, as presented in Chapter 5, whereas some of the values where easily determined by observing the respective encryption step.

More specifically, the structure defined to hold the CPB information was as follows:

```
typedef struct cbp_struct {
  char *name;                  /* name of the encryption step */
  enum cipher_type f_type;     /* type of the encryption step  */
  unsigned long source_msb;    /* 32 msb of the source */
  unsigned long source_lsb;    /* 32 lsb of the source */
  unsigned long target_msb;    /* 32 msb of the target */
  unsigned long target_lsb;    /* 32 lsb of the target */
  float totdiff;               /* total diffusion */
  float margdiff;              /* marginal diffusion */
  float conf;                  /* confusion */
}cbp;
```

Some of the above fields were updated during the design stage of the encryption steps, but some needed tests to be run. More analytically, the name and type values were defined in the design stage of the encryption steps. The source and target field values can also be determined during the design stage, but they were verified by the diffusion matrices of the encryption steps; it is feasible to assign the values of the source and target fields without any tests since this is merely done in the design stage. However, the round function would not be considered and although for a given source block the round function may have a certain length, not necessarily all source bits may be susceptible to transformation. This may result in having a hidden null block in the source block.

The total diffusion and marginal diffusion were calculated in *testsuite.c*. The total diffusion is expected to be constant for every trial, but the marginal diffusion should vary with a small variance. Therefore the marginal diffusion would be the mean of a number of measured marginal diffusions.

4.7 Security specifications

A list of security specifications were used in order to evaluate a block cipher instance. The security specifications were obtained either from the literature or after interpretation of the experimental results. Consequently, the security specifications described at this point would not be complete.

The security specifications consist of rules which are applied when examining the combination of the encryption steps. Function `evaluate()` in *cbp.c* is the main function for the assessment of the properties of the block cipher under consideration.

The first check is the *compatibility* between two combined encryption steps. More analytically, there should be some indication whether the confusion and diffusion is increased by a super-encryption of two encryption steps. The condition (rule) which accepts that the confusion and diffusion and thus the statistical behaviour of a product cipher is increased, is described by:

- the target of the previous block should be equal to the source of the next block.

This means that if block A is combined with block B in a way that the output of A is the input of B, then it is expected that the diffusion and confusion of the product would be greater than each of the two blocks alone, if the target of A is equal to the source of B. However, if the two values are not the same, they may differ but still their difference may permit a considerable improvement of confusion and diffusion. Function `compare_st()` compares the source and the target between two consecutive encryption blocks and returns the Hamming weight of their difference, i.e. the Hamming weight between $(target)_A \oplus (source)_B$. In addition, a sign was appended to this value to indicate whether the source is greater than the target (positive), or the opposite (negative). **Perfect match** between two encryption steps is when `compare_st(A,B)=0`.

The well known DES algorithm was used as reference point in terms of the statistical properties of the cipher. Starting from a single round DES and gradually increasing the number of rounds, the tests summarise the confusion and diffusion, so some conclusions can be drawn about the number of rounds. The results were also checked for the Blowfish algorithm because its round function is much more stronger than the DES (Appendix B.4). This category of tests assist in determining the minimum number of rounds for a required level of security. Since it is impossible to investigate every possible outcome due to the vast search space, it would be more feasible to accept that the level of security is related to the statistical properties of the block ciphers.

Against this background, a rule concerning the number of rounds is as follows:

- for a homogeneous balanced Feistel network, the minimum number of rounds should be five, given that the total diffusion of a one round is more than 0.25.

This security specification follows from Coppersmith's paper (1996), where an attack could not only distinguish the round function from a random permutation, but it could derive the actual contents of the round function. However, the complexity of the attack is relatively high and it is feasible only for a chosen ciphertext attack. Resistance against such attack was not one of the design objectives of the cryptosystem.

Function `evaluate()` also estimates the confusion of the block cipher instance and revises a weak cipher according to rules which consist of both theoretical findings and experimental conclusions. Therefore, the complete specification of the function is described in Chapter 5 (experimental results).

4.8 Internetworking

A talk utility was developed for a communication between a client end a server, in order to implement the security protocol which involves algorithm negotiation. A talk utility is a simple utility for forwarding messages between two users. With the talk utility the overheads due to the negotiation protocol can be measured. A TCP connection was supported and the port assigned to the connection was 5000:

```
#define MYPORT 5000
```

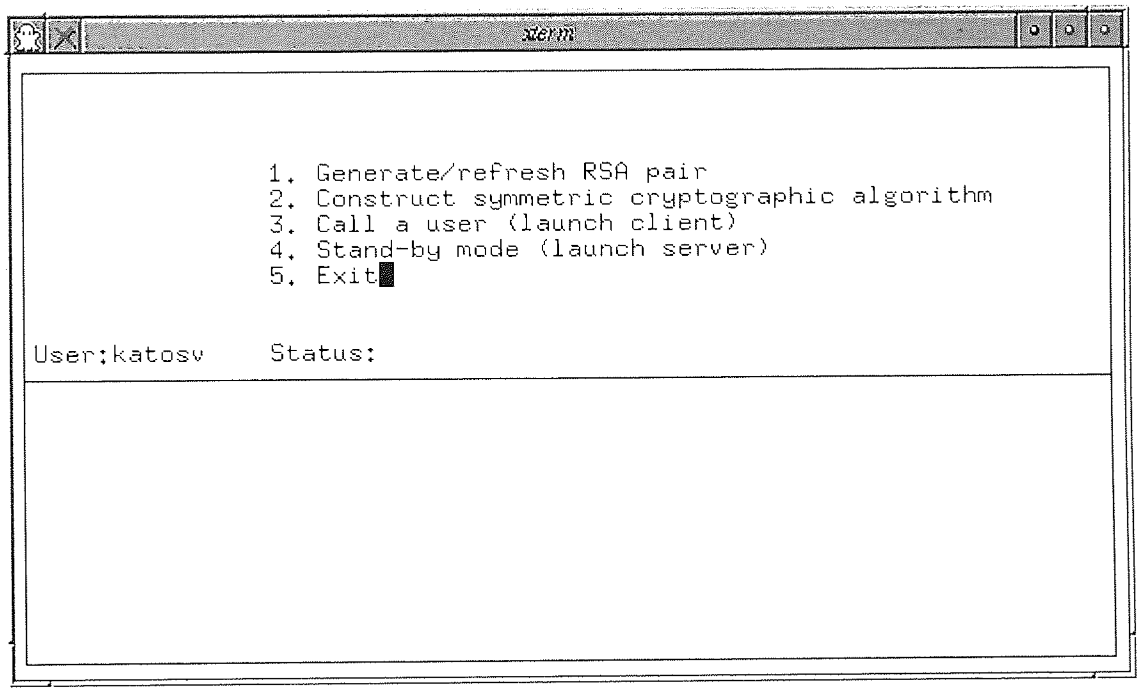
The main module is the *absenttalk.c* which calls either `tsrver()` from *server.c* or `tclient()` from *client.c* (Appendix D).

4.8.1 Talk

The talk framework is required for manipulation of the text screen of a terminal. The library for screen handling used was *curses*. It should be mentioned that the *talk* utility which is used in UNIX platforms is different from the one developed in this project. In the UNIX version there is a talk daemon which *sleeps* and waits for a talk request; once there is a request from a client, a child process is created with `fork()` in order to handle the communication between the two users.

The talk developed in this project does not include a talk daemon, but once the programme is executed the user has to launch the server in order to open the port and wait for the TCP connection request. Another user should run the client and specify the username and Internet address of the user which has launched the server.

Among executing the server and client processes, the talk framework provides a menu to also generate RSA keys and to construct a symmetric block cipher (Figure 4.10).



```
1. Generate/refresh RSA pair
2. Construct symmetric cryptographic algorithm
3. Call a user (launch client)
4. Stand-by mode (launch server)
5. Exit

User:katosv     Status:
```

Figure 4.10: The ABSENTtalk menu

Screen handling

The terminal window is divided into two windows. When there is no communication session, the upper window displays the menu and the lower window is used for any generated messages. The variable type which is specified in *curses* is `WINDOW` and it is a pointer to a window structure:

```
WINDOW *win_a,*win_b,*win_c;
```

In order to switch to the curses mode, the initialisation function must be called. `win_a` was used for the whole X-terminal window, which was later divided into the upper and lower windows, represented by the variables `win_b` and `win_a` respectively, while the last line of the upper window (`win_b`) was a status bar represented by a third window `win_c`:

```
win_a=initscr();
    ⋮
win_b=newwin(LINES/2-2,COLS-2,1,1);
win_a=newwin(LINES/2-2,COLS-2,LINES/2+1,1);
win_c=newwin(1,COLS-2,LINES/2-1,1);
```

where `LINES` and `COLS` are updated with the dimensions of the X-terminal window when `initscr()` is called.

4.8.2 The cryptographic protocol

A public key protocol was required for the exchange of the symmetric block ciphers which would be used during a communication session between the server and the client. More precisely, the protocol consisted of two main parts, namely the setup session and the data exchange session.

During the setup session, the symmetric cipher which was generated by the client, is sent to the server. During this particular stage, the information concerning the structure of the cipher is considered to be as sensitive - in terms of security - as a secret key of a symmetric cryptosystem. The most common scenario for exchanging secret keys is to employ some public key protocol. In this project, the RSA public key cryptosystem was used.

Once the client requests connection with the server by sending the username, the server sends its public key. The client encrypts the symmetric cipher with its private key and then with the server's public key and sends the server the result. The server performs the necessary decryptions and evaluates the proposed symmetric block cipher. If the cipher fulfils the security requirements, the setup session ends and both client and server enter the data exchange session. If the cipher fails to fulfil the server's security requirements, the server refines the cipher and sends it back to the client. The client updates to the revised version of the cipher and both the client and the server enter the data exchange session.

In general there should be no particular reason for the client to reject the revised cipher, since it should be more secure than the previous. If the client was willing to communicate with the server using the previous version in the first place, by default the revised cipher should be of a higher security and thus fulfil the security requirements of the client. If a server rejects the proposed cipher, one should expect that the revised version should have a greater security than the first one. The setup session is presented in Figure 4.11.

There is an obvious need for fast evaluation of the cryptographic strength of a cipher since the evaluation needed to be performed no-line. Moreover, the evaluation criteria must not vary between the participating parties. That is, both parties must accept the same measures of security to correspond to the same cryptographic strength.

```
$A \longrightarrow B : A, P_A$
$B \longrightarrow A : B, P_B$
$A \longrightarrow B : E_{P_B}(*c)$
$B$ : **if** $evaluate(*c)$
**then**
$B \longrightarrow A$ : "*accept*"
**else**
$*c' = f(*c)$
$B \longrightarrow A : E_{P_A}(*c')$
```

Figure 4.11: The protocol for the setup session.

RSA key pair generation

The two modules related to computation of large integers and generation of large primes where modifications of the programmes developed by Cooke (1995). More specifically, *LargeOp.c* consists of functions to add, subtract, multiply, divide, shift left or right, raise to a power mod some integer, calculate square root or compare two large integers.

PrimeTools.c use the previous functions in order to generate large primes, with the function `LargePrimeHunt()`.

According to Cooke (1995), the algorithm for finding a large prime consists of two types of tests. During the first test, an array of small primes is generated. This array holds the values of the first 1000 primes starting from 3. A random odd number is generated as a candidate prime (`Seed`) and the small prime array is also updated with a counter equal to the difference between the candidate prime and the closest multiple of the small prime greater than `Seed`, for all 100 primes.

If any of the counters is zero, then `Seed` is not prime and it is increased by two and the first step runs again. If `Seed` passes the first test, then it is subjected to the second test which is the Solovay and Strassen method for primality testing.

More specifically, the Solovay and Strassen primality test is as follows. Let $b_1, b_2, \dots, b_k$ be k randomly selected integers. For each of those integers b_j, $j = 1, 2, \dots k$ it is checked whether

$$b_j^{(n-1)/2} \equiv \left(\frac{b_j}{n}\right) \ (\mathbf{mod\ n})$$

If any of the congruences fail then n is composite. If all congruences hold, then the probability than n is not composite is $1/2^k$. In *PrimeTools.h* k is the constant `SOLOVAY_ITER` which is assigned with the value 20. This value means that if the primality test passes, the probability that the candidate integer is not prime is less than $(\frac{1}{2})^{20}$, which is very low. The right hand side of the congruence is the Jacobi symbol, defined as:

$$\left(\frac{a}{b}\right) = (-1)^{s_1 \frac{R_1^2-1}{8} + \dots + s_{n-1} \frac{R_{n-1}^2-1}{8} + \frac{R_1-1}{2} \frac{R_2-1}{2} + \dots + \frac{R_{n-2}-1}{2} \frac{R_{n-1}-1}{2}}$$

where R_j are the intermediate steps of the Euler division algorithm and s_i are the respective highest powers of 2 which divide the remainder in every step. Setting $R_0 = a$ and $R_1 = b$, the first step of the division algorithm would be:

$R_0 = R_1 q_1 + 2^{s_1} R_2$

The two modules were used in the module *rsa.c* for generating RSA key pairs and for performing RSA encryptions and decryptions. The function `GenerateRSAPair()`

accepts two large primes p and q and the public value e and attempts to find a suitable secret decryption key d. If it fails, it finds different large primes and the process is repeated until a random message is successfully encrypted and decrypted with the RSA pair e, d. Since the encryption and decryption procedures are identical for a public key cryptosystem, only one function was required for the RSA transformation, namely `RSA()`. This function accepts a message, the key and the modulus, and the message is updated as:

$$(message) = (message)^{(key)} \textbf{ mod n}$$

4.8.3 The server

The server may create an endpoint for communication, by calling the `socket()` function:

```
sock = socket( AF_INET, SOCK_STREAM, 0); /* create a socket */
```

where an Internet (TCP) connection is specified with the `AF_INET` family and `SOCK_STREAM` type of connection is used, since two-way connection byte streams were required.

Once the socket is created, the `bind()` system call binds an address to the local end of the connection, in order to assign a process to that end. Finally, the server waits for a connection request from a client, by the `listen()` and `accept()` system calls.

The server enters the setup protocol as presented in Figure 4.11 when a client connects and the user on the server end agrees to talk with the user on the client end. If the user on the server side refuses to talk to the requesting user on the client side, the request would be rejected and the server would return to the wait state, while the client would be informed that his peer has refused to talk.

After the setup protocol is completed successfully, the server creates two parallel processes which share the created socket- one for sending and one for receiving data. The data read from the standard input would be copied to `win_b` which is the upper half of the X-terminal window and then encrypted using the symmetrical block cipher before they are *written* to the socket, so that the client could read them. Similarly, the data read from the socket, are decrypted and displayed on `win_a` which would be the bottom half of the X-terminal window.

4.8.4 The client

The implementation of the talk client is function `tclient()` in *client.c*. Before this function is called, a symmetric block cipher should have been created. The block cipher information together with the remote user information (remote username and remote host) and the local user are passed to `ctclient()`. The client sends a request to the server by sending the local username and the remote username and waits for confirmation. The client establishes communication with the following steps. First, the Internet address of the server is found:

```
address = inet_addr(host); /* if host is in the Internet notation */
```

where `inet_addr()` is specified in *arpa/inet.h* and returns the Internet address as a `long` integer, given that `host` is a string containing the address in Internet standard notation. In the case where `host` is a DNS entry, `gethostbyname()` defined in *netdb.h* is called instead. If the name is found, then `address` would be a positive integer and `socket()` is called to create a socket. Furthermore the following function is called, in order to enable local reuse of the socket:

```
setsockopt(s,SOL_SOCKET,SO_REUSEADDR,(char *)&on, sizeof(on));
```

where `on=1`. Finally, the connection with the server is established with:

```
connect(s, (struct sockaddr *) &sin, sizeof(sin);
```

which would return a positive integer if the connection was successfully established. It should be noticed that `sin` is a structure which would have the information about the host where the server runs; such information was obtained from either `inet_addr()` or `gethostbyname()` as presented above:

```
struct sockaddr_in {
        short   sin_family;         /* address family */
        u_short sin_port;           /* 2 octet port number */
        struct  in_addr sin_addr;   /* 4 octet IP address */
        char    sin_zero[8];        /* not used */
       }sin;
```

Once the server agrees to start the talk service, the client enters the setup protocol, by accepting the server's public key and sending the symmetric algorithm encrypted with his private key and the server's public key. The client then waits for the server's reply which would either be a new symmetric block cipher, or an indication that the symmetric block cipher is accepted.

Once any block cipher is agreed by the two peers, the client creates two parallel processes- one for the incoming data and one for the outgoing data. Similar to the server, the data from the standard input are copied to `win_b` which handles the upper window of the X-terminal and encrypts them for transmission. The incoming data from the server are *read* from the socket, decrypted and displayed at the lower half window of the X-terminal, which is described by the `WINDOW` variable `win_a`.

The communication between the client and the server is shown at Figure 4.12.

4.9 Concluding remarks

This chapter has described the components used in the prototype. More specifically, the prototype consisted of a number of encryption steps and feedback transformations. For the encryption steps both weak and strong primitives were included, whereas for the feedbacks simple functions were selected. It should be noted that ciphers with feedbacks (i.e. modes of operations) are not examined in depth and are mainly for enabling further research and to increase the search space.

The testsuite is the component of the prototype in which the encryption steps as well as the cipher instances are tested. The tests involve tests for randomness and calculations on the diffusion and confusion matrices in order to measure the confusion

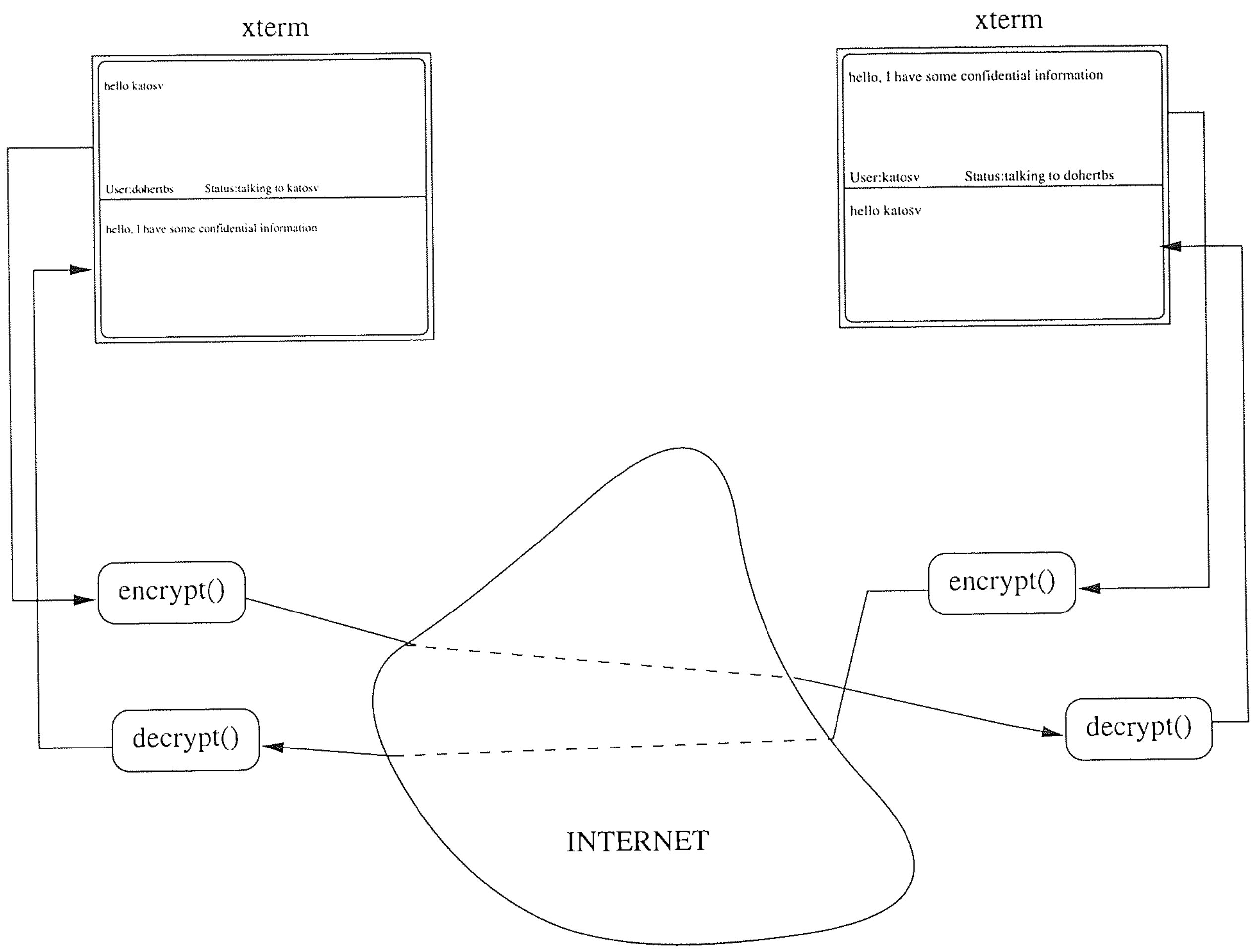

xterm
hello katosv
User:dohertbs Status:talking to katosv
hello, I have some confidential information
xterm
hello, I have some confidential information
User:katosv Status:talking to dohertbs
hello katosv
encrypt()
decrypt()
encrypt()
decrypt()
INTERNET

Figure 4.12: The client/server communication of a talk session.

and diffusion respectively. It was also shown that the diffusion and confusion matrices can be used to generate more results and two new tests - the depth and the diffusion distinguisher test (Chapter 3) - were also developed. The tests performed on a cipher instance measure its security mainly in terms of the confusion and diffusion as defined in Chapter 3, whereas the tests performed on an encryption step provide information for its CBP.

Finally the talk utility was developed to provide experimental evidence on the speed on the cryptographic algorithm negotiation protocol, in which the proposed cryptosystem is intended to be used in. The evaluation step in the protocol would make use of the CBP data which would make the negotiation possible in a short time as required in a communication protocol.

Chapter 5

Experimental results

5.1 Introduction

The sequence of the experiments was as follows:

- First the characteristics needed for the CBPs were computed. That is, each encryption step was examined individually, by constructing block ciphers consisting of only one step.

- Second, homogeneous block ciphers were examined. Starting from one round and gradually increasing the number of rounds using the same encryption step, the parameters were computed.

- Third, combinations between the different encryption steps were examined. The compatibility between two cascading blocks was the comparison between the target and source fields of these two blocks in their respective CBPs, as described in Chapter 4.

- Finally, the results between the different product ciphers, combined with the compatibility between the source and target for every product was summarised and compared in order to derive an approximation function for the confusion and diffusion. This forms the evaluation algorithm implemented by the `evaluate()`

function in *cbp.c*.

5.2 Cryptographic Block Profile

A CBP summarises some parameters of the underlying encryption step. Some of them were updated without any tests, whereas some require experimental results. The name and type fields were updated without any computation involved. The source and target fields were obtained from the diffusion matrix as follows:

```
for each encryption step N do
 if type of N is not ``permutation'' or ``vigenere'' then
   define cipher:=name.N
   D:=diffusion matrix
   source:=rows of D where no. of ones >1
   target:=columns of D where no.of ones >1
  end
```

The positions of rows and columns are represented by the bits of the large integers, where element d_{00} of the matrix is mapped to the most significant bit of the source and target *_msb* integers.

In the case of the DES encryption step, the diffusion matrix would be:

$$\mathcal{D} = \begin{bmatrix} 0_{32} & I_{32} \\ I_{32} & A_{32}(0.19) \end{bmatrix}$$

where:

$$\mathbf{A}_{32}(0.19) = \begin{bmatrix}
0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&1&0&0&0&1&0&1&0&0&0&0&1&1&0&1&0&0&1&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&1&0&0&0&1&0&1&0&0&0&0&1&1&0&1&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&1&0&1&0&0&0&1&0&1&0&0&0&1&0&0&0\\
0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&1&0&1&0&0&0&1&0&1&0&0&0&1&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0\\
0&1&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&1&0&1&0&0&0&1&0&1\\
0&1&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&1&0&1&0&0&0&1&0&1\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&1\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&1\\
0&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&1&0&1&0&1&0&0&0&1\\
0&1&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&1&0&1&0&1&0&0&0&1\\
0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&1&1&0&0&0&0&1&0&0&1&0&0&1&0&0&0&0&1&0&0&0&0&0&1&0&0&0&1&0\\
0&0&0&1&1&0&0&0&0&1&0&0&1&0&0&1&0&0&0&0&1&0&0&0&0&0&1&0&0&0&1&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&1&0&1&0&0&1&0&0&0&1&0&1&0&0&0&1&0&0&0&0&0&0&0&0&0&1&1&0\\
0&0&0&0&1&0&1&0&0&1&0&0&0&1&0&1&0&0&0&1&0&0&0&0&0&0&0&0&0&1&1&0\\
0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0\\
1&0&1&0&0&0&0&1&1&0&0&1&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&1&0&0&0\\
1&0&1&0&0&0&0&1&1&0&0&1&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&1&0&0&0\\
1&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&1&0&0&0&1&0&0&0&1&1&0&0&0&0&1&0&0&1&0&0&0&0&0&1&0&0&0&0&0\\
1&0&0&1&0&0&0&1&0&0&0&1&1&0&0&0&0&1&0&0&1&0&0&0&0&0&1&0&0&0&0&0
\end{bmatrix}$$

The number in the brackets denotes the density of ones of the matrix (i.e. the ratio of total number of ones over the total number of entries), which is characteristic of the DES round function. The diffusion matrices of all the encryption steps are presented in Appendix B.4.

The total diffusion was invariant for every encryption step since the final value converged after 12 encryption steps on average. Yet the marginal diffusion inherently depended on the first two encryptions hence on the random input values, so a mean was calculated. More specifically, the marginal diffusion was the mean value of a set of

300 marginal diffusions. In all cases the variance was sufficiently small ($exp(-5)$); this denotes that for every independent test with uniformly random inputs would produce values which are likely to pass the hypothesis $H_0 : m_s = m_t$, where m_s and m_t are any two marginal diffusions calculated with uniformly distributed random inputs. The following `testsuite` script was executed in order to assess the marginal diffusion:

```
edit +1 4        %construct a one round DES
matlab=desone.   %open output file for raw data
repeat=300       %300 loops
margdiff         %compute marginal (and total) diffusion
next             %close loop
```

which generated 300 files containing total diffusion and marginal diffusion values. All files were concatenated into one file, producing a 300×2 array, with the first column a constant (diffusion) and the second column the marginal diffusion values. The mean of the latter would be the marginal diffusion assigned to the CBP of the underlying encryption step.

The confusion was determined after 8000 encryptions; the following script was executed:

```
matlab=desone.   %open output file for raw data
input=plain      %assign plaintext as random input
loops=8000       %number of encryptions
block            %compute probabilities of input/output bit pairs
```

The results for all tests for the encryption steps are summarised at Appendix B.5.

5.3 Homogeneous ciphers

5.3.1 The DES encryption steps

Firstly, the parameters and statistical properties of the full DES were computed, in order to provide an insight of what values a "good" cipher should have.

Table 5.1: The product of one-round DES encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion
1	0.0625	0.0104	$1.7970e^{-6}$	0.0415
2	0.3206	0.0728	$3.5784e^{-5}$	0.2906
3	0.7349	0.1760	$8.5010e^{-5}$	0.7062
4	0.9690	0.2394	$6.6771e^{-5}$	0.9568
5	1	0.2502	$4.7094e^{-5}$	0.9996
6	1	0.2498	$4.4160e^{-5}$	0.9999
7	1	0.2497	$4.2048e^{-5}$	0.9999
8	1	0.25	$4.5161e^{-5}$	0.9999

For more than eight rounds, the results are similar to those of the sixth round. The confusion is actually the mean value of 8000 trials. It should be also noticed that the variance of the marginal diffusion is very small. This would imply that one could independently measure the diffusion and obtain results close to those presented at Table 5.1.

5.3.2 The Blowfish encryption steps

`Blowfish L` and `Blowfish R` produce the same values for diffusion and confusion. Their differences are the *source* and *target* values which are complementary. Therefore in a product cipher the impact of these values to the confusion and diffusion of the cipher could be examined.

Table 5.2: The product of the two Blowfish encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion
1	0.2656	0.0623	$1.7290e^{-6}$	0.2492
2	0.7578	0.1864	$5.1442e^{-5}$	0.7454
3	1	0.2498	$8.2240e^{-5}$	0.9960
4	1	0.2506	$4.4845e^{-5}$	0.9998
5	1	0.2501	$4.9253e^{-5}$	0.9999
6	1	0.2499	$4.5010e^{-5}$	0.9999

5.3.3 The Balanced Feistel Network steps

Due to the source-target symmetry, it is expected that one round affected one quadrant of the confusion matrix, two rounds should affect three quadrants and three rounds all quadrants.

Table 5.3: The product of the Feistel #1 encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion
1	0.2656	0.0624	$1.0545e^{-6}$	0.2500
2	0.7578	0.1875	$3.6743e^{-5}$	0.7499
3	1	0.2505	$4.7070e^{-5}$	0.9999
4	1	0.2498	$4.8624e^{-5}$	0.9999
5	1	0.2505	$4.5241e^{-5}$	0.9999
6	1	0.2492	$4.4861e^{-5}$	0.9999

Table 5.4: The product of the Feistel #2 encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion
1	0.2656	0.0623	$1.1523e^{-6}$	0.2500
2	0.7578	0.1872	$3.3162e^{-5}$	0.7499
3	1	0.2498	$4.3145e^{-5}$	0.9999
4	1	0.2505	$4.8620e^{-5}$	0.9999
5	1	0.2502	$4.4710e^{-5}$	0.9999
6	1	0.2501	$4.7126e^{-5}$	0.9999

The results confirm that indeed one round affected one quadrant (confusion equal to 25%), two rounds affected three quadrants (confusion equal to 74.9% $\approx$ 75%) and three rounds affected all quadrants (confusion equal to 99.99% $\approx$ 100%).

5.3.4 The Unbalanced Feistel Network steps

The target heavy and source heavy encryption steps defined produce similar results, i.e. gradual increase of the diffusion and confusion values. An additional column has been added, which is the comparison between the target and source of two consecutive encryption blocks which are the same encryption step (homogeneous case). The comparison values are also used later on the regression analysis. It should be noted that the comparison is non zero; this was expected since the Feistel is not balanced, so the output target could not match with the input source. This resulted in requiring more rounds to achieve complete diffusion.

Table 5.5: The product of the target heavy 16:48 UFN encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion	comp.
1	0.2031	0.0462	$1.5329e^{-6}$	0.1858	-
2	0.4492	0.1092	$2.5695e^{-5}$	0.4355	-32
3	0.6953	0.1716	$3.4261e^{-5}$	0.6853	-32
4	0.9414	0.2347	$4.2756e^{-5}$	0.9354	-32
5	1	0.2505	$5.0069e^{-5}$	0.9994	-32
6	1	0.2499	$5.0990e^{-5}$	0.9999	-32
7	1	0.2502	$4.6150e^{-5}$	0.9999	-32
8	1	0.2498	$4.6139e^{-5}$	0.9999	-32

Table 5.6: The product of the source heavy 40:24 UFN encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion	comp.
1	0.2500	0.0586	$1.1486e^{-6}$	0.2337	-
2	0.6191	0.1518	$2.8222e^{-5}$	0.6082	16
3	0.9102	0.2262	$4.2726e^{-5}$	0.9053	16
4	1	0.2500	$4.3462e^{-5}$	0.9996	16
5	1	0.2505	$4.2442e^{-5}$	0.9999	16
6	1	0.2495	$4.3668e^{-5}$	0.9999	16
7	1	0.2498	$5.2079e^{-5}$	0.9999	16
8	1	0.2497	$5.0140e^{-5}$	0.9999	16

It can be seen from these two tables that the greater the absolute value of the comparison, the more rounds are needed to obtain maximum diffusion.

5.4 Regression analysis

Regression analysis is extensively used to model experimental data, in order to identify trends and influence of parameters. In this thesis the relation between the diffusion of a product cipher and the encryption blocks has been established theoretically. However, for the confusion a relation is calculated from the experimental results. This is done by applying linear regression techniques.

Three approaches were tested, depending on the type of the product cipher. The first approach considered homogeneous product ciphers. The second approach applied linear regression to heterogeneous product ciphers with two rounds. The third approach consisted of data from heterogeneous product ciphers with two or more rounds.

For the first approach, it is assumed that in a product cipher, every round contributes to the confusion by taking the value from the previous block and increasing the confusion towards one by its characteristic elasticity (Chatterjee & Price 1977). The increment of confusion is due to cryptographic composition and the slope of its increment is related to the round function and the topology of the blocks of bits (elasticity) in which they are combined with the round function.

Since we are dealing with product encryption, it would be reasonable to attempt to establish a relation between the consecutive blocks. Starting with the DES encryption steps, if y_t denotes the confusion after t DES steps, a scatter plot between y_t and y_{t-1} is as shown in Figure 5.1.

It can be seen that the relation is not linear. Yet a standard assumption in regression analysis is that a model describing the data is linear (Chatterjee & Price 1977). Therefore it is required to apply transformations in order to linearise the data. A number of transformations are available in the literature and could be tested against the experimental data in order to establish which of the functions fits better to these

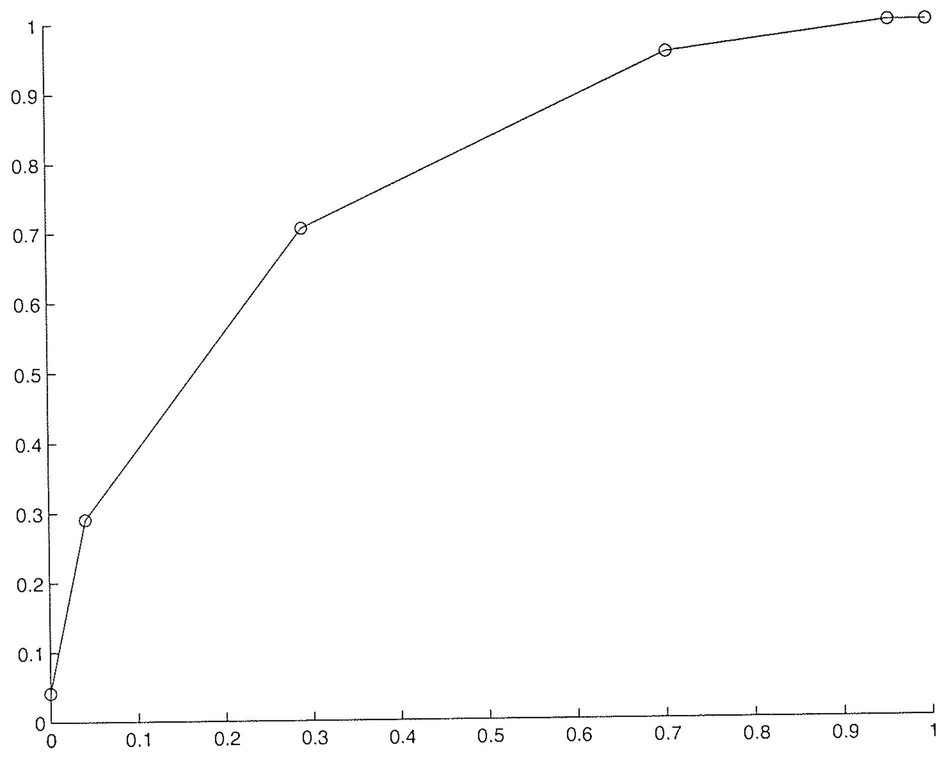


Figure 5.1: Scatter plot of y_t against y_{t-1} for $1 \leq t \leq 16$ for the DES encryption step.

data. These transformations are used extensively in Econometrics (Chatterjee & Price 1977); several alternatives are presented in Table 5.7, where for each function, the transformation which linearises it is also shown.

Table 5.7: Linearisable functions with corresponding transformations (source: Chatterjee & Price 1977).

Function	Transformation	Linear form
$y = \alpha x^{\beta}$	$y' = \log y, x' = \log x$	$y' = \log \alpha + \beta x'$
$y = \alpha e^{\beta x}$	$y' = \ln y$	$y' = \ln \alpha + \beta x$
$y = \alpha + \beta \log x$	$x' = \log x$	$y = \alpha + \beta x'$
$y = \dfrac{x}{\alpha x - \beta}$	$y' = \dfrac{1}{y}, x' = \dfrac{1}{x}$	$y' = \alpha - \beta x'$
$y = \dfrac{e^{\alpha+\beta x}}{1+e^{\alpha+\beta x}}$	$y' = \ln(\dfrac{y}{1-y})$	$y' = \alpha + \beta x$

5.4.1 Linear regression of DES

An Ordinary Least Squares estimation was performed for the above functions using the confusion values for different number of steps and the results are summarised in Table 5.8. It should be noted that all logarithms were on base e and that $y = y_t$ and $x = y_{t-1}$.

Table 5.8: Ordinary Least Squares estimation of the DES.

Function	α	β	$^{\dagger}R^2$	Probability α	Probability β
$y_t = \alpha y_{t-1}^{\beta}$	1.01	0.37675	0.98588	0.290	0.000
$y_t = \alpha e^{\beta y_{t-1}}$	2.62	0.98756	0.82209	0.000	0.000
$y_t = \alpha + \beta \log y_{t-1}$	1.0022	0.22471	0.99705	0.000	0.000
$y_t = \dfrac{y_{t-1}}{\alpha y_{t-1} - \beta}$	0.90405	-0.10593	0.99581	0.000	0.000
$y_t = \dfrac{e^{\alpha+\beta y_{t-1}}}{1+e^{\alpha+\beta y_{t-1}}}$	-2.2721	11.2593	0.94863	0.005	0.000

†coefficient of determination

The quantity R^2 is the coefficient of determination which is a measure of "goodness of fit", measures the proportion or percentage of the total variation in the dependent variable explained by the regression model (Gujarati 1988).

The probabilities presented on the last two columns of the table, present the significance in which the independent variable influences the dependent variable. More analytically, for every coefficient the probability presents whether the underlying coefficient passed the hypothesis test of being equal to zero (t test):

H_0 : $\beta = 0$: the corresponding independent variable does not explain some of the variation of the dependent variable $[p > 0.05]$

H_a : $\beta \neq 0$: the corresponding independent variable explains some of the variation of the dependent variable $[p < 0.05]$

In terms of interpretation, if the probability is less than 0.05, the coefficient of the underlying independent variable is not zero and thus influences the dependent variable. In the opposite case, when the probability is greater than 0.05, the independent variable does not influence the dependent variable and should be excluded from the model. Therefore the objective in a regression analysis is to obtain high R^2 and zero - or less than 0.05 - probabilities.

From the above results it could be seen that $y_t = \alpha + \beta \log y_{t-1}$ provided the best fit, since this model would be capable of explaining 99.7% of the experimental data. The remaining functions were also good candidates for modelling the confusion, since the probabilities were low or zero. Although the probability for the first equation ($y_t = \alpha y_{t-1}^{\beta}$) was high (29%), it remains a good candidate since that probability refers to the constant.

The DES is an example of a homogeneous balanced Feistel network. The regressions of the remaining homogeneous Feistel networks implemented in this project are described in Appendix B.3. The relations derived for DES and general Feistel networks show ability to estimate confusion.

Because the cipher space contains more heterogeneous than homogeneous Feistel networks, it would be more useful to establish a confusion model for heterogeneous Feistel networks. By doing this, one would be able to estimate the confusion of a product cipher of two or more encryption steps. This is addressed in the next section.

5.4.2 Heterogeneous product ciphers

The equations presented above are useful for homogeneous ciphers or homogeneous parts of a heterogeneous cipher. That is because the elasticity was determined using the same encryption step. However, one could not determine with the same experimental approach the elasticities for all combinations, first because it is computationally infeasible and second, if all tests could be performed, there would be no reason to model the confusion since one would have the actual values.

The problem for using the homogeneous models for the heterogeneous case can be seen by reference to Table 5.9.

Table 5.9: Confusion of two round heterogeneous product ciphers.

first round		second round		result
0.0415	(DES)	0.2500	(Feistel 1)	0.5415
0.0415	(DES)	0.2492	(Blowfish R)	0.5377
0.2500	(Feistel 1)	0.0415	(DES)	0.5415
0.2492	(Blowfish R)	0.0415	(DES)	0.5407
0.2500	(Feistel 1)	0.2492	(Blowfish R)	0.7461
0.2492	(Blowfish R)	0.2500	(Feistel 1)	0.7492

Consider for example the third combination, the `Feistel 1` with `DES`. If it is assumed that the final confusion is the confusion produced by the `Feistel` and influenced by the last block which is `DES`, it should be:

$y_t = \alpha + \beta \log y_{t-1}$

where $\alpha = 1.0022$, $\beta = 0.22471$ (Table 5.8) and $y_{t-1} = y_0 = 0.0415$. However, this function would estimate that the confusion after the two rounds should be $y_1 = 0.69$, which is quite different from the experimental 0.5415.

Table B.12 in Appendix B summarises all tests for encryption steps with confusions ranging from 0.03 to 0.25. These tests where performed on balanced Feistel Networks with `compare_st(s1,s2)=0`. To determine the relation between the confusion of the product cipher and its encryption blocks, a three dimensional plot is constructed. More specifically, a scatter plot of the triples (x, y, z) where x is the confusion of the first encryption step, y is the confusion of the second and z is the resulting confusion would appear as in Figure 5.2.

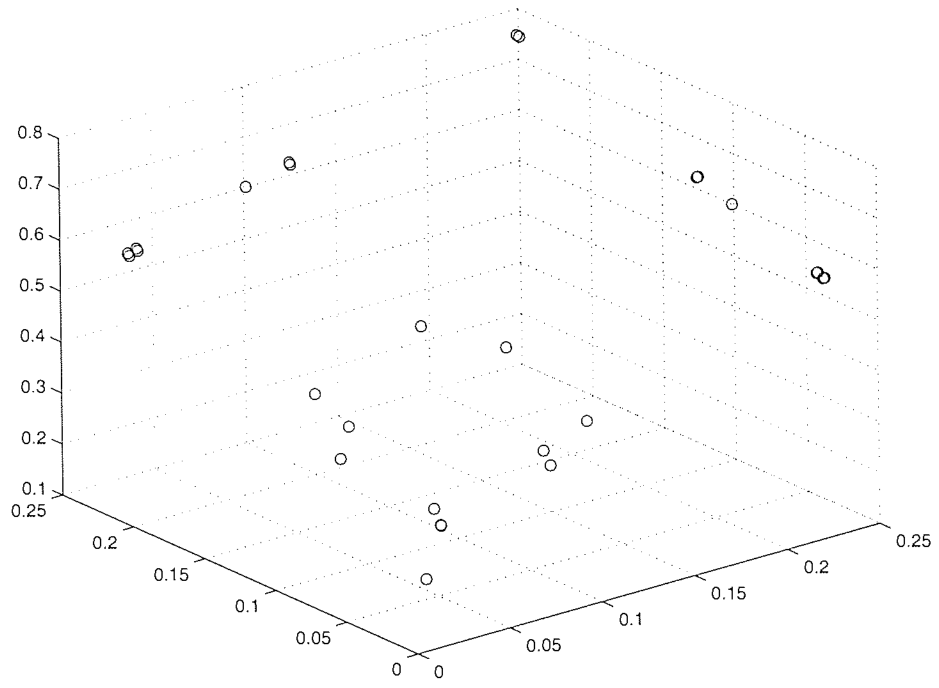


Figure 5.2: Scatter plot of confusion of two round balanced heterogeneous Feistel Networks.

It can be seen that a pattern emerges with two main characteristics:

- there is a symmetry of the distribution of points around the vertical plane $x = y$.
- the surface in which the points lie resembles the density of zeros of a product matrix (Appendix A), see Figure 5.3.

The second observed characteristic is very crucial in determining the approximation function of the confusion. More specifically, the number of expected zeros of a matrix resulting from a product of two (square) matrices with densities of zeros (uniformly distributed) p_1 and p_2 respectively is (Appendix A):

$$E(p) = (p_1 + p_2 - p_1 p_2)^n \tag{5.1}$$

where n is the dimension of the matrix. A plot of the surface defined by equation (5.1) is presented in Figure 5.3.

Indeed the correlation between the actual and expected confusion was found to be equal to 0.9833, for $n = 1$. A regression of the following equation:

$p_3 = m + ap_1 + bp_2 + cp_1p_2$

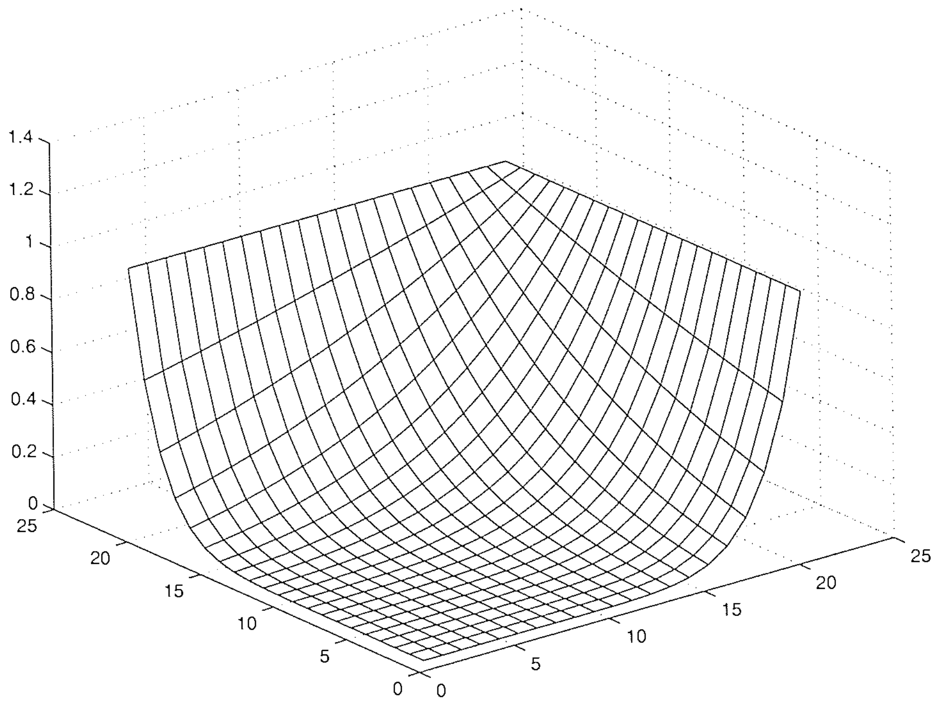


Figure 5.3: Surface plot of $(p_1 + p_2 - p_1p_2)^n$, for $n = 10$.

provided the results below:

$$p_3 = 0.12419 + 1.52p_1 + 1.5738p_2 - 2.4293p_1p_2 \tag{5.2}$$

$$[prob.] \quad [.000] \quad [.000] \quad [.000] \quad [.003]$$

$$R^2 = 0.96286$$

It should be noted that equation (5.2) of the confusion is valid only where there is perfect match, i.e. `compare_st(A,B)=0`.

However, the `compare_st(A,B)` rule can not be applied on encryption steps other than Feistel steps, because confusion and diffusion may decrease. This is illustrated with the following example where a permutation (transposition) is inserted between two perfect matching (Feistel) blocks. This is a typical example where increment of complexity may lower the strength. Consider the following cipher instance:

```
DES->Feistel#1->DES
```

which has confusion and diffusion of 0.9998 and 1 respectively. If an extra round is added between the first two blocks, i.e.:

`DES->Permute#1->Feistel#1->DES,`

the confusion and diffusion would be 0.8854 and 0.8887 respectively. However, if the diffusion was 1 at the beginning of the permutation block, there would be no decrease, since every entry in the diffusion matrix would be equal to one. This observation generated the following rule:

- If diffusion is less than one and greater than zero, only "feistel-type" of encryption blocks are allowed.

The "greater than zero" implies that the cipher may begin with any type of encryption step.

However, for more than two rounds equation (5.2) fails to estimate the confusion. Therefore more parameters were considered, introduced and examined. More specifically, the marginal diffusion was considered, which is an indication of the ability of an encryption step to change (increase) its diffusion. Marginal diffusion is inherently related to the confusion, since the latter is also a measure of change of output bits with respect to the input bits.

After a series of trial and error selection of parameters, the best approximation function found was of the form:

$$\begin{aligned} p_3 =& a + bp_1 + cp_2 + dp_1p_2 + ep_1m_1 + fp_1m_2 + gp_2m_2 + hp_2m_{f_1} + ix_1m_{f_1} + \\ & jd_2/m_2 + kp_1p_2m_{f_1} \end{aligned} \tag{5.3}$$

for block ciphers with more than two rounds and with `|compare_st(A,B)|=abs(16)` between the last two rounds. The coefficients as well as their probabilities are presented at Table 5.10. Since there are more than one independent variables, the parameter which denotes the level in % by which the regression equation explains the variability of the dependent variable, is the $\bar{R}^2$, which is the *determination coefficient corrected for the degrees of freedom.*

Equation (5.3) was a result of a trial-and-error approach. Econometrics suggest that when adopting such an approach, diagnostic tests should be also performed in

Table 5.10: Regression results for heterogeneous product cipher with source/target difference equal to 16 (eq. 5.3).

coefficient	value	[probability]
a	-54.4435	[0.000]
b	-5.9836	[0.002]
c	176.8139	[0.000]
d	-408.512	[0.000]
e	-3.2549	[0.000]
f	1796.0	[0.000]
g	-1486.1	[0.000]
h	-14.8299	[0.011]
i	168.3958	[0.000]
j	7.9446	[0.000]
k	-879.9524	[0.000]
$R^2 = .97398$	$\bar{R}^2 = .95952$	

order to establish whether the equation can be used for further estimations. In these tests it is required that the probabilities are greater than 0.05.

More specifically, in evaluating the estimated functions, the following diagnostic tests - hypotheses - were employed (Green 1993):

I: Specification of function (F test)

H_0 : Correct functional form $[p > 0.05]$

H_a : Incorrect functional form $[p < 0.05]$

II: Normality of the error term (χ^2 test)

H_0 : Error term is distributed normally $[p > 0.05]$

H_a : Error term is not distributed normally $[p < 0.05]$

III: Heteroscedasticity (Gujarati 1988) of the error term variance (F test)

H_0 : Error term variance is homoscedastic $[p > 0.05]$

H_a : Error term variance is heteroscedastic $[p < 0.05]$

IV: Stability of regression coefficients (F test)

H_0 : Regression coefficients are stable with an increase in the number of observations [$p > 0.05$]

H_a : Regression coefficients are not stable with an increase in the number of observations [$p < 0.05$]

V: Multicollinearity (correlation coefficients among independent variables and comparison with R)

H_0 : The independent variables are not collinear [$p > 0.05$]

H_a : The independent variables are collinear [$p < 0.05$]

VI: Goodness of fit of regression equation (F test)

H_0 : $R^2 = 0$ {the relationship is not linear} [$p > 0.05$]

H_a : $R^2 > 0$ {the relationship is linear} [$p < 0.05$]

[instead for the coefficient of multiple determination R2 the adjusted for degrees of freedom R2 is also used. Closer to 1 means better fit.]

VII: Predictive power of the regression equation (Theil's indexes) (Theil 1971)

- ME = Mean error (should be close to 0)
- RC = Regression coefficient of actual on predicted values (should be close to 1)
- U = Theil's inequality coefficient (should be close to 0; 0 = perfect prediction, 1 = completely bad prediction)
- UM = Fraction of error due to bias (should be close to 0)
- US = Fraction of error due to different variation (should be close to 0)
- UC = Fraction of error due to difference covariation (should be close to 1)

The diagnostic tests of (5.3) are summarised in Table 5.11.

Table 5.11: Diagnostic tests for eq. (5.3).

I.	$F(1,17) = 0.13974$	[0.713]
II.	$\chi^2(2) = 0.62062$	[0.733]
III.	$F(1,27) = 0.85379$	[0.364]
IV.	$F(2,16) = 0.47508$	[0.630]
V.	No correlation coefficient of independent variables greater than R	
VI.	$F(10,18) = 67.3755$	[0.000]
VII.	$ME = 0.00022$ $RC = 1.00035$ $U = 0.017754$ $UM = 0.00005$ $US = 0.00695$ $UC = 0.99300$	

Note: Figures in parentheses are degrees of freedom and figures in brackets are probabilities.

From the diagnostic tests above it seems that equation (5.3) is correctly specified and it has very high predictive power.

For the case of the product cipher where the comparison for the last two blocks was equal to `abs(8)`, the following approximation function provided the best results:

$$\begin{aligned} p_3 = {} & a + bp_1 + cp_2 + dp_1p_2 + ep_1m_1 + fp_1m_2 + gp_2m_2 + hp_1m_{f_1} + \\ & ip_2m_{f_1} + jd_1/m_1 \end{aligned} \tag{5.4}$$

where the coefficients as well as the probabilities are summarised at Table 5.12.

The results from the diagnostic tests are presented in Table 5.13.

From the diagnostic tests above it seems that equation (5.4) is correctly specified and it has very high predictive power.

Table 5.12: Regression results for heterogeneous product cipher with source/target difference equal to 8 (eq. 5.4).

coefficient	value	[probability]
a	-0.090831	[0.744]
b	0.99873	[0.000]
c	-13.7654	[0.001]
d	-146.1946	[0.004]
e	-1.0970	[0.001]
f	585.1066	[0.004]
g	173.6688	[0.003]
h	-5.4003	[0.000]
i	23.6309	[0.000]
j	0.27484	[0.000]

$R^2 = .99269 \quad \bar{R}^2 = .98831$

Table 5.13: Diagnostic tests for eq. (5.4).

I.	$F(1,17) = 2.4914$	[0.137]
II.	$\chi^2(2) = 1.1162$	[0.572]
III.	$F(1,27) = 0.00859$	[0.927]
IV.	$F(2,16) = 0.67755$	[0.525]
V.	No correlation coefficient of independent variables greater than R	
VI.	$F(9,15) = 226.4509$	[0.000]
VII.	$ME = 0.00000 \quad RC = 1.00001 \quad U = 0.005245$ $UM = 0.00000 \quad US = 0.00184 \quad UC = 0.99816$	

Finally, for the case of the product cipher where the comparison for the last two blocks was equal to 0, the following approximation function provided the best results:

$$\begin{aligned} p_3 =& a + bp_2 + cm_{f_1} + dm_1m_2 + ep_2m_1 + fd_2m_2 + gp_1m_{f_1} + \\ & hd_1 + id_1d_2 \end{aligned} \tag{5.5}$$

The coefficient values as well as the probabilities are presented in Table 5.14. The diagnostic tests are summarised in Table 5.15. From the diagnostic tests it seems that equation (5.5) is correctly specified and it has very high predictive power.

Table 5.14: Regression results for heterogeneous product cipher with source/target difference equal to 0 (eq. 5.5).

coefficient	value	[probability]
a	-42.7047	[0.015]
b	1301.7	[0.014]
c	11.2914	[0.0001]
d	24341.9	[0.019]
e	-6035.2	[0.020]
f	-17026.9	[0.014]
g	-11.8173	[0.001]
h	0.76361	[0.010]
i	-10.5354	[0.052]

$R^2 = .96195 \quad \bar{R}^2 = .92812$

Table 5.15: Diagnostic tests for eq. (5.5).

I.	$F(1,8) = 3.5868$	[0.095]
II.	$\chi^2(2) = 0.78064$	[0.677]
III.	$F(1,16) = 7.9856$ Shows some heteroscedasticity:	[0.012]
IV.	$F(2,9) = 4.4038$	[0.046]
V.	No correlation coefficient of independent variables greater than R	
VI.	$F(8,9) = 28.4375$	[0.000]
VII.	$ME = 0.00420$ $RC = 1.00244$ $U = 0.015539$ $UM = 0.02279$ $US = 0.01207$ $UC = 0.96513$	

Regression of the marginal diffusion

In the above relations for the estimation of confusion, the marginal diffusion is also required. It was found that the marginal diffusion followed a linear relation with $\log(p_3)$, i.e. the corresponding confusion, for all three classes of target-source comparisons. That is, the relation of m_3 and p_3 was defined by

$$m_3 = a + b\log(p_3) \tag{5.6}$$

Table 5.16 summarises the coefficients as well as the significance levels, for the three comparison classes. It can be seen that the coefficients increase with the increment of the comparison value. Therefore, for intermediate values of comparison, the value of the marginal diffusion could be derived by interpolation.

Table 5.16: Regression results for marginal diffusion for equation (5.6).

`|compare_st()|`	coefficient	value	[probability]	R^2
0	a	0.24722	[0.000]	
	b	0.19418	[0.000]	0.98624
8	a	0.24743	[0.000]	
	b	0.19645	[0.000]	0.99056
16	a	0.24824	[0.000]	
	b	0.22469	[0.000]	0.97697

5.5 Graphical representation of confusion and diffusion

The confusion matrix was described in a previous chapter as a square 64×64 matrix with the entries describing the number of the output bits changing given that the corresponding input bit has changed, normalised to the range $[0, 1]$. The actual value of confusion was the deviation from the ideal 1, due to the sum of errors which in turn were the squares of differences between the actual values and 0.5, the expected value.

For a statistically *good* cipher, the confusion should be equal to or greater than 0.9999, for all keys. If the confusion had such a value for a cipher, then it would be likely to resist linear cryptanalysis. Confusion of 0.9999 would imply that if any input bit changed, then all output bits would change with a probability of 0.5. Such value of confusion would restrict the diffusion to be equal to 1 since in any other case- i.e. if the diffusion was lower than 1- there would be *at least* one pair of input/output bits which would not be related, so the value of confusion would be at most $1 - 0.0002 = 0.9998$. Thus, a linear relationship with higher probability would emerge.

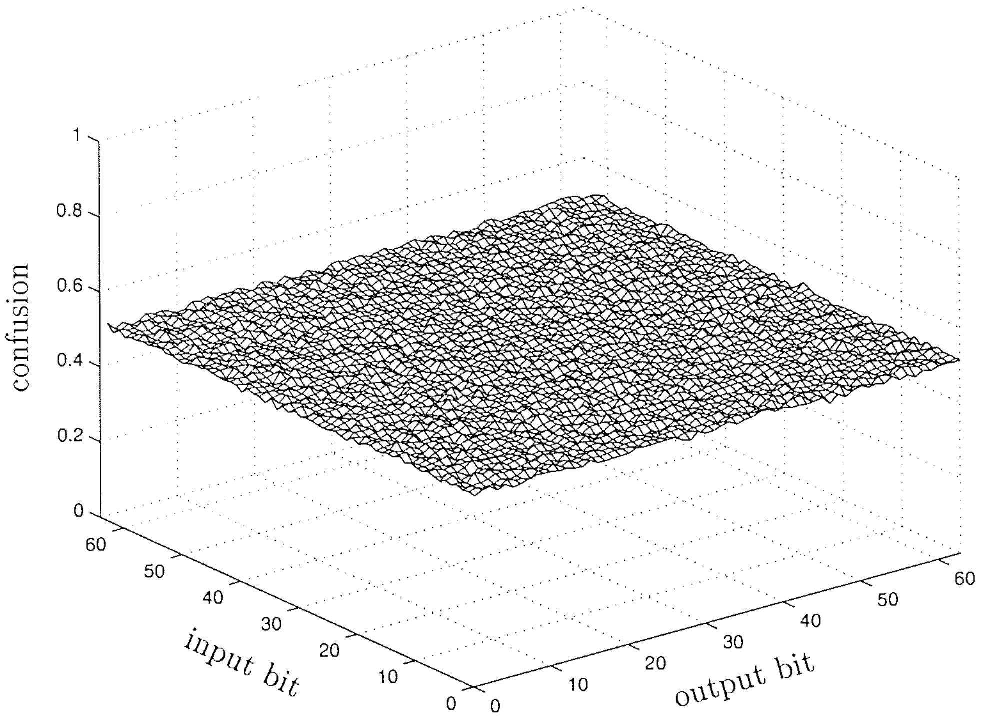


Figure 5.4: Confusion matrix of the full DES algorithm.

The confusion matrix is represented as a surface as shown in Figure 5.4 which is a plot of the values of the matrix. The sixteen round DES would produce a confusion of 0.9999 so any linear relation would not be apparent. Yet it is known from the literature that after extensive analysis, a linear relation was found. Consequently, although this approach may safely conclude that a cipher is weak, the opposite is not necessarily true.

Similarly, the Ψ matrices used for the calculation of the diffusion matrix would produce a surface as in Figure 5.5. Again, this is the desired surface for a potentially strong cipher; the 64 Gaussian distributions should all have mean equal to 32.

5.5.1 The round function

It has been argued by Luby and Rackoff that a round function should have properties of a pseudorandom bit generator. In that case, three rounds of such construction would have a small distinguishing probability from a random permutation generator.

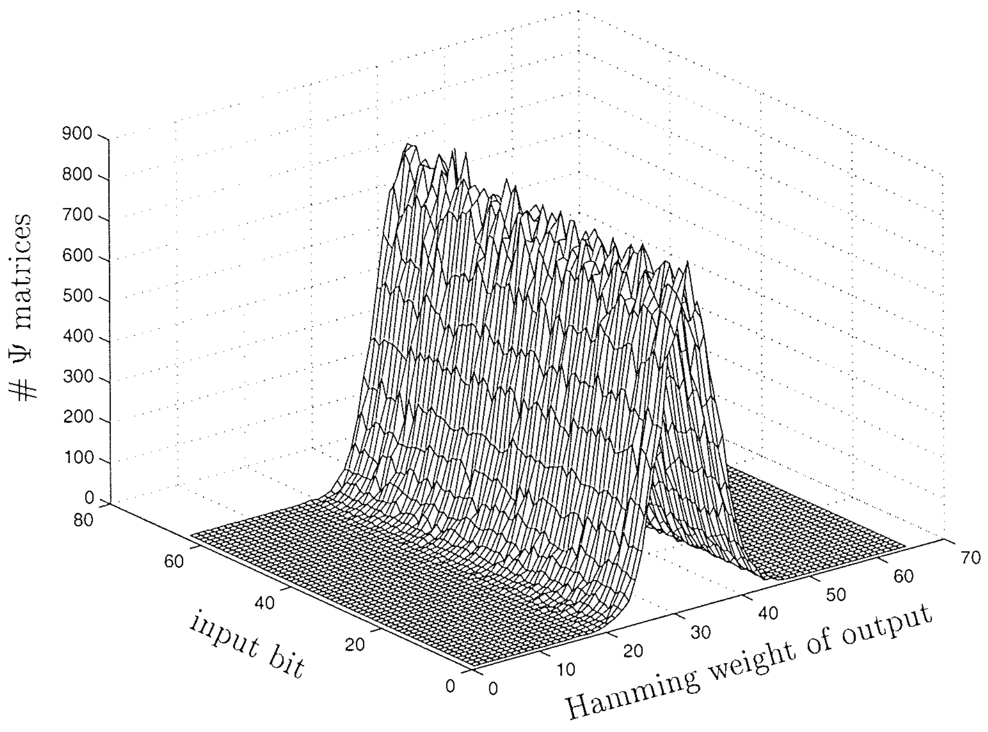


Figure 5.5: Ψ matrix distribution of diffusion of the full DES algorithm.

Assuming that the above is applied for a balanced 64 bit (homogeneous) Feistel Network, the confusion matrix graph for the desired function should appear as in Figure 5.6. The linear regions and the region where the round function (the surface on average height of 0.5) is applied could be clearly distinguished. If a function behaves as a random bit generator, then for every input to that function, all outputs should have the chance to be affected. The confusion would then be equal to 0.25, which is due to the first quadrant- i.e. the quadrant where the round function operates- whereas there linear regions (the second and fourth quadrants) do not offer any ambiguity, since the relations defined hold with probability 1. The total diffusion would be more than 0.25; in fact, it would be equal to 0.2656. That is the result of the 0.25 due to the first quadrant, plus the two sets of 32 bits of the linear relations.

If one more round is added, the resulting confusion matrix would be as in Figure 5.7. The confusion would now jump to 0.75, whereas the total diffusion would be equal to 0.76. For a three round cipher, the result would be to have a surface with a mean

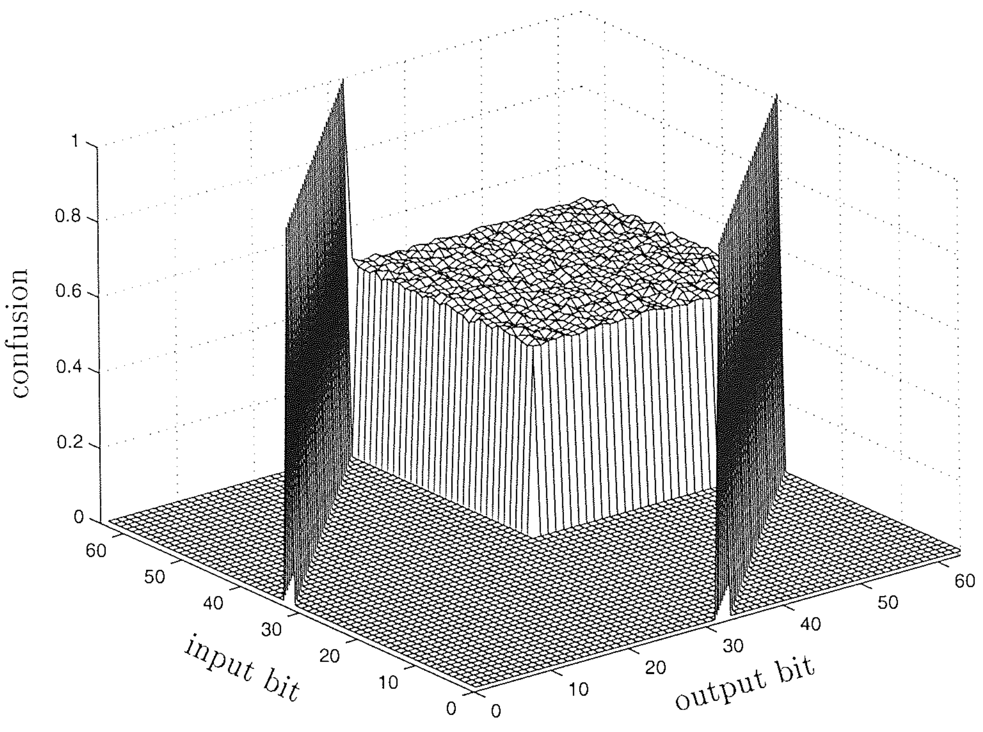


Figure 5.6: Confusion matrix of a one round Luby-Rackoff construction.

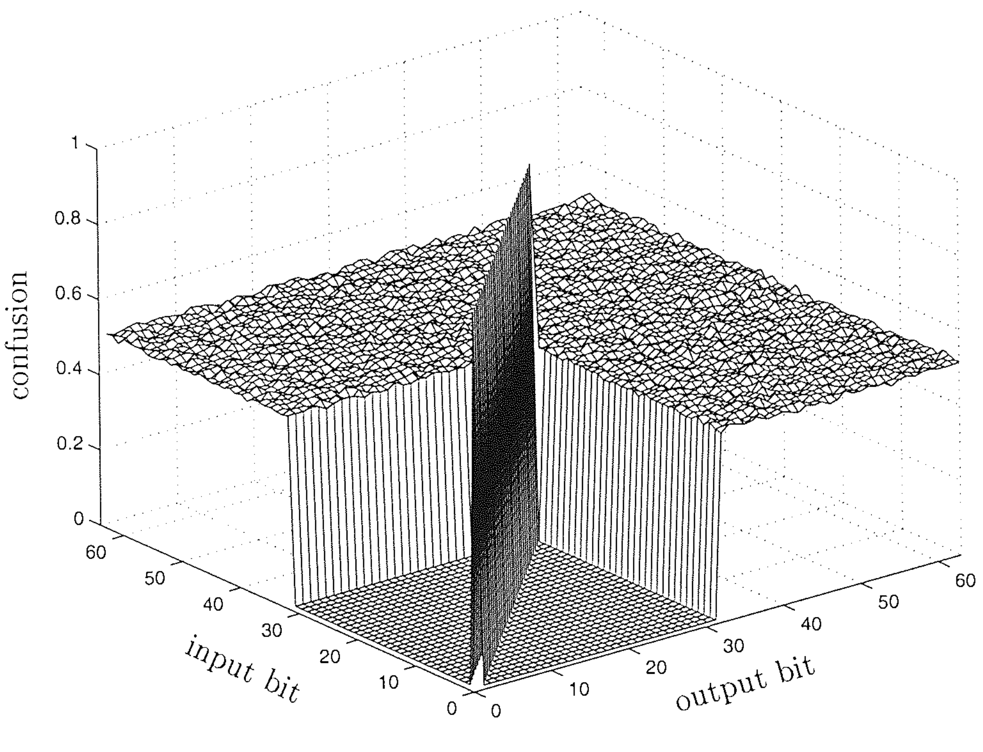


Figure 5.7: Confusion matrix of a two round Luby-Rackoff construction.

of 0.5, as in Figure 5.4, in the case of the full DES algorithm.

Luby and Rackoff concluded that the round function of the DES was not a random function. This could be seen from the graph of a one round DES (Figure 5.8). The confusion is very low, equal to 0.0415 and the total diffusion is equal to 0.0625.

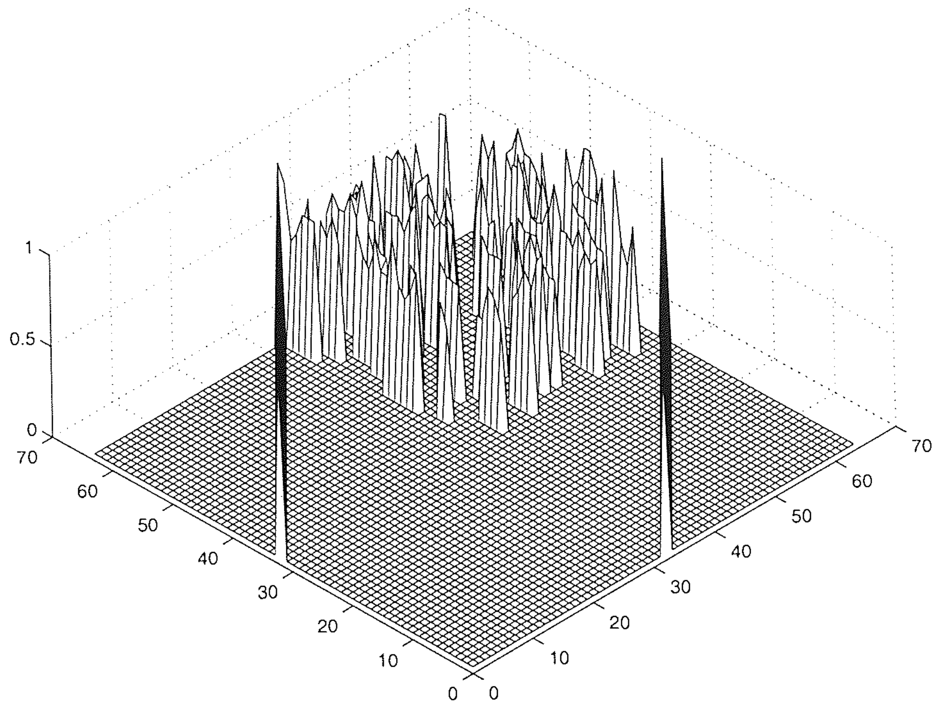


Figure 5.8: Confusion matrix of a one round DES.

In order to achieve total diffusion, five rounds of the DES step are needed, whereas for full confusion six rounds are required. The five round DES is interesting, because although the total diffusion has its maximum value (1), the confusion is 0.9996. This could be seen in Figure 5.9 where a region on the third quadrant reveals that five iterations of the DES step are inadequate and therefore this bias could be exploited.

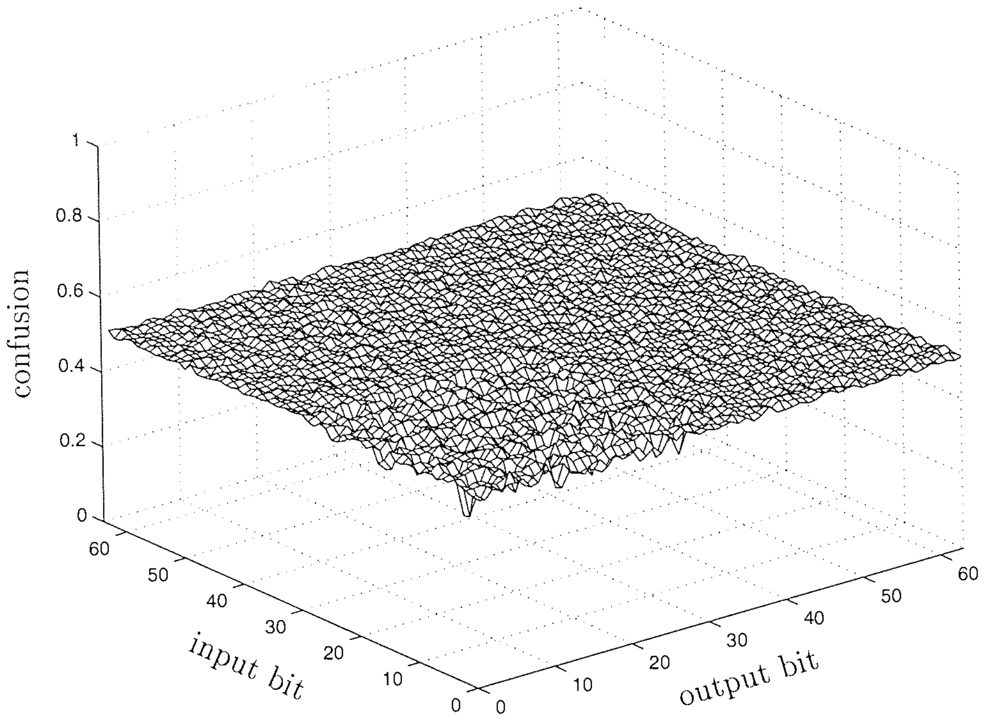


Figure 5.9: Confusion matrix of a five round DES.

5.6 Diffusion distinguisher

It is argued in this thesis that the confusion, diffusion and Ψ matrices contain more information than the confusion and diffusion levels of a cipher. The diffusion distinguisher described in this thesis demonstrates how to extract information from the Ψ matrices, which can be used for distinguishing a cipher from a pseudorandom permutation generator.

The distinguisher runs on the Ψ matrices (section 3.5.4) which are generated for the computation of the confusion and diffusion. The three properties the Ψ matrices must have for a potentially strong cipher, are:

- the number of ones should be equal to the number of zeroes,
- the ones (and zeroes) should be *randomly* distributed in the matrix,
- Ψ_i and Ψ_j should not be *similar* for $i \neq j$.

The first and partly the second requirements are dealt with a series of autocorrelation tests which are applied on the rows and columns of the matrix. The third requirement which involves comparison of the tables, is partly examined with the depth test as described in Chapter 3.

The diffusion distinguisher complements the tests for the second and third requirements. After obtaining the equations for the expected number of zeroes in a product of two matrices (Appendix A), the random distribution of the zeroes (and ones) could be assessed, given that the equations hold for random distribution of zeroes (and ones).

As described in Chapter 3, the matrix is split into its four quadrants which are compared through the distinguisher. This was done in order to overcome two problems:

- for a square matrix of a relatively large dimension n (in this case $n = 64$), the expected number of zeroes in the product saturates to zero for a low density of ones in the original matrices, approximately equal to 0.33. For $n = 32$, the expected density of zeroes is zero at approximately 0.52 density of ones.

- Since there is an inherent structure in a one-round Feistel block, where there is at least a zero sub-matrix, two identity sub-matrices and a matrix related to the round function, the distinguisher trial would not provide any further information. However, the distribution of ones and zeroes in the sub-matrix which corresponds to the round function could be assessed for randomness on the distribution of ones and zeroes. The same applies for a two-round Feistel block or more, if the block is unbalanced.

Consequently, to examine an encryption step with the distinguisher one could start by constructing one block with that step and add the same block until the distinguisher fails to distinguish structure in the output of the cipher.

5.6.1 DES

For a one-round DES, the distinguisher trial provided the following results:

```
Zeros in product:0.807861
expected zeros in prod:0.090274

                     Expected
        Q1         Q2         Q3         Q4
 -----------------------------------------
Q1 1.000000 1.000000 1.000000 1.000000
Q2 1.000000 0.970138 0.969218 0.881416
Q3 1.000000 0.969218 0.970138 0.881416
Q4 1.000000 0.881416 0.881416 0.000266

                      Actual
        Q1         Q2         Q3         Q4
 ----------------------------------------
Q1 1.000000 1.000000 1.000000 1.000000
Q2 1.000000 0.968750 0.968750 0.874023
Q3 1.000000 0.968750 0.968750 0.874023
Q4 1.000000 0.874023 0.874023 0.523438
```

The actual and expected zeroes in the product of the whole matrix differ about 70%(= (0.807861 − 0.090274) × 100%), which was expected, since there is a structure in the matrix. The figures which are of great interest are the expected and actual values (Q_4, Q_4), which relate to the DES round function. Since there is a big difference $(0.523438 - 0.000266 = 0.52)$, we conclude that the output of the round function is far from random. For convenience $\mathbf{E}(Q_i, Q_j)$ would represent the element of the ith-row-jth-column of the expected values table and$\mathbf{A}(Q_i, Q_j)$ would represent similarly the element in the actual values table.

For a two-round DES, the distinguisher trial generated the following results:

```
Zeros in product:0.242188
expected zeros in prod:0.000000

                     Expected
        Q1         Q2         Q3         Q4
 -----------------------------------------
Q1 0.970138 0.889227 0.889227 0.688485
Q2 0.889227 0.003239 0.642429 0.241046
Q3 0.889227 0.642429 0.002456 0.241046
Q4 0.688485 0.241046 0.241046 0.000000
```

```
                  Actual
       Q1         Q2         Q3         Q4
  --------------------------------------
Q1 0.968750 0.882812 0.882812 0.628906
Q2 0.882812 0.594727 0.600586 0.208984
Q3 0.882812 0.596680 0.585938 0.173828
Q4 0.628906 0.189453 0.187500 0.009766
```

Since Q_1 is the identity matrix, $\mathbf{A}(Q_1, Q_i)$ (and $\mathbf{A}(Q_i, Q_1)$) would be the densities of zeroes in Q_i, for $i = 1, \ldots, 4$. Moreover, since $Q_2 = Q_3$, it is verified that $\mathbf{A}(Q_i, Q_j) \approx \mathbf{A}(Q_k, Q_l)$, for $i, j, k, l = 2, 3$.

The actual weakness of the DES round function can be apparent on a three-round DES:

```
Zeros in product:0.004150
expected zeros in prod:0.000000

                  Expected
       Q1         Q2         Q3         Q4
  --------------------------------------
Q1 0.003994 0.241739 0.204115 0.126188
Q2 0.241739 0.000000 0.006472 0.001255
Q3 0.204115 0.006472 0.000000 0.000523
Q4 0.126188 0.001255 0.000523 0.000000

                  Actual
       Q1         Q2         Q3         Q4
  --------------------------------------
Q1 0.601562 0.246094 0.192383 0.088867
Q2 0.216797 0.033203 0.013672 0.003906
Q3 0.179688 0.023438 0.014648 0.000000
Q4 0.077148 0.003906 0.001953 0.000000
```

A balanced Feistel network completes a cycle after three rounds. This would mean that every bit involved participated in both source and target of the cipher. It would be therefore expected that the round function would offer adequate level of randomness. However, in the case of the DES in every product in which Q_1 is involved, the actual value differs from the expected. The difference is significant in (Q_1, Q_1), where $|\mathbf{E}(Q_1, Q_1) - \mathbf{A}(Q_1, Q_1)| \approx 0.60$. The differences in the remaining products are less significant, yet high enough for the distinguisher to succeed.

The problem remaining though is the lower limit of the differences where the distinguisher could successfully distinguish a cipher from a random source. If the distinguisher runs on the whole DES cipher or any other cipher which is considered to be strong, the results could be accepted as the lowest differences. However, since this test performs only on a sample of input blocks, one could accept the lower limit to be as the maximum of the values that may occur after a large number of trials.

The highest difference between expected and actual value was found to be equal to 0.000295, after 10^4 trials. For a cipher which produces repeatedly values greater than 2.95×10^4, it could be argued that the distinguisher can successfully distinguish the cipher from a random source.

5.7 Evaluation performance

Equations 5.3, 5.4, 5.5 and 5.6 participate in the `evaluation()` function, where a cipher instance is evaluated by estimating the confusion and marginal diffusion, and applying the evaluation rules. Computation of the evaluation speed is required in order to determine whether the evaluation function can participate in a communication session without causing significant overheads (Aim 3).

The speed of evaluation was measured on a SUN Ultra Sparc 10, for cipher instances ranging from 2 to 10 layers. It can be seen from Figure 5.10 that the evaluation speed follows a linear relation with the number of layers. Linear regression provided the following relation:

$t = -0.01389 + 0.079167 \times (layers) \quad \text{msec.}$

with $R^2 = 0.9962$

The time overheads are in the order of a fraction of a msec and therefore, the evaluation can be indeed performed on-line (Aim 3). It is expected as a rough guide that the delay due to the cipher evaluation stage will be between 1 and 2 msec.

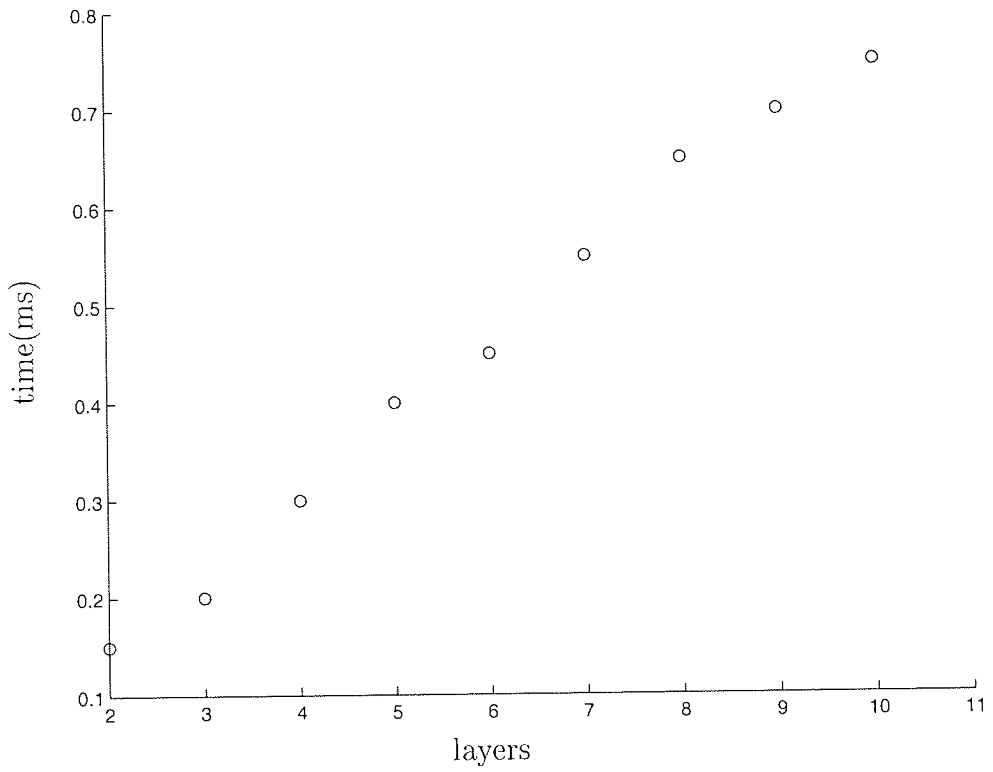


Figure 5.10: Performance evaluation().

5.8 Concluding remarks

This chapter presented the two stages of the tests, followed by additional proposals for graphic representation of the results as well as the diffusion distinguisher test.

During the first stage, the tests were performed on the encryption steps, in order to determine the values stored in the CBPs. The tests involved calculation of the diffusion and confusion, as well as the marginal diffusion. The diffusion and marginal diffusion were calculated from the diffusion matrix and the Ψ matrices respectively. The confusion was calculated from the confusion matrix.

The second stage of the results consisted of three categories of tests. All three categories involved regression analysis techniques. The purpose for applying linear regression was to develop a model for estimating the confusion for any cipher instance of the defined cipher space. The first category of tests focused on homogeneous Feistel networks. It was shown that in homogeneous Feistel networks, the confusion can be estimated only by using the confusion of the underlying step.

The first category failed to estimate the confusion of a heterogeneous Feistel network. Hence, the second category of tests examined two round heterogeneous Feistel networks. A relation with high R^2 was found, in which the confusion of the two round Feistel network was dependent on the confusions of the two underlying encryption blocks.

The third category of the tests considered more than two rounds of heterogeneous Feistel networks and after a number of attempts, more parameters were introduced in order to obtain a high $\bar{R}^2$. More specifically, different equations were calculated according to the comparison of the target and source of two consecutive encryption blocks. The different equations revealed the significance of the influence of the topology of a Feistel network. It should be also noted, that since we were dealing with more than one independent variables, the quantity $\bar{R}^2$ is observed instead of R^2, since the latter is not appropriate to equations with more than one independent variables.

The results for the heterogeneous Feistel networks apart from the CBP values of confusion, diffusion, target, source and marginal diffusion, included the marginal diffusion of the $n-1$ sub-product of the $n-$round product cipher. A regression provided a logarithmic relation between marginal diffusion and confusion with high coefficient of determination R^2, and thus the marginal diffusion can be calculated from the confusion of a cipher.

Graphical representation of the confusion and diffusion matrices help identify weaknesses of a cipher regarding the relations between input and output bits. Moreover, if the graphical representation is used on one encryption step, it could be seen how close the round function is to a pseudorandom permutation generator, since the area in which the round function applies should have confusion values of 0.5 for a random source.

Finally, the diffusion distinguisher test demonstrated that the information stored in the matrices can be used for developing further tests which may advance cryptanalysis.

Chapter 6

Conclusions

6.1 Introduction

This chapter summarises the findings of the thesis. These consist of evaluation, conclusions and observations, followed by suggestions for further research.

6.2 Aims of thesis

The aims of this thesis were:

1. to define a family of symmetric block ciphers composed from a set of publicly known encryption steps;

   (a) offer practical security against a known plaintext attack.

   (b) offer the option of trade-off between complexity over speed.

   (c) use and allow standardisation.

   (d) accommodate evolution of cryptographic technology.

2. to develop a framework for evaluating their cryptographic strength;

   (a) analyse confusion and diffusion by investigating contributing parameters and describing them with quantitative means and relate the strength of the cipher to these values.

   (b) offer rules to filter out known weak instantiations.

3. to develop a fast on-line cipher evaluation method to be embedded in the cryptographic algorithm negotiation protocol and to give an indicative measure of the cipher evaluation speeds in relation to the algorithm;

   (a) develop a method in which the results from the long term computations are summarised and utilised in the algorithm negotiation protocol, in order to speed the negotiation.

   (b) develop a message forwarding utility to test the algorithm negotiation protocol.

6.3 Evaluation

6.3.1 Literature

The literature involved investigating various sources, such as journals, indexed search on electronic libraries, as well as sources on the Internet. These provided an extensive source of relevant information. In addition, personal communication with authors in the field of cryptography and network security was useful. Cross-references from bibliographies and references in sources were also investigated.

An extensive literature survey was completed which revealed a large body of current literature in some areas and also showed that some areas covered in this thesis have not yet been much reported.

Cryptography

The literature in cryptographic composition suggests that by combining weak encryption steps the result would be a cipher much more secure than its components, due to the avalanche effect. The most common structure used in the literature to observe this effect were product ciphers which consisted of Feistel networks.

Another issue in the literature on product ciphers, was the design of a key schedule, which generates the subkeys used in every round. The distinction and definition of a strong and weak key schedule by Knudsen (1994b), combined with a method for generating an arbitrary number of sub keys developed by the same researcher, simplified the design of the proposed cryptosystem.

On the use of cryptographic modes of operation, the two main conclusions in the literature were that modes of operation should be used to hide patterns of the plaintext within the ciphertext and that every mode of operation should be extensively studied. For this reason the literature in this area was studied although the project described in this thesis has done little on ciphers with feedbacks. A full investigation on ciphers with feedbacks would require extensive work beyond the scope of this project.

The literature on cryptography provided:

- combination of weak steps to produce a stronger cipher,
- information on weak and strong key schedules,
- importance of cryptographic modes of operation in hiding patterns.

Cryptographic primitives

The literature in this area was studied because the proposed system was built on cryptographic primitives.

Feistel transformations are widely used in symmetric block ciphers and were the main components in product ciphers. The proposed project focuses on product ciphers, and it became apparent from the literature that most product ciphers were built on Feistel networks, so theoretical papers concerning Feistel Networks were reviewed and the results included in the survey. The theoretic work reported by Luby and Rackoff on the analysis of the security of the DES round function led to the idea of a distinguisher. The distinguisher was based on the concept in which the output of a cipher should be "indistinguishable" from the output of a random function generator. The conditions under which a three round Feistel transformation performs is provably secure were demonstrated by Luby and Rackoff.

Zheng *et al.*(1990) discussed practical implementation of a Luby and Rackoff construction and also showed that a distinguisher can be constructed for a three round Feistel transformation using the same function.

The work in this thesis is aimed to be a practical implementation of a Luby and Rackoff construction but took note of Coppersmith's results which showed that more than four Feistel rounds were needed. The provable security aspects were not addressed in this thesis which aimed at effective practical security. Investigation of the provable security aspects is proposed for further work.

In the proposed method heterogeneous Feistel transformations account for the largest part of the defined cipher space.

However, unbalanced Feistel and heterogeneous networks have not been reported extensively in the literature. A plausible reason is that most standard product ciphers are based on balanced homogeneous Feistel transformations and there was no motivation for such research.

The literature on cryptographic primitives provided:

- what classes of primitives are used in existing ciphers,
- the widespread of Feistel transformations in product ciphers,
- conditions on provable security in Feistel transformations, provided many lemmas and conlcusions used to guide the tests.

Cryptanalysis

Following the review of cryptanalytic attacks, it became apparent from the literature that the practical security of the proposed cryptosystem could be defined as the requirement to conceal plaintext information during a communications session. Consequently, the properties of confusion and diffusion became of primary interest in the project. The evaluation techniques which were developed in this thesis followed the tests for randomness found in the literature.

The literature on cryptanalysis provided:

- the idea of practical security,
- use of confusion and diffusion as measures of security.

Computer network security

Network security issues were not studied in depth. However, the common approaches were described in order to identify the current trends in network security, and more specifically in cipher negotiation.

The literature on computer network security provided:

- implementation considerations,
- information on use of multiple ciphers.

6.3.2 Analysis and definition of measures

The design objectives of the proposed cipher were constructed by close mapping from the aims related to the cipher to the individual design objectives.

The proposed cryptosystem was outlined and described to provide a basis for the representation of the cipher instances in the negotiation protocol. This representation and description proved satisfactory for this purpose, since the implementation of the negotiation was completed using only the content of the representation and description.

The analysis and definition of measures covered a number of issues. The calculation of the total search space (secondary aim B.) demonstrated its non-polynomial nature and proved to be a major advance over the search space of current ciphers. For example, for a cipher with eight blocks, the search space was calculated to be equal to 10^{112} approximately.

The key schedule provided a source of strong sub-keys suitable for use with a variable-length product cipher. The algorithm used was based on a strong hash function which provided a convenient balance between computational speed and cryptographic strength. Although the key schedule is not part of the evaluation of the prototype, it is recommended for a working system.

The definition of the confusion and diffusion matrices provided an effective way to measure confusion and diffusion. Furthermore, the theoretical proof on the multiplicative property of the diffusion matrix verified Luby's and Rackoff's requirement for the round function being a random permutation generator for a provably secure cipher.

The versions of confusion and diffusion described in the literature were structure dependent and therefore unsuitable for the proposed cipher, which has a variable structure. The definitions of confusion and diffusion were therefore generalised to take account of the variable structure while remaining applicable to existing block ciphers. These prove to be very effective measures, which confirmed theoretical results in the literature and also proved to be internally consistent in predicting the confusion would always be less than diffusion, which was confirmed by experiment.

There appears, to the writer of this thesis, to be a considerable amount of information in the Ψ matrices. Some methods were proposed to produce measures based on information extracted from the matrices: the autocorrelation test, depth test and diffusion distinguisher test were proposed, and while not providing a complete test of cryptographic strength, gave a first indication of whether or not a proposed cipher contained a sufficient number of rounds.

The confusion and diffusion values of an encryption step are stored in the CBP, along with other values which are needed for the estimation of confusion and diffusion, and are described below.

6.3.3 Prototype

The prototype was constructed in the C programming language making use of publicly available code for the various ciphers. The flexible pointer handling provided by C allowed development of compact functions to implement any cipher instance in the cipher space.

The prototype was able to generate all required results. The flexible construction of the prototype would allow it to be easily adapted to further development, by allowing more tests and encryption steps to be embedded.

6.4 Tests

There were two series of tests in this thesis. The first series were tests on the encryption steps, in order to update their CBP fields. The second series were on cipher instances: confusion, diffusion and marginal diffusion were measured and recorded, and the results were used in the regression analysis, which determined parameters influencing confusion and diffusion, and stored them in the CBP.

The concept of the Cryptographic Block Profile, CBP, was introduced in order to summarise parameters of the encryption steps which were derived from long-run tests. These parameters are used in the evaluation function which was included in the algorithm negotiation protocol. Consequently the purpose of the CBP was to make on-line evaluation the cipher feasible in terms of soft time constraints (Aim 3.(a)). It has been shown in this thesis that the confusion is always less than the diffusion of a cipher. So to test the strength of a cipher it is sufficient to test the confusion. The cipher can be considered secure when the confusion reaches its maximum. More accurate information could be obtained if the diffusion is also calculated, but the CBP should be kept small, so the use of the diffusion matrix is optional.

The use of regression analysis on the experimental results provided evidence that indeed the confusion increases with cryptographic composition and the parameters which influence this increment were identified. The results from the regression form the estimation equations for the confusion which were embedded in the on-line evaluation process. The regression analysis provided fits to experimental data which showed a high level of correlation (in excess of 90%). Hypothesis testing further supported the quality of the fit which gives high level of confidence of the estimation equations created for the on-line evaluation. Hypothesis testing on the estimation equations confirmed their high predictive power.

The limitation of this method is that the approximation function can be applied only to Feistel-type encryption steps.

The definition of the marginal diffusion was useful in the approximation of confusion and its use as a parameter increased the coefficient of determination ($\bar{R}$) of the regression. It should be also noted that calculation of marginal diffusion is not time consuming: since the variance is small ($exp(-5)$), only a small number of trials are needed.

The graphical representation of the confusion and diffusion matrices was an alternative way to qualitatively determine how close a round function is to a pseudorandom permutation generator and identify any linear relations or biases between input and output bits.

The graphical representations proved very helpful by highlighting weaknesses which suggested areas for formal investigation.

The diffusion distinguisher relates theoretical (expected) with actual results on density of zeroes in the product of two matrices, and tests showed that it could successfully distinguish a cipher with a small number of rounds from a random permutation generator. The success of the diffusion distinguisher in distinguishing a cipher from a random permutation generator, depended on the number of rounds and was different for different ciphers. It is conjectured in this thesis that by dividing a Ψ matrix into smaller segments the distinguisher would improve its ability to distinguish. This is supported by preliminary testing in which the distinguisher was more effective when the matrix was segmented into four quadrants.

Finally, a simple security protocol was used to demonstrate the effectiveness of the on-line evaluation of the cipher instances (Aim 3.(b)). Features such as time-stamps, certificates, participation of a trusted third party, etc. were not included. In real applications they would be added to the security protocol described in this thesis.

6.5 Recommendations for future research

These recommendations are set out in the order of importance, with the most important first.

The work described in this thesis excluded differential and linear cryptanalysis. It is proposed that further work should investigate these aspects. Fields in the CBP could be added to include characteristics related to differential and linear cryptanalysis. For example, a field in the CBP could hold the value of the best differential characteristic so that in the product encryption of any of the encryption steps, the differential characteristic would be the product the respective characteristics and the resistance of the product cipher to differential cryptanalysis could be obtained. The relations of differential characteristics in product ciphers in terms of the encryption blocks have been established in the literature (Chapter 3). The introduction of resistance to these attacks would be a major issue in this work, since the cryptosystem would then be resistant to a chosen ciphertext attack, which is considered as the strongest attack.

This thesis has developed a relationship between the diffusion matrix of a product cipher and the diffusion matrices of the encryption steps. The relation between the confusion matrix of a product cipher and the confusion matrices of the underlying encryption blocks could be investigated. This may result in finding a more accurate function for the confusion to replace the estimating equations derived from linear regression.

The determination of the set of encryption steps to be used in the cryptosystem is an important issue. The set used by this thesis is not necessarily the best. The encryption steps were selected to fit the specific research purposes, which were mainly to model parameters rather than to find the best combination of encryption steps.

Another limitation of the proposed method is that the complexity could be arbitrarily high, and there may be a composition where the strength of the cipher may decrease. This is possible if the cipher space forms a group under the cryptographic composition of the steps, or if the composition of certain steps may be the decryption of another step (or composition of steps), without necessarily forming a group. Finding such instances may be a difficult problem, similar to finding collisions in one-way functions. Therefore the cryptosystem needs to be extensively analysed. However this is a difficult task and if the design objectives are updated to prohibit such relations, it is suggested that the encryption steps should be designed with systematic methods to avoid such occurrences.

Despite the possibility of the composed cipher being arbitrarily large, it is possible that not all permutation mappings from plaintext to ciphertext could be generated. The proposed cryptosystem could be modified to include encryption steps which generate the missing mappings (alternating group). It has been shown in the literature that Feistel networks can generate this group.

The diffusion and confusion matrices as well as the Ψ matrices, contain useful information concerning both the topology and the round function of the cipher. Further tests, other than the depth and the diffusion distinguisher test described in this thesis, could be investigated.

Furthermore, tests on the cryptographic primitives making up the encryption steps could also be integrated in the `testsuite` environment. Currently `testsuite` includes tests which are applied to the encryption steps which are treated as transformation blocks, rather than tests on the underlying primitive. For example, the environment could include tests for S-box resistance to differential and linear cryptanalysis.

Provable security provides information which is generally useful in the long run and guides the design of ciphers. Although the proposed cryptosystem is not provably secure, it is possible to convert it by changing the cryptographic algorithm generator to allow only instances which were provable secure. The proposed structure by Anderson and Biham (1996b) for example, specified a three round cipher by using a keyed hash function and a stream cipher. If the encryption steps consist of stream ciphers and hash functions, and a combination rule of `TYPE:hash-TYPE:stream-TYPE:hash` is used, then every instance would be a provably secure cipher.

A protocol specification based on the IPSec specifications could be developed. Since the IPSec suite is the current internetworking standard, the efficiency of the proposed method would increase once integrated in the IP infrastructure.

6.6 Conclusion

Overall the research has achieved its aims, providing a method for product cipher negotiation using on-line evaluation for private communications over computer networks. The use of a variable cipher space generates a large total space to enhance security. The need for on-line validation of proposed cipher structures arises because it is infeasible to evaluate cryptographic strength over all instances in a time acceptable to on-line negotiation.

In this work measures of confusion and diffusion are used to evaluate cryptographic strength. In order to provide fast on-line evaluation, estimation equations for confusion based on parameters of the encryption steps, are generated for experimental testing of some composed ciphers. These estimating equations allow fast evaluation of any proposed cipher.

The measures for the cryptographic strength proposed in this thesis agreed with theoretical results in the literature. This increased the reliability of the definitions and more particularly of the confusion and diffusion. However, as in all issues related to cryptographic strength, the proposed methods should be exposed to public scrutiny, to establish their real value.

The definition of the confusion and diffusion matrix were used to calculated the measures of confusion, diffusion and marginal diffusion also proposed in this thesis. The theoretic proof of the multiplicative property of the diffusion matrix and the modelling of the confusion were also performed in this thesis.

Furthermore, two new tests were described - the depth test and the diffusion distinguisher - which, although not part of the aims of this thesis, may encourage further research.

Finally, the introduction of the CBP, combined with successful application of regression analysis techniques made fast evaluation feasible.

It has been shown that it is feasible to produce fast on-line evaluation from a sound basis and to incorporate this in a working protocol.

References

Adams, C. & Tavares, S. (1990). "The Structured Design of Cryptographically Good S-Boxes", *Journal of Cryptology* **3**(1): 27–41.

Aiello, W. & Venkatesan, R. (1996). "Benes: A Non-Reversible Alternative to Feistel", *EUROCRYPT 96*, Vol. LNCS 1070 of *Advances in Cryptology*, Springer-Verlag, Espoo, Finland, May 31 - June 4, pp. 307–320.

Andelman, D. & Reeds, J. (1982). "On the Cryptanalysis, of Rotor Machines and Substitution-Permutation Networks", *IEEE Trans. Inf. Theory* **IT 28**(4): 578–584.

Anderson, R. (1995a). "The Classification of Hash Functions", *Cryptography and coding IV*, Springer-Verlag, pp. 83–93.

Anderson, R. (1995b). "Searching for the Optimum Correlation Attack", *K.U. Leuven Workshop on Cryptographic Algorithms*, Springer-Verlag.

Anderson, R. & Biham, E. (1996a). "Tiger: A Fast New Hash Function", http://www.cl.cam.ac.uk/users/rja14/.

Anderson, R. & Biham, E. (1996b). "Two Practical and Provably Secure Block Ciphers: Bear and LION", *Fast Software Encryption*, Vol. LNCS 1039, Springer-Verlag, pp. 113–120.

Aronsson, H. (1995). "Zero Knowledge Protocols and Small Systems", http://www.niksula.cs.hut.fi/haa/study/zeroknowledge.html.

Atkinson, R. (1995a). "Security Architecture for the Internet Protocol", RFC1825.

Atkinson, R. (1995b). "IP Authentication Header", RFC1826.

Atkinson, R. (1995c). "IP Encapsulating Security Payload (ESP)", RFC1827.

Baritaud, T., Gilbert, H., Girault, M. (1993). "FFT Hashing is not Collision-Free", *EUROCRYPT 92*, Vol. LNCS 658 of *Advances in Cryptology*, Springer-Verlag, Balatonfüred, Hungary,May 24-28.

Bauer, F. (1997). *Decrypted Secrets: Methods and Maxims of Cryptology*, Springer-Verlag.

Beker, H. & Piper, F. (1982). *Cipher Systems: The Protection of Communications*, Northwood.

Bell, D. & LaPadula, E. (1974). "Secure Computer Systems: Mathematical Foundations and Model", MITRE corp.

Bellovin, S. (1996). "Problem Areas for the IP Security Protocols", *Proc, of the 6th USENIX Security Symposium*, USENIX assoc., pp. 205–214.

Benaloh, J. (1987). "Secret Sharing Homomorphisms: Keeping Shares of a Secret", *CRYPTO 86*, Vol. LNCS 263 of *Advances in Cryptology*, Springer-Verlag, pp. 213–222.

Berlekamp, E. (1984). *Algebraic Coding Theory*, Aegean Park Press.

Beth, T. & Piper, F. (1984). "The Stop-and-Go Generator", *EUROCRYPT 84*, Vol. LNCS 209, Springer-Verlag, Paris, France, April 9-11, pp. 88–92.

Biham, E. (1993). "On Modes of Operation", *Fast Software Encryption*, Vol. LNCS 809, Springer-Verlag, Cambridge, U.K., December 9-11.

Biham, E. (1994). "Cryptanalysis of Multiple Modes of Operation", *ASIACRYPT 94*, Vol. LNCS 917, Springer-Verlag, Wollongong, Australia, November 28 - December 1.

Biham, E. (1995). "On Matsui's Linear Cryptanalysis", *EUROCRYPT 94*, Vol. LNCS 950, Springer-Verlag, Paris, France, April 9-11, pp. 398–412.

Biham, E. (1996). "Cryptanalysis of Triple-Modes of Operation", *Technical Report CS0885*, Technion Israel Institute of Technology.

Biham, E. & Shamir, A. (1991). "Differential Cryptanalysis of DES-like Cryptosystems", *CRYPTO 90*, Vol. LNCS 537 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 11-15, pp. 2–21.

Biham, E. & Shamir, A. (1992). "Differential Cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and LUCIFER", *CRYPTO 91*, Vol. LNCS 576 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 11-15, pp. 156–171.

Biham, E. & Shamir, A. (1993a). *Differential Cryptanalysis of the Data Encryption Standard*, Springer-Verlag.

Biham, E. & Shamir, A. (1993b). "Differential Cryptanalysis of the Full 16-round DES", *CRYPTO 92*, Vol. LNCS 740 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 16-20, pp. 487–511.

Biham, E. & Shamir, A. (1997). "Differential Fault Analysis of Secret Key Cryptosystems", *CRYPTO 97*, Vol. LNCS 1294 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 17-21, pp. 513–525.

Bird, R., Gopal, I., Herzberg, A., Janson, P., Kutten S., Molva, R., Yung, M. (1995). "The KryptoKnight Family of Light-Weight Protocols for Authentication and Key Distribution", *IEEE/ACM Transactions on Networking* **3**(1): 31–41.

Bishop, M. (1992). "Security Analyses of Network Time Services", ftp://ftp.funet.fi /pub/unix/security/docs/papers/nts-security.ps.gz.

Blakeley, G. (1979). "Safeguarding Cryptographic Keys", *Proc. of the National Computer Conference, American Federation of Information Processing Societies*, pp. 242–268.

Blum, M. & Micali S. (1984). "How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits", *SIAM J. on Computing* **13**: 850–64.

Boneh, D., Demilio, R., Lipton, R. (1997). "On the Importance of Checking Cryptographic Protocols for Faults", *EUROCRYPT 97*, Vol. LNCS 1233 of *Advances in Cryptology*, Springer-Verlag, Konstanz, Germany, May 11-15, pp. 37–51.

Brands, S. (1994). "Untraceable Off-Line Cash in Wallet with Observers", *CRYPTO*

93, Vol. LNCS 773 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 16-20, pp. 302–318.

Brassard, G. (1979). "A Note on the Complexity of Cryptography", *IEEE Trans. on Information Theory* **IT-25**(2): 232–33.

Brickell, E. & Stinson, D. (1990). "The Detection of Cheaters in Threshold Schemes", *CRYPTO 88*, Vol. LNCS 403 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 21-25, pp. 564–577.

Brickell, E., Denning, D., Kent, S., Maher, D., Tuchman, W. (1993). "The SKIPJACK Algorithm", http://www.alw.nih.gov/Security/FIRST/papers/crypto /skipjack.txt.

Brown, L., Pieprzyk, J., Seberry, J. (1990). "LOKI: A Cryptographic Primitive for Authentication and Secrecy Applications", *AUSCRYPT 90*, Advances in Cryptology, Springer-Verlag, pp. 229–236.

Campbell, K. & Wiener, M. (1993). "DES is not a Group", *CRYPTO 92*, Vol. LNCS 740 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 16-20, pp. 512–520.

Carlet, C. (1993). "Partially-Bent Functions", *CRYPTO 92*, Vol. LNCS 740 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 16-20, pp. 280–91.

Chatterjee S. & Price, B. (1977). *Regression Analysis by Example*, Wiley.

Chaum, D. & Evertse, J. (1986). "Cryptanalysis of DES with a Reduced Number of Rounds; Sequences of Linear Factors in Block Ciphers", *CRYPTO 85*, Vol. LNCS 218 of *Advances in Cryptology*, Santa Barbara, California, USA, August 18-22, pp. 192–211.

Cheng, P, Garay, J., Herzberg, A., Krawczyk, H. (1995). "Design and Implementation of Modular Key Management Protocol and IP Secure Tunnel on AIX.", *Proc. 5th USENIX UNIX Security Symposium*, Salt Lake City, Utah.

Chikazawa, T. & Inove, T. (1990). "A New Key Sharing System for Global Telecommunications", *GLOBECOM '90*, IEEE Global Telecommunications Conference, pp. 1069–72.

Cooke, C. (1995). *"Cryptographic Techniques for Personal Communication Systems Security"*, PhD thesis, Aston University.

Coppersmith, D. (1994). "The Data Encryption Standard and its Strength Against Attacks", *IBM Journal of Research and Development* **38**(3): 243–250.

Coppersmith, D. (1996). "Luby-Rackoff: Four Rounds is not Enough", IBM Research Report, RC 20674.

Cusick, T. & Wood, M. (1991). "The REDOC II Cryptosystem", *CRYPTO 90*, Vol. LNCS 537 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 11-15, pp. 545–563.

Daemen, J., Govaerts, R., Vandewalle, J. (1994). "Weak Keys for IDEA", *CRYPTO 93*, Vol. LNCS 773 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 16-20, pp. 224–230.

Dai, Z & Yang, J. (1991). "Linear Complexity of Periodically Repeated Random Sequences", *EUROCRYPT 91*, Vol. LNCS 547 of *Advances in Cryptology*, Springer-Verlag, Brighton, UK, April 8-11, pp. 168–175.

Damgård., I. & Knudsen, L. (1994). "The Breaking of the AR Hash Function", *EUROCRYPT 93*, Vol. LNCS 765 of *Advances in Cryptology*, Springer-Verlag, Lofthus, Norway, May 23-27, pp. 286–292.

Damgård., I. & Knudsen, L. (1996). "Multiple Encryption with Minimum Key", Vol. LNCS 1029, Springer-Verlag, pp. 156–164.

Davies, D. & Price, W. (1984). *Security for Computer Networks*, Wiley.

Davis, D. & Swick, R. (1990). "Network Security via Private-Key Certificates", MIT Project Athena.

den Boer, B. (1988). "Cryptanalysis of FEAL", *EUROCRYPT 88*, Advances in Cryptology, Springer-Verlag, pp. 293–300.

den Boer, B. & Bosselaers, A. (1994). "Collisions for the Compression Function of

MD5", *EUROCRYPT 93*, Vol. LNCS 765 of *Advances in Cryptology*, Springer-Verlag, Lofthus, Norway, May 23-27, pp. 293–304.

Desmedt, Y. (1991). "The 'A' Cipher Does Not Necessarily Strengthen Security", *Cryptologia* **15**(3): 203–6.

Desmedt, Y. & Frankel, Y. (1990). "Threshold Cryptosystems", *CRYPTO 89*, Vol. LNCS 435 of *Advances in Cryptography*, Springer-Verlag, Santa Barbara, California, USA, August 20-24, pp. 307–315.

Desmedt, Y. & Frankel, Y. (1992). "Shared Generation of Authentication and Signatures", *CRYPTO 91*, Vol. LNCS 576 of *Advances in Cryptography*, Springer-Verlag, Santa Barbara, California, USA, August 11-15, pp. 457–469.

Devargas, M. (1993). *"Network Security*, NCC Blackwell.

Diffie, W. & Hellman, M. (1976). "New Directions in Cryptography", *IEEE Trans. Inf. theory* **IT-22**(6): 644–654.

Diffie, W., Van Oorschot, Wiener, M. (1992). "Authentication and Authenticated Key Exchanges", *Designs, Codes and Cryptography* **2**: 107–125.

DoD Computer Security Center (1985). "DoD Trusted Computer System Evaluation Criteria".

Dusse, S., Hoffman, P., Ramsdell, B., Lundblade, L., Repka, L. (1998). "S/MIME Version 2 Message Specification", RFC2311.

Even, S. & Goldreich, O. (1985). "On the Power of Cascaded Ciphers", *ACM Trans. on Computer Systems* **3**(2): 108–116.

Feige, U., Fiat, A., Shamir, A. (1987). "Zero-Knowledge Proofs of Identity", *Proc. of the 19th ACM Symposium on Theory of Computing*, pp. 210–217.

Feistel, H. (1974). "Block Cipher Cryptographic System", U.S. Patent #3,798,359.

Feistel, H., Notz, W., Smith, J. (1975). "Some Cryptographic Techniques for Machine-to-Machine Data Communications", *Proc. of the IEEE* **63**(11): 1545–1554.

Freed, N. & Borenstein, N. (1996). "MIME Part 1: Format of Internet Message Bodies", RFC2045.

Freirer, A., Karlton P., Kocher, P. (1996). "The SSL Protocol Version 3.0", Internet Draft.

Garey., M. & Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman.

Garon, G. & Outerbridge, R. (1991). "DES Watch: An Examination of the Sufficiency of the Data Encryption Standard for Financial Institution Information Security in the 1990's", *Cryptologia* **15**(3): 177–93.

Geffe, P. (1973). "How to Protect Data with Ciphers that are Really Hard to Break", *Electronics* **46**(1): 99–101.

Gilbert, H. & Chase, G. (1991). "A Statistical Attack on the FEAL-8 Cryptosystem", *CRYPTO 90*, Vol. LNCS 537 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 11-15, pp. 22–33.

Goldreich, O., Goldwasser, S., Micali, S. (1984). "How to Construct Random Functions", *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*.

Goldwasser, S.,Micali, S., Rivest, S. (1988). "A Digital Signature Scheme Secure Against Adaptive Chosen Message Attacks", *SIAM J. in Computing* **17**: 281–308.

Golomb, S. (1967). *Shift register sequences*, Holden-Day.

Good, J. (1957). "On the Serial Test for Random Sequences", *Ann. Math. Statistics* **28**: 262–264.

Greene, W. (1993). *Econometric Analysis*, Macmillan.

Gujarati, D. (1988). *Basic Econometrics*, McGraw-Hill.

Günther, C. (1988). "Alternating Step Generators Controlled by De Bruijn Sequences", *EUROCRYPT 87*, Vol. LNCS 304 of *Advances in Cryptology*, Springer-Verlag, pp. 5–14.

Günther, C. (1990). "An Identity-Based Key-Exchange Protocol", *EUROCRYPT 89*, Vol. LNCS 434 of *Advances in Cryptology*, pp. 29–37.

Heys, H. & Tavares, S. (1996). "Substitution-Permutation Networks Resistant to Differential and Linear Cryptanalysis", *Journal of Cryptology* **9**(1): 1–19.

Ingemasson, I. & Simmons, G. (1991). "A Protocol to Set Up Shared Secret Schemes without the Assistance of a Mutually Trusted Party", *EUROCRYPT 90*, Vol. LNCS 473 of *Advances in Cryptology*, Springer-Verlag, Aarhus, Denmark, May 21-24, pp. 266–282.

Ioannidis, J. & Blaze, M. (1994). "The Architecture and Implementation of Network-Layer Security Under Unix", http://hightop.nrl.navy.mil /docs/ps_files/swipe.ps.

ISO N179 (1992). "AR Fingerprint Function", draft, ISO-IEC/JTC1/SC27/WG2.

ISO/IEC 9796 (1991). "Information Technology- Security Techniques- Digital Signature Scheme Giving Message Recovery".

Jansen, C. & Boekee, D. (1988). "Modes of Blockcipher Algorithms and their Protection Against Active Eavesdropping", *EUROCRYPT 87*, Vol. LNCS 304 of *Advances in Cryptology*, Springer-Verlag, pp. 281–286.

Janson, P. & Tsudik, G. (1993). "Secure and Minimal Protocols for Authenticated Key Distribution", http://www.ccs.neu.edu/home/thigpen/docs /Security_Papers/misc/protocols/braided.ps.

Jennings, S. (1980). *"A Special Class of Binary Sequences"*, PhD thesis, University of London.

Johnson, D., Dolan, G., Kelly, M., Le, A., Matyas, S. (1991). "Common Cryptographic Architecture Programming Interface", *IBM Systems Journal* **30**(2): 130–49.

Kahn, D. (1976). *The Codebrakers*, McMillan.

Kaliski, B. (1998). "personal communication".

Katsavanos, P. & Varadharajan, V. (1993). "Security Protocol for Frame Relay", *Computer Communication Review* **23**(5): 17–35.

Kaufman, C. (1993). "DASS: Distributed Authentication Security Service", RFC1507.

Kille, S. (1991). *Implementing X.400 and X.500: The PP and QUIPU Systems*, Artech House.

Knudsen, L. (1994a). *"Block Ciphers - Analysis, Designs, Applications"*, PhD thesis, Aarhus University.

Knudsen, L. (1994b). "Practically Secure Feistel Ciphers", *Fast Software Encryption 93*, Vol. LNCS 809, Springer-Verlag, Cambridge, U.K., December 9-11, pp. 211–221.

Knuth, D. (1981). *The Art of Computer Programming*, Vol. 2, Addison-Wesley.

Koblitz, N. (1987). *A Course in Number Theory and Cryptography*, Springer-Verlag.

Koyama, K. & Terada, R. (1993). "How to Strengthen DES-like Cryptosystems Against Differential Cryptanalysis", *IEICE Trans. Fundamentals* **E76-A**(1): 63–69.

Krawczyk, H. (1996). "SKEME: A Versatile Secure Key Exchange Mechanism for Internet", *IEEE, Proceedings of the 1996 Symposium on Network and Distributed Systems Security*.

Kumar, B. & Crowcroft, J. (1993). "Integrating Security in Inter-Domain Routing Protocols", *Computer Communication Review* **23**(5): 36–52.

Lai, X. (1992). *On the Design and Security of Block Ciphers*, Vol. 1 of *ETH Series in Information Processing*, Konstanz: Hartung-Gorre Verlag.

Lai, X., & Massey, J. (1991). "A proposal for a New Block Encryption Standard", *EUROCRYPT 90*, Vol. LNCS 473 of *Advances in Cryptology*, Springer-Verlag, Aarhus, Denmark, May 21-24, pp. 389–404.

Lai, X., Massey, J., Murphy S. (1991). "Markov Ciphers and Differential Cryptanalysis", *EUROCRYPT 91*, Vol. LNCS 547 of *Advances in Cryptology*, Springer-Verlag, Brighton, UK, April 8-11, pp. 17–38.

Luby, M. & Rackoff, C. (1986). "Pseudo-random Permutation Generators and Cryptographic Composition", *Proc. 18th Annual Symposium on Theory of Computing*, pp. 356–63.

Luby, M. & Rackoff, C. (1988). "How to Construct Pseudorandom Permutations from Pseudorandom Functions", *SIAM J. Computing* **17**(2): 373–86.

Martin, K. (1993). "Untrustworthy Participants in Perfect Secret Sharing Schemes", *in* M. Ganley (ed.), *Cryptography and Coding III*, Clarendon Press, pp. 255–264.

Matsui, M. (1994). "Linear Cryptanalysis Method for DES Cipher", *EUROCRYPT 93*, Vol. LNCS 765 of *Advances in Cryptology*, Springer-Verlag, Lofthus, Norway, May 23-27, pp. 386–397.

Maugham, D., Patrick, B., Schertler M. (1995). "Internet Security Association & Key Management Protocol (ISAKMP)", Internet-daft @nic.nordu.net, draft-nsa-isakmp-00.ps.

Maurer, U. (1993). "A Simplified and Generalized Treatment of Luby-Rackoff Pseudo-random Permutation Generators", *EUROCRYPT 92*, Vol. LNCS 658 of *Advances in Cryptology*, Springer-Verlag, Balatonfüred, Hungary,May 24-28, pp. 239–255.

Meier, W. (1994). "On the Security of the IDEA Block Cipher", *EUROCRYPT 93*, Vol. LNCS 765 of *Advances in Cryptology*, Springer-Verlag, Lofthus, Norway, May 23-27, pp. 371–385.

Menezes, A. (1993). *Elliptic Curve Public Key Cryptosystems*, Kluwer.

Merkle, R. (1979). *"Secrecy, Authentication, and Public Key Systems"*, PhD thesis, Stanford University.

Merkle, R. (1990). "A Fast Software One-Way Hash Function", *J. of Cryptology* **3**(1): 43–58.

Meyer, C. & Matyas, M. (1982). *Cryptography: A New Dimension In Computer Data Security*, Willey.

Meyer, C. & Schilling, M. (1988). "Secure Program Load with Manipulation Detection Code", *SECURICOM 88*, pp. 111–130.

Mills, D. (1992). "Network Time Protocol (Version 3)- Specifications, Implementation and Analysis", RFC1305.

Mirhakkak, M. (1993). "A Distributed System Security Architecture: Applying the Transprot Layer Security Protocol", *Computer Communication Review* **23**(5): 6–16.

Mund, S., Gollman, D., Beth, T. (1988). "Some Remarks on the Cross Correlation Analysis of Pseudorandom Generators", *EUROCRYPT 87*, Vol. LNCS 304 of *Advances in Cryptology*, Springer-Verlag, pp. 25–35.

Naor, M., Ostrovsky, R., Venkatesan, R., Yung, M. (1992). "Perfect Zero-Knowledge Arguments for NP Can Be Based on General Complexity Assumptions", ftp://ftp.icsi.Berkeley.edu/pub/techreports/1992/tr-92-082.ps.Z.

National Institute of Standards & Technology (1992). "Proposed Federal Information Processing Standard for Secure Hash Standard (DSS)", *Federal Register* **57**(21): 29–40.

Needham, R. (1994). "Denial of Service: An Example", *Communications of the ACM* **37**(11): 42–46.

O'Connor (1994a). "An Analysis of a Class of Algorithms for S-box Construction", *J. of Cryptology* **7**(3): 133–152.

O'Connor (1994b). "On the Distribution of Characteristics in Bijective Mappings", *EUROCRYPT 93*, Vol. LNCS 765 of *Advances in Cryptology*, Springer-Verlag, Lofthus, Norway, May 23-27, pp. 360–70.

O'Higgins, B. & Schnider, S. (1990). "Securing Information in X.25 Networks", *GLOBECOM '90*, IEEE Global Telecommunications Conference, pp. 1073–8.

Okamoto, T. (1993). "Provably Secure and Practical Identification Schemes and Corresponding Signature Schemes", *CRYPTO 92*, Vol. LNCS 740 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 16-20, pp. "1–15 to 1–25".

Ostrovsky, R. & Wigderson, A. (1993). "One-Way Functions are Essential for Non-Trivial Zero Knowledge", *Proc. of the 2nd Israel Symposium on Theory of Computing and Systems (ISTCS93)*.

Pfleeger, C. (1989). *Security in Computing*, Prentice.

Pieprzyk, J. (1991). "How to construct Pseudorandom Permutations, from Single Pseudorandom Functions", *EUROCRYPT 90*, Vol. LNCS 473 of *Advances in Cryptology*, Springer-Verlag, Aarhus, Denmark, May 21-24, pp. 140–150.

Pieprzyk, J. (1996). "Cryptographic Algorithms: Properties, Design and Analysis", Invited lecture PRAGOCRYPT 96.

Piper, F. (1998). "Personal communication".

Postel, J. & Reynolds, J. (1985). "File Transfer Protocol", RFC959.

Rabin, M. (1979). "Digitalised Signatures and Public-Key Functions as Intractable as Factorisation", "Laboratory of Computer Science, MIT, MIT/LCS/TR-212".

RACE, Research and Development in Advanced Communication Technologies in Europe (1992). "RIPE Integrity Primitives: Final Report of RACE Integrity Primitives Evaluation", R1040.

Rivest, R. (1991). "The MD4 Message Digest Algorithm", *CRYPTO 90*, Vol. LNCS 537 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 11-15, pp. 303–311.

Rivest, R. (1992a). "The MD4 Message Digest Algorithm", RFC1320.

Rivest, R. (1992b). "The MD5 Message Digest Algorithm", RFC1321.

Rivest, R. (1994). "The RC5 Encryption Algorithm", *Fast Software Encryption*, Vol. LNCS 1008, Springer, pp. 86–96.

Rivest, R., Shamir, A., Adleman, M. (1978). "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM* **21**(2): 120–126.

RSA Laboratories (1993). "PKCS #7: Cryptographic Message Syntax Standard", Technical note 003-903022-150-000-000.

Rueppel, R. (1988). "When Shift Registers Clock Themselves", *EUROCRYPT 87*, Vol. LNCS 304 of *Advances in Cryptology*, Springer-Verlag, pp. 53–64.

Schneier, B. (1995). *E-Mail Security*, Wiley.

Schneier, B. (1996). *Applied Cryptography*, 2nd edn, Wiley.

Schneier, B. & Kelsey, J. (1996). "Unbalanced Feistel Networks and Block Cipher Design", *Fast Software Encryption*, Third International Workshop, Springer-Verlag, pp. 121–144.

Schnorr, C. (1991). "Method for Identifying Subscribers and for Generating and Verifying Electronic Signatures In A Data Exchange System", US PAT No.4,995,082.

Scott, R. (1985). "Wide Open Encryption Design Offers Flexible Implementations", *Cryptologia* **9**(1): 75–90.

Seberry, J., Zhang, X., Zheng, Y. (1995). "Pitfalls in Designing Substitution Boxes", *CRYPTO 94,*, Vol. LNCS 839 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 21-25, pp. 383–396.

Seigenthaler, T. (1986). "Cryptanalysts Representation of Non-linearly Filtered ML-Sequences", *EROCRYPT 85*, Advances in Cryptology, Springer-Verlag, pp. 103–110.

Shamir, A. (1979). "How to Share a Secret", *Communications of the ACM* **24**(11): 612–613.

Shannon, C. (1949). "Communication Theory of Secrecy Systems", *Bell System Technical Journal,* **28**: 656–715.

Shimizu, A. & Miyaguchi, S.Z (1988). "Fast Data Encipherment Algorithm, FEAL", *EUROCRYPT 87*, Vol. LNCS 304 of *Advances in Cryptology*, Springer-Verlag, pp. 267–278.

Tardy-Corfdir, A. & Gilber, H. (1992). "A Known Plaintext Attack of FEAL-4 and FEAL-6", *CRYPTO 91*, Vol. LNCS 576 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 11-15, pp. 172–182.

Theil, H. (1971). *"Principles of Econometrics"*, Wiley.

Tsudik, G. (1992). "Message Authentication with One-Way Hash Functions", *Computer Communication Review* **22**(5): 29–38.

Vaudenay, S. (1992). "FFT-Hash-II Is Not Yet Collision-Free", ftp://ftp.ens.fr/pub/reports/liens/liens-92-17.A4.ps.

Vaudenay, S. (1996). "Hidden Collisions on DSS", ftp://ftp.ens.fr/pub/reports/liens/liens-96-9.A4.ps.Z.

Wagner, D. & Schneier, B. (1996). "Analysis of the SSL 3.0 Protocol", *2nd USENIX Workshop on Electronics Commerce*, USENIX Press, pp. 29–40.

Webster, A. & Tavares, S. (1986). "On the Design of S-Boxes", *CRYPTO 85*, Vol. LNCS 403 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 21-25, pp. 523–534.

Winfield, G. & Wolman, A. (1993). "X Through the Firewall and Other Application Relays", http://www.ccs.neu.edu/home/thigpen/docs/Security_Papers /treese/93.10.ps.

Zheng, Y., Matsumoto, T., Imai, H. (1990). "On the Construction of Block Ciphers Provably Secure and Not Relying on Any Unproved Hypotheses", *CRYPTO 89*, Vol. LNCS 435 of *Advances in Cryptology*, Springer-Verlag, Santa Barbara, California, USA, August 20-24, pp. 461–480.

Zheng,Y., Pieprzyk, J, Seberry, J. (1993). "HAVAL- A One-Way Hashing Algorithm with Variable Length of Output", *AUSCRYPT 92*, Advances in Cryptology, Springer-Verlag, pp. 83–104.

Part IV

Appendices

Appendix A

Matrix multiplication

This section determines the expected density of zeros of a matrix C, which is a product of two square matrices A and B with densities of zeros p_a and p_b respectively. It is assumed that the zeros in the two matrices A and B are uniformly distributed.

Let A B and C three $n \times n$ square matrices, where $C = A \times B$. For A, the density of zeroes would be:

$$p_a = P(a_{ik} = 0) = \frac{\#(\text{zeros in } A)}{n^2}.$$

Similarly, for B:

$$p_b = P(b_{kj} = 0) = \frac{\#(\text{zeros in } B)}{n^2}.$$

For every element in C, the following relation holds:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{A.1}$$

The probability to obtain a zero is obtained from (A.1):

$$\begin{aligned}
P(c_{ij} = 0) &= \prod_{k=1}^{n} P(a_{ik} = 0 \text{ or } b_{kj} = 0) \\
&= \prod_{k=1}^{n} (P(a_{ik} = 0)P(b_{kj} = 1) + P(a_{ik} = 1)P(b_{kj} = 0) + P(a_{ik} = 0)P(b_{kj} = 0)) \\
&= \prod_{k=1}^{n} (p_a(1 - p_b) + (1 - p_a)p_b + p_a p_b) \\
&= \prod_{k=1}^{n} (p_a + p_b - p_a p_b) \\
&= (p_a + p_b - p_a p_b)^n
\end{aligned}$$

However the above equation holds for two matrices where the distribution of zeros in the first matrix is independent to the distribution of zeros in the second matrix. In

the case of multiplying the matrix with itself, the expected density of zeros would be determined as follows:
Let z_a the number of zeros in A. Following the notation above:
$p_a = \frac{z_a}{n^2}$.
Let $D = A^2$. The probability to have a zero element in D would be:

$$\begin{aligned}
P(d_{ij} = 0) &= \prod_{k=1}^{n}[P(a_{ik} = 0, a_{kj} = 0) + P(a_{ik} \neq 0, a_{kj} = 0) + P(a_{ik} = 0, a_{kj} \neq 0)] \\
&= \prod_{k=1}^{n}[P(a_{ik} = 0|a_{kj} = 0)P(a_{kj} = 0) + P(a_{ik} \neq 0|a_{kj} = 0)P(a_{kj} = 0) + \\
&\qquad P(a_{ik} = 0|a_{kj} \neq 0)P(a_{kj} \neq 0)]
\end{aligned} \tag{A.2}$$

where:

$$\begin{aligned}
P(a_{ik} = 0|a_{kj} = 0) &= \frac{z_a - 1}{n^2 - 1} \\
P(a_{ik} \neq 0|a_{kj} = 0) &= 1 - \frac{z_a - 1}{n^2 - 1} \\
P(a_{ik} = 0|a_{kj} \neq 0) &= \frac{z_a}{n^2 - 1}
\end{aligned}$$

and (A.2) would become:

$$\begin{aligned}
P(d_{ij} = 0) &= \prod_{k=1}^{n}\left[\frac{z_a - 1}{n^2 - 1} \times \frac{z_a}{n^2} + \frac{n^2 - 1 - z_a + 1}{n^2 - 1} \times \frac{z_a}{n^2} + \frac{z_a}{n^2 - 1}(1 - \frac{z_a}{n^2})\right] \\
&= \prod_{k=1}^{n}\left[\frac{(z_a - 1)z_a}{n^2(n^2 - 1)} + \frac{(n^2 - z_a)z_a}{n^2(n^2 - 1)} + \frac{(n^2 - z_a)z_a}{n^2(n^2 - 1)}\right] \\
&= \prod_{k=1}^{n}\frac{(2n^2 - z_a - 1)z_a}{n^2(n^2 - 1)} \\
&= \left[\frac{(2n^2 - z_a - 1)z_a}{n^2(n^2 - 1)}\right]^n
\end{aligned}$$

Appendix B

Summary of experimental results

B.1 Homogeneous product ciphers

Table B.1: The product of one-round DES encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion
1	0.0625	0.0104	$1.7970e^{-6}$	0.0415
2	0.3206	0.0728	$3.5784e^{-5}$	0.2906
3	0.7349	0.1760	$8.5010e^{-5}$	0.7062
4	0.9690	0.2394	$6.6771e^{-5}$	0.9568
5	1	0.2502	$4.7094e^{-5}$	0.9996
6	1	0.2498	$4.4160e^{-5}$	0.9999
7	1	0.2497	$4.2048e^{-5}$	0.9999
8	1	0.2500	$4.5161e^{-5}$	0.9999
9	1	0.2499	$4.5996e^{-5}$	0.9999
10	1	0.2508	$5.1262e^{-5}$	0.9999
11	1	0.2504	$3.7668e^{-5}$	0.9999
12	1	0.2492	$5.1280e^{-5}$	0.9999
13	1	0.2501	$5.1924e^{-5}$	0.9999
14	1	0.2497	$4.6379e^{-5}$	0.9999
15	1	0.2499	$4.9419e^{-5}$	0.9999
16	1	0.2499	$4.1520e^{-5}$	0.9999

Table B.2: The product of the two Blowfish encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion
1	0.2656	0.0623	$1.7290e^{-6}$	0.2492
2	0.7578	0.1864	$5.1442e^{-5}$	0.7454
3	1	0.2498	$8.2240e^{-5}$	0.9960
4	1	0.2506	$4.4845e^{-5}$	0.9998
5	1	0.2501	$4.9253e^{-5}$	0.9999
6	1	0.2499	$4.5010e^{-5}$	0.9999
7	1	0.2500	$4.5994e^{-5}$	0.9999
8	1	0.2492	$5.1267e^{-5}$	0.9999

Table B.3: The product of the Feistel #1 encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion
1	0.2656	0.0624	$1.0545e^{-6}$	0.2500
2	0.7578	0.1875	$3.6743e^{-5}$	0.7499
3	1	0.2505	$4.7070e^{-5}$	0.9999
4	1	0.2498	$4.8624e^{-5}$	0.9999
5	1	0.2505	$4.5241e^{-5}$	0.9999
6	1	0.2492	$4.4861e^{-5}$	0.9999
7	1	0.2501	$4.4066e^{-5}$	0.9999
8	1	0.2503	$4.9837e^{-5}$	0.9999

Table B.4: The product of the Feistel #2 encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion
1	0.2656	0.0623	$1.1523e^{-6}$	0.2500
2	0.7578	0.1872	$3.3162e^{-5}$	0.7499
3	1	0.2498	$4.3145e^{-5}$	0.9999
4	1	0.2505	$4.8620e^{-5}$	0.9999
5	1	0.2502	$4.4710e^{-5}$	0.9999
6	1	0.2501	$4.7126e^{-5}$	0.9999
7	1	0.2500	$4.4618e^{-5}$	0.9999
8	1	0.2501	$4.8760e^{-5}$	0.9999

Table B.5: The product of the target heavy 16:48 UFN encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion	comp.
1	0.2031	0.0462	$1.5329e^{-6}$	0.1858	-
2	0.4492	0.1092	$2.5695e^{-5}$	0.4355	-32
3	0.6953	0.1716	$3.4261e^{-5}$	0.6853	-32
4	0.9414	0.2347	$4.2756e^{-5}$	0.9354	-32
5	1	0.2505	$5.0069e^{-5}$	0.9994	-32
6	1	0.2499	$5.0990e^{-5}$	0.9999	-32
7	1	0.2502	$4.6150e^{-5}$	0.9999	-32
8	1	0.2498	$4.6139e^{-5}$	0.9999	-32

Table B.6: The product of the source heavy 40:24 UFN encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion	comp.
1	0.2500	0.0586	$1.1486e^{-6}$	0.2337	-
2	0.6191	0.1518	$2.8222e^{-5}$	0.6082	16
3	0.9102	0.2262	$4.2726e^{-5}$	0.9053	16
4	1	0.2500	$4.3462e^{-5}$	0.9996	16
5	1	0.2505	$4.2442e^{-5}$	0.9999	16
6	1	0.2495	$4.3668e^{-5}$	0.9999	16
7	1	0.2498	$5.2079e^{-5}$	0.9999	16
8	1	0.2497	$5.0140e^{-5}$	0.9999	16

Table B.7: The product of the LFSR #1 encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion	comp.
1	0.0293	0	0	0	-
2	0.0430	0	0	0	-64
3	0.0723	0	0	0	-64
4	0.0820	0	0	0	-64
5	0.0757	0	0	0	-64
6	0.0801	0	0	0	-64
7	0.0781	0	0	0	-64
8	0.0713	0	0	0	-64
9	0.0737	0	0	0	-64
10	0.0723	0	0	0	-64

Table B.8: The product of the LFSR #2 encryption steps

No. of rounds	total diffusion	marginal diffusion	variance	confusion	comp.
1	0.1462	0.0318	$2.3237e^{-5}$	0.1272	-
2	0.1890	0.0425	$1.2085e^{-5}$	0.1705	-64
3	0.1890	0.0423	$1.1558e^{-5}$	0.1700	-64
4	0.1899	0.0430	$1.0862e^{-5}$	0.1720	-64
5	0.1899	0.0430	$1.2610e^{-5}$	0.1720	-64
6	0.1899	0.0429	$1.2550e^{-5}$	0.1720	-64
7	0.1899	0.0433	$1.0734e^{-5}$	0.1720	-64
8	0.1904	0.0429	$1.2192e^{-5}$	0.1727	-64
9	0.1892	0.0425	$7.3371e^{-5}$	0.1696	-64
10	0.1892	0.0424	$7.9165e^{-5}$	0.1696	-64

B.2 Heterogeneous product ciphers

Tables B.9 to B.11 refer to two round product ciphers as shown in Figure B.1.

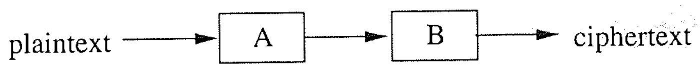


Figure B.1: A heterogeneous ($A \neq B$) two round product cipher.

Table B.9: Confusion of two round heterogeneous product cipher

A / B	DES	Feistel#1	Feistel#2	LFSR#1	LFSR#2	Blow L	Blow R	UFN 16:48	UFN 40:24
DES		.5415	.2500	.0831	.1662	.2493	.5407	.2903	.5424
Feistel#1	.5415		.2500	.3125	.2840	.2500	.7492	.4988	.7336
Feistel#2	.2500	.2500		.2500	.4682	.7492	.2500	.3120	.2967
LFSR#1	.0415	.2500	.4999		.2499	.4988	.2492	.2482	.2337
LFSR#2	.5228	.6611	.3124	.2484		.3120	.6608	.4111	.7852
Blow L	.2466	.2500	.7461	.2463	.4643		.2496	.3116	.2951
Blow R	.5377	.7461	.2500	.3086	.2805	.2496		.4949	.7330
UFN 16:48	.5750	.6859	.3120	.2797	.1870	.3116	.6856		.8104
UFN 40:24	.4159	.6243	.2941	.2337	.2966	.2939	.6236	.4797	

Table B.12 summarises the confusion resulting from two round heterogeneous product ciphers with various values of confusion.

Table B.10: Marginal diffusion of two round heterogeneous product cipher

A B	DES	Feistel#1	Feistel#2	LFSR#1	LFSR#2	Blow L	Blow R	UFN 16:48	UFN 40:24
DES		.1357	.0625	.0207	.0416	.0623	.1348	.0727	.1356
Feistel#1	.1353		.0626	.0779	.0708	.0621	.1823	.1243	.1840
Feistel#2	.0626	.0628		.0626	.1173	.1872	.0624	.0782	.0741
LFSR#1	.0105	.0625	.1241		.0627	.1245	.0627	.0623	.0584
LFSR#2	.1305	.1658	.0781	.0619		.0779	.1661	.1024	.1966
Blow L	.0615	.0628	.1863	.0624	.1155		.0619	.0779	.0742
Blow R	.1347	.1868	.0623	.0771	.0702	.0626		.1234	.1836
UFN 16:48	.1442	.1710	.0781	.0699	.0469	.0780	.1717		.2026
UFN 40:24	.1039	.1559	.0737	.0587	.0739	.0733	.1558	.1200	

Table B.11: Target/source comparison between the encryption steps

A B	DES	Feistel#1	Feistel#2	LFSR#1	LFSR#2	Blow L	Blow R	UFN 16:48	UFN 40:24
DES	0	0	-64	-24	-48	-64	0	-16	8
Feistel#1	0	0	-64	-24	-48	-64	0	-16	8
Feistel#2	64	64	0	40	-16	0	64	48	56
LFSR#1	40	40	-24	-64	-40	-24	40	-40	32
LFSR#2	-16	-16	-48	-40	-64	-48	-16	-32	-8
Blow L	64	64	0	40	-16	0	-64	48	56
Blow R	0	0	-64	-24	48	64	0	-16	8
UFN 16:48	-16	-16	-48	-40	-64	-48	-10	-32	8
UFN 40:24	8	8	-56	-32	-40	-56	8	8	16

Table B.12: Confusion of two round heterogeneous balanced Feistel Networks

A	B	result	A	B	result
0.0370	0.0415	0.1575	0.2500	0.1012	0.6010
0.0370	0.2500	0.5369	0.1272	0.1012	0.4449
0.0370	0.2492	0.5318	0.0415	0.2500	0.5415
0.0415	0.0370	0.2649	0.2500	0.0415	0.5415
0.2500	0.0370	0.5369	0.2500	0.2492	0.7461
0.2492	0.0370	0.5362	0.2492	0.2500	0.7492
0.1249	0.0415	0.3778	0.0415	0.0415	0.2906
0.1249	0.2500	0.6250	0.0367	0.0415	0.1573
0.1249	0.2492	0.6212	0.0367	0.2500	0.5366
0.0415	0.1249	0.4116	0.0367	0.1012	0.3194
0.2500	0.1249	0.6249	0.0415	0.0367	0.2637
0.2492	0.1249	0.6242	0.2500	0.0367	0.5366
0.1012	0.0415	0.3439	0.1012	0.0367	0.3202
0.1012	0.2500	0.6012	0.0415	0.2492	0.5377
0.1012	0.1272	0.4809	0.2492	0.0415	0.5407
0.0415	0.1012	0.3778			

B.3 Regression analysis

Table B.13: Ordinary Least Squares estimation of the DES.

				Probability	
Function	α	β	R^2	α	β
$y_t = \alpha y_{t-1}^{\beta}$	1.01	0.37675	0.98588	0.290	0.000
$y_t = \alpha e^{\beta y_{t-1}}$	2.62	0.98756	0.82209	0.000	0.000
$y_t = \alpha + \beta \log y_{t-1}$	1.0022	0.22471	0.99705	0.000	0.000
$y_t = \dfrac{y_{t-1}}{\alpha y_{t-1} - \beta}$	0.90405	-0.10593	0.99581	0.000	0.000
$y_t = \dfrac{e^{\alpha+\beta y_{t-1}}}{1 + e^{\alpha+\beta y_{t-1}}}$	-2.2721	11.2593	0.94863	0.005	0.000

Table B.14: Ordinary Least Squares estimation of Feistel #1.

				Probability	
Function	α	β	R^2	α	β
$y_t = \alpha y_{t-1}^{\beta}$	1.00	0.20293	0.95742	0.465	0.000
$y_t = \alpha e^{\beta y_{t-1}}$	0.71	0.35509	0.89465	0.000	0.000
$y_t = \alpha + \beta \log y_{t-1}$	1.0040	0.17633	0.95742	0.000	0.000
$y_t = \dfrac{y_{t-1}}{\alpha y_{t-1} - \beta}$	0.88576	-0.11100	0.9865	0.000	0.000
$y_t = \dfrac{e^{\alpha+\beta y_{t-1}}}{1 + e^{\alpha+\beta y_{t-1}}}$	-0.62799	10.0118	0.89465	0.572	0.000

Table B.15: Ordinary Least Squares estimation of Feistel #2.

				Probability	
Function	α	β	R^2	α	β
$y_t = \alpha y_{t-1}^{\beta}$	1.00	0.20293	0.95742	0.465	0.000
$y_t = \alpha e^{\beta y_{t-1}}$	0.71	0.35509	0.89465	0.000	0.000
$y_t = \alpha + \beta \log y_{t-1}$	1.0040	0.17633	0.95742	0.000	0.000
$y_t = \dfrac{y_{t-1}}{\alpha y_{t-1} - \beta}$	0.88576	-0.11100	0.9865	0.000	0.000
$y_t = \dfrac{e^{\alpha+\beta y_{t-1}}}{1 + e^{\alpha+\beta y_{t-1}}}$	-0.62799	10.0118	0.89465	0.572	0.000

Table B.16: Ordinary Least Squares estimation of Blow L,R.

				Probability	
Function	α	β	R^2	α	β
$y_t = \alpha y_{t-1}^{\beta}$	1.00	0.20568	0.96150	0.458	0.000
$y_t = \alpha e^{\beta y_{t-1}}$	0.70	0.35949	0.90098	0.000	0.000
$y_t = \alpha + \beta \log y_{t-1}$	1.0027	0.17826	0.96227	0.000	0.000
$y_t = \dfrac{y_{t-1}}{\alpha y_{t-1} - \beta}$	0.88499	-0.11286	0.98964	0.000	0.000
$y_t = \dfrac{e^{\alpha+\beta y_{t-1}}}{1 + e^{\alpha+\beta y_{t-1}}}$	-1.9548	11.0766	0.98428	0.000	0.000

Table B.17: Ordinary Least Squares estimation of UFN 16:48.

				Probability	
Function	α	β	R^2	α	β
$y_t = \alpha y_{t-1}^{\beta}$	1.01	0.48524	0.98092	0.296	0.000
$y_t = \alpha e^{\beta y_{t-1}}$	0.43	0.87987	0.90150	0.000	0.000
$y_t = \alpha + \beta \log y_{t-1}$	1.0060	0.34157	0.98442	0.000	0.000
$y_t = \dfrac{y_{t-1}}{\alpha y_{t-1} - \beta}$	0.69684	-0.29989	0.99354	0.000	0.000
$y_t = \dfrac{e^{\alpha+\beta y_{t-1}}}{1 + e^{\alpha+\beta y_{t-1}}}$	-4.0988	13.0041	0.94241	0.002	0.000

Table B.18: Ordinary Least Squares estimation of UFN 40:24.

				Probability	
Function	α	β	R^2	α	β
$y_t = \alpha y_{t-1}^{\beta}$	1.00768	0.33245	0.97796	0.348	0.000
$y_t = \alpha e^{\beta y_{t-1}}$	0.56653	0.57997	0.90847	0.000	0.000
$y_t = \alpha + \beta \log y_{t-1}$	1.0050	0.26448	0.98718	0.000	0.000
$y_t = \dfrac{y_{t-1}}{\alpha y_{t-1} - \beta}$	0.79991	-0.19635	0.99805	0.000	0.000
$y_t = \dfrac{e^{\alpha+\beta y_{t-1}}}{1 + e^{\alpha+\beta y_{t-1}}}$	-3.4807	12.5682	0.95299	0.003	0.000

B.4 Diffusion matrices

1. `permute #1.` $\mathcal{D} = \begin{bmatrix} \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 \\ \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 \end{bmatrix}$

2. `vigenere.` $\mathcal{D} = \begin{bmatrix} \mathbf{A}_8(\mathbf{0.62}) & \mathbf{0}_8 & \cdots & \mathbf{0}_8 \\ \mathbf{0}_8 & \mathbf{A}_8(\mathbf{0.62}) & & \\ \vdots & & \ddots & \\ \mathbf{0}_8 & & & \mathbf{A}_8(\mathbf{0.62}) \end{bmatrix}$

where $\mathbf{A}_8(\mathbf{0.62}) = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 1&1&1&0&0&0&0&0 \\ 1&1&1&1&0&0&0&0 \\ 1&1&1&1&1&0&0&0 \\ 1&1&1&1&1&1&0&0 \\ 1&1&1&1&1&1&1&0 \\ 1&1&1&1&1&1&1&1 \end{bmatrix}$

3. `permute #2`. $\mathcal{D} = \begin{bmatrix} \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 \\ \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 \\ \mathbf{I}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 & \mathbf{0}_8 \end{bmatrix}$

4. `1-round DES` (Chapter 5).

5. `Feistel #1`. $\mathcal{D} = \begin{bmatrix} \mathbf{0}_{32} & \mathbf{I}_{32} \\ \mathbf{I}_{32} & \mathbf{A}_{32}(1) \end{bmatrix}$

6. `Feistel #2`. $\mathcal{D} = \begin{bmatrix} \mathbf{A}_{32}(1) & \mathbf{I}_{32} \\ \mathbf{I}_{32} & \mathbf{0}_{32} \end{bmatrix}$

7. `LFSR #1`. $\mathcal{D} = \begin{bmatrix} \mathbf{I}_{24} & & \mathbf{0}_{24\times 40} \\ \mathbf{A}_{8\times 24}(0.69) & \mathbf{I}_8 & \mathbf{B}_{8\times 32}(0.31) \\ \mathbf{0}_{32} & \mathbf{I}_{32} & \end{bmatrix}$

$$\text{where } \mathbf{A}_{8\times 24}(0.69) = \begin{bmatrix} 1&1&0&1&1&1&1&1&0&1&1&1&0&1&1&0&1&1&0&1&0&1&1&1 \\ 1&1&1&0&1&0&1&0&1&1&1&0&1&0&1&1&1&0&1&0&1&0&0&1 \\ 1&1&1&1&0&1&0&1&1&1&1&1&0&1&0&1&1&1&0&1&0&1&0&0 \\ 1&1&1&1&1&0&1&0&0&1&1&1&1&0&1&0&0&1&1&0&1&0&1&0 \\ 1&1&1&1&1&0&0&0&0&1&1&0&1&1&0&1&0&1&1&1&0&1&1&1 \\ 1&1&1&1&1&1&0&0&1&0&1&1&0&1&1&0&1&0&1&1&1&0&1&1 \\ 0&1&1&1&1&1&1&0&1&1&0&1&1&0&1&1&0&1&0&1&1&1&0&1 \\ 1&0&1&1&1&1&1&1&1&1&1&0&1&1&0&1&1&0&1&0&1&1&1&0 \end{bmatrix},$$

$$\mathbf{B}_{8\times 32}(0.31) = \begin{bmatrix} 1&1&0&0&0&1&0&0&1&0&0&0&1&1&0&0&0&0&0&0&1&0&1&1&0&0&0&0&0&0&0&1 \\ 0&1&1&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&0&0&0&0&1&0&1&0&0&0&0&0&0&0 \\ 0&0&1&1&0&0&0&1&0&0&1&0&0&0&0&1&0&1&1&0&0&0&0&1&0&1&0&0&0&0&0&0 \\ 1&0&0&1&1&0&0&0&1&0&0&1&0&0&0&0&0&0&1&1&0&0&0&0&1&0&1&0&0&0&0&0 \\ 0&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&0&1&0&1&1&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&0&1&0&1&1&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&0&1&0&1&1&0&0&0&0&0&0&1&0&0 \\ 1&0&0&0&1&0&0&1&0&0&0&1&1&0&0&1&0&0&0&0&0&1&0&1&1&0&0&0&0&0&1&0 \end{bmatrix}$$

8. `LFSR #2`. $\mathcal{D} = \begin{bmatrix} \mathbf{I}_{48} & & \mathbf{0}_{48\times 16} \\ \mathbf{A}_{16}(0.68) & \mathbf{B}_{16\times 40}(1) & \mathbf{I}_{16} \end{bmatrix}$

$$\text{where } \mathbf{A}_{16}(0.68) = \begin{bmatrix}
1&1&1&1&1&1&0&0&1&1&1&1&1&1&1&1\\
1&1&1&1&1&0&1&0&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
0&0&0&0&0&0&0&0&1&1&1&1&1&0&0&0\\
0&0&0&0&0&0&0&0&1&1&1&1&1&0&0&0\\
0&0&0&0&0&0&0&0&1&1&1&1&1&1&0&0\\
0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&0\\
0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1\\
1&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1\\
1&1&0&0&0&0&0&0&1&1&1&1&1&1&1&1\\
1&1&1&0&0&0&0&0&1&1&1&1&1&1&1&1\\
1&1&1&1&0&0&0&0&1&1&1&1&1&1&1&1\\
1&1&1&1&1&0&0&0&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&0&0&1&1&1&1&1&1&1&1\\
1&1&1&1&1&0&1&0&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1
\end{bmatrix}$$

9. `Blow R` and `Blow L` have the same diffusion matrices as `Feistel #1` and `Feistel #2` respectively.

10. `UFN 16:48`. $\mathcal{D} = \begin{bmatrix} \mathbf{0}_{48\times16} & \mathbf{I}_{48} \\ \mathbf{I}_{16} & \mathbf{A}_{16\times48}(\mathbf{1}) \end{bmatrix}$

11. `UFN 40:24`. $\mathcal{D} = \begin{bmatrix} \mathbf{0}_{24\times40} & \mathbf{I}_{24} \\ \mathbf{I}_{40} & \mathbf{A}_{40\times24}(\mathbf{1}) \end{bmatrix}$

B.5 Cryptographic Block Profiles

Table B.19: CBPs of the encryption steps

Name:	permute #1	vigenere	permute#2	1-round DES	Feistel #1
Type:	permutation	substitution	permutation	feistel	feistel
Source(msb):	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
Source(lsb):	0x00000000	0x00000000	0x00000000	0xffffffff	0xffffffff
Target(msb):	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
Target(lsb):	0x00000000	0x00000000	0x00000000	0xffffffff	0xffffffff
Tot.diffusion:	0.0156	0.0588	0.0156	0.0625	0.2656
Marg.diffusion:	0.0	0.0054	0.0	0.0104	0.1876
Confusion:	0.0	0.0219	0.0	0.0415	0.2499

Name:	Feistel #2	LFSR #1	LFSR #2	Blowfish L	Blowfish R
Type:	feistel	other	other	feistel	feistel
Source(msb):	0xffffffff	0x000000ff	0x00000000	0xffffffff	0x00000000
Source(lsb):	0x00000000	0x00000000	0x0000ffff	0x00000000	0xffffffff
Target(msb):	0xffffffff	0xffffff00	0xffffffff	0xffffffff	0x00000000
Target(lsb):	0x00000000	0xffffffff	0xffff0000	0x00000000	0xffffffff
Tot.diffusion:	0.2656	0.0293	0.0457	0.2656	0.2656
Marg.diffusion:	0.0613	0.0	0.0061	0.0552	0.0645
Confusion:	0.2499	0.0	0.1621	0.2492	0.2492

Name:	Des.IP	Des.IP^{-1}	2-round DES	UFN 16:48	UFN 40:24
Type:	permutation	permutation	feistel	feistel	feistel
Source(msb):	0x00000000	0x00000000	0x00000000	0x00000000	0x000000ff
Source(lsb):	0x00000000	0x00000000	0xffffffff	0x0000ffff	0xffffffff
Target(msb):	0x00000000	0x00000000	0x00000000	0x0000ffff	0x00000000
Target(lsb):	0x00000000	0x00000000	0xffffffff	0xffffffff	0x00ffffff
Tot.diffusion:	0.0156	0.0156	0.3206	0.0498	0.2500
Marg.diffusion:	0.0	0.0	0.0671	0.0498	0.0530
Confusion:	0.0	0.0	0.2905	0.1755	0.2336

B.6 Data used in the regression analysis of heterogeneous product ciphers

Table B.20: Data produced from *compare_st*$(A, B) = 0$.

c_1	c_2	m_1	m_2	d_1	d_2	c_3	m_{f_1}
0.2906	0.25	0.0728	0.0624	0.3206	0.2656	0.749	0.0104
0.749	0.2492	0.185	0.0623	0.7659	0.2656	0.7489	0.0104
0.5377	0.0415	0.1347	0.0104	1.0	0.0625	0.995	0.0623
0.995	0.2492	0.2465	0.0623	1.0	0.2656	0.9987	0.0104
0.995	0.25	0.2465	0.0624	1.0	0.2656	0.9987	0.0104
0.995	0.0415	0.2465	0.0104	1.0	0.0625	0.9986	0.0104
0.5415	0.0415	0.1353	0.0104	0.5547	0.0625	0.9999	0.0624
0.9999	0.2492	0.2514	0.0623	1.0	0.2656	0.9999	0.0624
0.9999	0.25	0.2514	0.0624	1.0	0.2656	0.9999	0.0624
0.7627	0.25	0.1915	0.0624	0.7734	0.2656	0.8935	0.0104
0.7627	0.0415	0.1915	0.0104	0.7734	0.0625	0.8718	0.0104
0.8689	0.0415	0.2128	0.0104	0.875	0.0625	0.9999	0.0624
0.8689	0.25	0.2128	0.0624	0.875	0.2656	0.9999	0.0624
0.5415	0.0415	0.1357	0.0104	0.5547	0.0625	0.749	0.0104
0.7643	0.0415	0.1871	0.0104	0.7734	0.0625	0.8745	0.0104
0.7643	0.25	0.1871	0.0624	0.7734	0.2656	0.8955	0.0104
0.2903	0.25	0.0727	0.0624	0.3086	0.2656	0.6664	0.0104
0.2903	0.0415	0.0727	0.0104	0.3086	0.0625	0.5197	0.0104

Table B.21: Data produced from $|compare\_st(A, B)| = abs(8)$.

c_1	c_2	m_1	m_2	d_1	d_2	c_3	m_{f_1}
0.4159	0.0415	0.1039	0.0104	0.4316	0.0625	0.8534	0.0586
0.4159	0.25	0.1039	0.0624	0.4316	0.2656	0.8845	0.0586
0.8534	0.2337	0.2125	0.0586	0.8613	0.25	0.9686	0.0104
0.9686	0.0415	0.2406	0.0104	0.9727	0.0625	0.9924	0.0586
0.9686	0.25	0.2406	0.0624	0.9727	0.2656	0.9924	0.0586
0.6243	0.0415	0.1559	0.0104	0.6348	0.0625	0.9103	0.0586
0.6243	0.25	0.1559	0.0624	0.6348	0.2656	0.9368	0.0586
0.9103	0.2337	0.2271	0.0586	0.915	0.25	0.9731	0.0104
0.9731	0.0415	0.2458	0.0104	0.9756	0.0625	0.9933	0.0586
0.9731	0.25	0.2458	0.0624	0.9756	0.2656	0.9932	0.0586
0.2906	0.2337	0.0728	0.0586	0.3206	0.25	0.6341	0.0104
0.7062	0.2337	0.176	0.0586	0.7349	0.25	0.8941	0.0104
0.5415	0.2337	0.1353	0.0586	0.5547	0.25	0.885	0.0624
0.9999	0.2337	0.2512	0.0586	1.0000	0.25	0.9999	0.0104
0.5415	0.2337	0.1357	0.0586	0.5547	0.25	0.7289	0.0104
0.749	0.2337	0.1864	0.0586	0.7659	0.25	0.9051	0.0104
0.5424	0.2337	0.1356	0.0586	0.5576	0.25	0.7616	0.0104
0.7616	0.0415	0.1912	0.0104	0.7705	0.0625	0.949	0.0586
0.7616	0.25	0.1912	0.0624	0.7705	0.2656	0.9521	0.0586
0.2903	0.2337	0.0727	0.0586	0.3086	0.25	0.5721	0.0104
0.5721	0.0415	0.1449	0.0104	0.584	0.0625	0.9009	0.0586
0.5721	0.25	0.1449	0.0624	0.584	0.2656	0.916	0.0586
0.7336	0.2337	0.184	0.0586	0.7422	0.25	0.9527	0.0624
0.9009	0.2337	0.2265	0.0586	0.9075	0.25	0.9847	0.0586
0.916	0.2337	0.2314	0.0586	0.9199	0.25	0.9998	0.0586

Table B.22: Data produced from $|compare\_st(A, B)| = abs(16)$.

c_1	c_2	m_1	m_2	d_1	d_2	c_3	m_{f_1}
0.2337	0.2337	0.0586	0.0586	0.025	0.25	0.6082	0.0586
0.6082	0.2337	0.1518	0.0586	0.6191	0.25	0.9053	0.0586
0.9053	0.2337	0.2262	0.0586	0.9102	0.25	0.9996	0.0586
0.9996	0.2337	0.25	0.0586	1.0	0.25	0.9999	0.0586
0.0415	0.1858	0.0104	0.0462	0.0625	0.2031	0.575	0.0104
0.25	0.1858	0.0624	0.0462	0.2656	0.2031	0.6859	0.0624
0.1858	0.0415	0.0462	0.0104	0.2031	0.0625	0.2903	0.0462
0.1858	0.025	0.0462	0.0624	0.2031	0.2656	0.4988	0.0462
0.1858	0.2492	0.0462	0.0623	0.2031	0.2656	0.4949	0.0462
0.2492	0.1858	0.0623	0.0462	0.2656	0.2031	0.6856	0.0623
0.2906	0.1858	0.0728	0.0462	0.3206	0.2031	0.8674	0.0104
0.8674	0.25	0.229	0.0624	0.8831	0.2656	0.9952	0.0462
0.575	0.0415	0.1442	0.0104	0.5859	0.0625	0.7627	0.0462
0.7627	0.1858	0.1915	0.0462	0.7734	0.2031	0.936	0.0104
0.936	0.0415	0.2306	0.0104	0.9463	0.0625	0.9936	0.0462
0.936	0.25	0.23	0.0624	0.9463	0.2656	0.9952	0.0462
0.575	0.25	0.1442	0.0624	0.5859	0.2656	0.8626	0.0462
0.862	0.1858	0.2175	0.0462	0.875	0.2031	0.9998	0.0624
0.9998	0.0415	0.2431	0.0104	1.0	0.0625	0.9998	0.0462
0.9998	0.25	0.2431	0.0624	1.0	0.2656	0.9998	0.0462
0.2903	0.1858	0.0727	0.0462	0.3047	0.2031	0.8106	0.0104
0.8106	0.0415	0.1981	0.0104	0.832	0.0625	0.9848	0.0462
0.8106	0.25	0.1981	0.0624	0.832	0.2656	0.9875	0.0462
0.6859	0.25	0.171	0.0624	0.6875	0.2656	0.8689	0.0462
0.8689	0.1858	0.2128	0.0462	0.875	0.2031	0.9998	0.0624
0.9998	0.0415	0.25	0.0104	1.0	0.0625	0.9998	0.0462
0.6859	0.0415	0.171	0.0104	0.6875	0.0625	0.7643	0.0462
0.7643	0.1858	0.1871	0.0462	0.7734	0.2031	0.9375	0.0104
0.4988	0.1858	0.1243	0.0462	0.5078	0.2031	0.9254	0.0624

Appendix C

Description of the ABSENT testsuite environment

C.1 The list of available commands

Once the `testsuite` is run, the user enters the ABSENT command mode, where the prompt would be `ABSENT>`. By typing ?, a list of all available commands are displayed:

```
Welcome to ABSOLUTE ENCRYPTION test suite

Type ? for help.

ABSENT>?
seed         - create an algorithm by giving a seed
random       - create automatically a random algorithm
define       - construct an algorithm manually
edit         - edit the current algorithm
list         - list of cryptographic primitives
display      - display the current algorithm
graph        - draw the current algorithm
show         - demonstrate the transformation of a cryptographic primitive
key[bin]     - display or change the key [in binary format]
encrypt      - encrypt a string
run          - run encryption/decryption sequences
speed        - perform a time trial on the current algorithm
test         - measure confusion/diffusion
ciphertext   - encrypt a file
plaintext    - decrypt a file
save         - save the current algorithm
```

```
load         - load an algorithm
script       - execute an ABSENT script (? script for help).
quit         - quit ABSOLUTE ENCRYPTION

ABSENT>|
```

The commands could be divided into three categories, one for the construction of a symmetric block cipher instance, one for testing, and a category for batch processing, which is handled by a script interpreter.

C.2 Cipher development

list The `list` command display all available encryption steps usually named after the cryptographic primitive and the feedback blocks:

```
ABSENT>list

Cryptographic primitives:
-------------------------
1.permute #1     2.vigenere
3.permute #2     4.1-round DES
5.Feistel #1     6.Feistel #2
7.LFSR #1        8.LFSR #2
9.Blowfish L     10.Blowfish R
11.Des.IP                12.Des.IP^-1
13.2-round DES           14.UFN 16:48
15.UFN 40:24

Feedback blocks:
----------------
1.rotate left            2.rotate right
3.hash #1                4.times mod 2^32
5.plus  mod 2^32         6.permute #1
7.permute #2             8.inv. permute #1
9.vigenere               10.inv. vigenere
11.inv. permute2         12.1-round DES
13.inv.1-round DES       14.Feistel #1
15.inv. Feistel #1       16.Feistel #2
17.inv. Feistel #2       18.LFSR #1
19.LFSR #2               20.Blowfish L
21.Blowfish R
```

show The `show` command demonstrates a particular encryption step, using a preselected key. It should be noted that when `testsuite` is initiated, the key by default has the value *abcdefgh*:

```
ABSENT>show 1
Demonstrating cryptographic primitive permute #1
Input:abcdefgh
                        Key:abcdefgh
Output:cahdgebf
```

define The `define` command is used to construct the cipher by selecting the encryption steps as the blocks of the product cipher, and to combine any optional feedback blocks. The number of layers should be entered after the define command and for every layer a sequence of numbers separated by spaces should be entered. The first number denotes the encryption step and the remaining the feedback blocks, if any. The system would finally respond with the constructed cipher using the identification numbers of the blocks:

```
ABSENT>define

Number of layers (max 11):3
    :
Layer 1:1
    :
Layer 2:6 3
    :
Layer 3:2
Layers selected:3
1. 1:
2. 6:3-
3. 2:
```

display At any time the user could check which cipher has been constructed with the `display` command:

```
ABSENT>display
Layers selected:3
1. 1:
2. 6:3-
3. 2:
```

graph For a better visual presentation of the algorithm, the `graph` command draws the current cipher in text mode. It should be noted that the terminal should have enough columns for a correct representation of a large cipher:

```
ABSENT>graph

                                 /---------------\
                                 |               |
                              ___|hash #1        |<-
                             |   |               |  |
                             V   \_______________/  |
       /---------------\     |   /---------------\  ^
       |               |     V   |               |  |
->---->| permute #1    |--->(+)->| Feistel #2    |--->...-> ciphertext
       |               |         |               |
       \_______________/         \_______________/
```

random To select a random instance from the space of all possible ciphers, the `random` command could be used. The command uses the system clock to initialise a random number generator. It should be noted that the same cipher could not be selected in the future, since the seed is not apparent to the user.

seed This command behaves like the `random` command, with the difference that the initialisation value of the random number generator is selected by the user and not by the system clock. If the user records the seed of a given cipher, the latter could be reconstructed at any time.

edit The `edit` command could be used for modifying the current algorithm. The system responds by asking the layer which should be edited. If the number of layer is selected, the new layer then should be entered with the same way as in the definition of the cipher. In addition a layer could be removed with the minus sign (−) followed by the number of the layer, or a new layer could be inserted, with the plus sign (+), followed by the number of layer.

save This command saves the current cipher. The format of the file is the header "`ABSENT cryptographic algorithm structure`", followed by the indexes of the involved blocks.

load This command loads a cipher for testing. It should be noted that if there is any current cipher, it would be lost.

C.3 Cipher testing

encrypt The `encrypt` command requires a string and uses the current cipher for encrypting it.

run When the `run` command is typed, the "`ABSENT>`" prompt changes to "`Input:`". Every string which is typed is encrypted and decrypted. The string is also displayed in binary format. To return to the "`ABSENT>`" prompt, the input string should be "`exit`".

ciphertext This command is for encrypting a file. The program prompts for input filename and destination filename. The latter would be the encrypted version of the former, using the current cipher and key.

plaintext This command is the reverse of the previous. In order to restore the original file, the key as well as the defined cipher should be identical to the one used when running the `ciphertext` command.

speed The `speed` command performs a speed trial test of the current cipher, by encrypting and decrypting 10,000 characters. If the command is run more than once, the change of speed from the previous test is also produced.

test This command is the core command of the test suite. by typing `test`, the following options would appear:

```
ABSENT>test
Test to run:
            1. Frequency
            2. Serial
            3. Confusion-diffusion
            4. Autocorrelation
            5. Diffusion matrix
            6. Confusion matrix/depth analysis
            7. Total diffusion/marginal diffusion
            8. Diffusion distinguisher
Select test:|
```

The first two tests consider every output bit as a sequence generator, independent of the other output bits. The input could be one of the following:

```
Input:
      1. Linear (whole input)
      2. Linear per byte
      3. Random
      4. Linear byte - constant others
      5. Structured input
Select input:|
```

Selection (1) considers an input which is increased linearly. Selection (2) considers an input byte to be changed linearly, while the rest of the inputs are random. In selection (3) all input bit values are randomly selected, in a uniformal way. Selection (4) is as (2) with the difference that the remaining bytes are not random but remain constant. Finally, selection (5) provides preselected values which are mainly used for the Autocorrelation test. The default values were the following:

```
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
0xff,0x00,0x00,0x00,0x00,0x00,0x00,0x00
```

```
0x00,0xff,0x00,0x00,0x00,0x00,0x00,0x00
0x00,0x00,0xff,0x00,0x00,0x00,0x00,0x00
0x00,0x00,0x00,0xff,0x00,0x00,0x00,0x00
0x00,0x00,0x00,0x00,0xff,0x00,0x00,0x00
0x00,0x00,0x00,0x00,0x00,0xff,0x00,0x00
0x00,0x00,0x00,0x00,0x00,0x00,0xff,0x00
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff
0x00,0xff,0xff,0xff,0xff,0xff,0xff,0xff
0xff,0x00,0xff,0xff,0xff,0xff,0xff,0xff
0xff,0xff,0x00,0xff,0xff,0xff,0xff,0xff
0xff,0xff,0xff,0x00,0xff,0xff,0xff,0xff
0xff,0xff,0xff,0xff,0x00,0xff,0xff,0xff
0xff,0xff,0xff,0xff,0xff,0x00,0xff,0xff
0xff,0xff,0xff,0xff,0xff,0xff,0x00,0xff
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0x00
0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00
0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff
0x00,0xff,0x00,0xff,0x00,0xff,0x00,0xff
0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa
```

On the above data there is an obvious structure. The autocorrelation test uses these data to examine whether this structure is destroyed by the application of the cipher.

For the confusion-diffusion tests, the parameter which could be the key or the plaintext is selected and the change in every output bit is examined, given the change in every input bit (confusion), as well as the weight of the change vector (diffusion), as described in 2.13.4. The data could also be written in an output file.

If the fast generation of the diffusion matrix is required, option (5) could be selected. For most of the ciphers, the number of loops could be equal to 40; even if more loops are needed for the determination of the actual diffusion matrix, the generated matrix with 40 loops could be considered since the probability to gain more information for an additional test would be less than $(1/2)^{40}$. The first matrix generated for the computation of the diffusion matrix runs through the autocorrelation test, were the distribution of ones and zeroes of the rows and columns is assessed. The results are produced after the diffusion matrix.

The data generated for the depth analysis, could be also used to estimate the entries of the confusion matrix. The depth analysis investigates whether likely relationships exist between input and output bit pairs (section 3.5.7).

A relatively fast test is the estimation of total and marginal diffusion. The maximum number of encryptions is set to 100, since the probability of having an actual different state after those rounds is very low. In practice, for a strong cipher around 12 encryptions should be expected. The marginal diffusion which was stored in the cryptographic block profile was the second element of the marginal diffusion values.

The diffusion distinguisher test calculates the actual and expected densities of zeroes in the diffusion matrix as well as the respective quadrants it consists of. The expected densities are calculated on the products of the combination of the diffusion matrix and the submatrices.

key The key could be presented or modified with the `key` command. By typing `key`, the current value of the key would be produced, and the system would prompt for a new one. If nothing is typed, the key would maintain its old value. Every character beyond the eight one would be ignored, whereas if a shorter string than eight characters is typed, the remaining key values would be filled with the space character.

keybin This command is similar to `key`, but the key is produced and should be entered in a binary format.

C.4 ABSENT script

The script interpreter accepts the following commands:

outfile Specify an output file for the results. The format is:

```
outfile=<filename>
```

matlab Specify an output file to write only the results as raw data, without any header information. The format is:

```
matlab=<filename>
```

Depending on the test, `.X` is appended to the filename, where `X` is `mt`, `cd` or `depth`, for the marginal and total diffusion, block and depth test respectively.

repeat..next The `repeat..next` is equivalent to a `FOR` loop, where every command within the keywords `repeat` and `next` is repeated for n times. The format is:

`repeat=`n

command 1

command 2

⋮

command k

`next`

It should be noted that nested loops are not allowed.

random Create randomly an algorithm.

seed Create randomly an algorithm using a seed s. The format is:

```
seed=<s>
```

load Load a previously saved cipher. The format is:

```
load <filename>
```

edit This command is edits the product cipher by modifying, adding or removing a layer, as in the interactive mode. The difference is that the information is not inputted interactively, but has to be in the following format:

`edit <l> <new>`

where `<l>` is the layer to be edited, and `<new>` are the new values.

speed Performs a time trial of the current cipher.

loops Specify the number of loops for some tests. The format is:

`loops=n`

key Change the value of the key. The format is:

`key=<key_value>`

where `<key_value>` is a string.

keybin As above but `<key_value>` should be in a binary format.

input Specify the desired input parameters. The format is:

`input=<par>`

where `<par>` is one of the following:

`lin` for a linear input,

`lin <b>` for a linear byte `b=0,1,\ldots,7`,

`linct <b>` for a linear byte `b=0,1,\ldots,7` and other bytes constant,

`rand` for a random input,

`struct` for a structured predefined input,

`plain` to select the plaintext as the parameter and

`key` to select the key as the parameter.

freq Run the frequency test.

serail Run the serial test.

autocor Run the autocorrelation test.

diffusion Calculate diffusion matrix.

block Run the block cipher test (confusion/diffusion).

margdiff Calculate marginal and total diffusion.

depth Calculate depth matrix.

newtest Close all output files.

Appendix D

Listings

D.1 Makefile

```
SRC=cagscr2.c cagdes.c ecb_enc.c set_key.c scr.c rmd128.c rsa.c PrimeTools.c LargeOp.c server.c client.c cbp.c
OBJS=cagscr2.o cagdes.o ecb_enc.o set_key.o scr.o rmd128.o rsa.o PrimeTools.o LargeOp.o server.o client.o cbp.o
CC=gcc
CFLAGS=-g -fpic -I/usr/openwin/include
LIBS=-lcurses -lm -lnsl -lsocket

TESTSRC=testsuite.c cagdes.c ecb_enc.c set_key.c rmd128.c
TESTOBJS=testsuite.o cagdes.o ecb_enc.o set_key.o rmd128.o
TESTLIBS=-lm

main: $(OBJS)
$(CC) -o absent $(CFLAGS) $(OBJS) $(LIBS)
/bin/rm -f *.o

tests:  $(TESTOBJS)
$(CC) -o testsuite $(CFLAGS) $(TESTOBS) $(TESTLIBS)
/bin/rm -f testsuite.o

cagscr2.c: fstuff.h
rmd128.c: rmd128.h

rsa.c: LargeOp.h PrimeTools.h rsa.h
LargeOp.c: LargeOp.h
PrimeTools: LargeOp.h PrimeTools.h

clean:
/bin/rm -f client.o server.o
```

D.2 fstuff.h

```
/* This header contains information about the number of functions used to built the cryptographic algorithm */

#define NUM_FUNCT  21      /*  Number of normal functions */
#define NUM_CFUNCT 15     /*  Number of cryptographic functions */
#define MIN_HEIGHT 1      /*   Minimum height of layer */
#define MAX_HEIGHT 7      /*   Maximum height of layer */
#define MIN_LAYERS 5      /*   Minimum number of layers */
#define MAX_LAYERS 11      /*   Maximum number of layers */

#define MSG 8             /*    Message length         */

#define END_OF_CIPHER 0xFF  /* terminator for the exchange protocol */
```

D.3 cagdes.c

```
/*
   cagdes.c, Vasilios Katos, 1998.
   the cryptographic algorithm generator */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include "fstuff.h"
#include "rmd128.h"
#include <math.h>
```

```
#include <sys/time.h>
#include "des_locl.h"
#include "spr.h"
/* convert a byte to unsigned long*/
#define BYTE_TO_UL(stpt)      \
        (((unsigned long) *((stpt)+3) << 24) | \
 ((unsigned long) *((stpt)+2) << 16) | \
 ((unsigned long) *((stpt)+1) <<  8) | \
 ((unsigned long) *(stpt)))
/* convert a byte to unsigned int*/
#define BYTE_TO_UI(stpt) \
        (((unsigned int) *((stpt)+1) << 8) | \
         ((unsigned int) *(stpt)))

/* cyclically rotate x by n bits to the right. ROL(x,n) is defined in rmd128.h */
#define ROR(x,n)  (((x) >> (n)) | ((x) << (32-(n))))
#define ROL16(x,n)  (((x) << (n)) | ((x) >> (16-(n))))
#define ROR16(x,n)  (((x) >> (n)) | ((x) << (16-(n))))
#define RMOsize 128

extern void encrypt();
extern void decrypt();
extern void cag();
extern void copyarr();
extern void initcag();
extern void generate_round_keys();

/* Declaration of pointers to feedback transformation functions */
static unsigned char *crl(unsigned char *);
static unsigned char *crr(unsigned char *);
static unsigned char *hash1(unsigned char *);
static unsigned char *modmul(unsigned char *);
static unsigned char *modadd(unsigned char *);

/* Declaration of pointers to encryption and decryption steps */
static unsigned char *permute1(unsigned char *);
static unsigned char *ipermute1(unsigned char *);
static unsigned char *permute2(unsigned char *);
static unsigned char *ipermute2(unsigned char *);
static unsigned char *vigenere(unsigned char *);
static unsigned char *ivigenere(unsigned char *);
static unsigned char *des1(unsigned char *);
static unsigned char *ddes1(unsigned char *);
static unsigned char *elr2(unsigned char *);
static unsigned char *dlr2(unsigned char *);
static unsigned char *elr3(unsigned char *);
static unsigned char *dlr3(unsigned char *);
static unsigned char *elr4(unsigned char *);
static unsigned char *dlr4(unsigned char *);
static unsigned char *lfsr1(unsigned char *);
static unsigned char *lfsr2(unsigned char *);

/* Arrays of pointers to the functions above */
unsigned char *(*nfunct[NUM_FUNCT+1])();
unsigned char *(*nefunct[NUM_CFUNCT+1])();
unsigned char *(*ndfunct[NUM_CFUNCT+1])();

char *namef[NUM_FUNCT+1];
char *namec[NUM_CFUNCT+1];

extern char key[];                           /*  Master key */
static char key_schedule[MAX_LAYERS+1][8]; /* key schedule */
static des_key_schedule ks;                  /* key schedule for a DES round */
static unsigned char *hashkey;

int iround;                    /* round of algorithm, is global, so blocks know their position */

static const unsigned int primitive_poly[]={       /* primitive polynomials for */
     31,3,0,0,     /* use in LFSRs */
     31,6,0,0,
     31,7,0,0,
     31,13,0,0,
     32,7,6,2};

static const int prim_pol_num = 5;

static const unsigned long blow0[] = {             /* blow0-3: Blowfish S-boxes */
0xd1310ba6, 0x98dfb5ac, 0x2ffd72db, 0xd01adfb7, 0xb8e1afed, 0xba267e96,
0xba7c9045, 0xf12c7f99, 0x24a19947, 0xb3916cf7, 0x0801f2e2, 0x858efc16,
0x636920d8, 0x71574e69, 0xa458fea3, 0xf4933d7e, 0x0d95748f, 0x728eb658,
0x718bcd58, 0x82154aee, 0x7b54a41d, 0xc25a59b5, 0x9c30d539, 0x2af26013,
0xc5d1b023, 0x286085f0, 0xca417918, 0xb8db38ef, 0x8e79dcb0, 0x603a180e,
0x6c9e0e8b, 0xb01e8a3e, 0xd71577c1, 0xbd314b27, 0x78af2fda, 0x55605c60,
0xe65525f3, 0xaa55ab94, 0x57489862, 0x63e81440, 0x55ca396a, 0x2aab10b6,
0xb4cc5c34, 0x1141e8ce, 0xa15486af, 0x7c72e993, 0xb3ee1411, 0x636fbc2a,
0x2ba9c55d, 0x741831f6, 0xce5c3e16, 0x9b87931e, 0xafd6ba33, 0x6c24cf5c,
0x7a325381, 0x28958677, 0x3b8f4898, 0x6b4bb9af, 0xc4bfe81b, 0x66282193,
0x61d809cc, 0xfb21a991, 0x487cac60, 0x5dec8032, 0xef845d5d, 0xe98575b1,
0xdc262302, 0xeb651b88, 0x23893e81, 0xd396acc5, 0x0f6d6ff3, 0x83f44239,
0x2e0b4482, 0xa4842004, 0x69c8f04a, 0x9e1f9b5e, 0x21c66842, 0xf6e96c9a,
0x670c9c61, 0xabd388f0, 0x6a51a0d2, 0xd8542f68, 0x960fa728, 0xab5133a3,
0x6eef0b6c, 0x137a3be4, 0xba3bf050, 0x7efb2a98, 0xa1f1651d, 0x39af0176,
0x66ca593e, 0x82430e88, 0x8cee8619, 0x456f9fb4, 0x7d84a5c3, 0x3b8b5ebe,
0xe06f75d8, 0x85c12073, 0x401a449f, 0x56c16aa6, 0x4ed3aa62, 0x363f7706,
0x1bfedf72, 0x429b023d, 0x37d0d724, 0xd00a1248, 0xdb0fead3, 0x49f1c09b,
0X075372C9, 0x80991b7b, 0x25d479d8, 0xf6e8def7, 0xe3fe501a, 0xb6794c3b,
0x976ce0bd, 0x04c006ba, 0xc1a94fb6, 0x409f60c4, 0x5e5c9ec2, 0x196a2463,
0x68fb6faf, 0x3e6c53b5, 0x1339b2eb, 0x3b52ec6f, 0x6dfc511f, 0x9b30952c,
0xcc814544, 0xaf5ebd09, 0xbee3d004, 0xde334afd, 0x660f2807, 0x192e4bb3,
0xc0cba857, 0x45c8740f, 0xd20b5f39, 0xb9d3fbdb, 0x5579c0bd, 0x1a60320a,
0xd6a100c6, 0x402c7279, 0x679f25fe, 0xfb1fa3cc, 0x8ea5e9f8, 0xdb3222f8,
0x3c7516df, 0xfd616b15, 0x2f501ec8, 0xad0552ab, 0x323db5fa, 0xfd238760,
0x53317b48, 0x3e00df82, 0x9e5c57bb, 0xca6f8ca0, 0x1a87562e, 0xdf1769db,
0xd542a8f6, 0x287effc3, 0xac6732c6, 0x8c4f5573, 0x695b27b0, 0xbbca58c8,
0xe1ffa35d, 0xb8f011a0, 0x10fa3d98, 0xfd2183b8, 0x4afcb56c, 0x2dd1d35b,
0x9a53e479, 0xb6f84565, 0xd28e49bc, 0x4bfb9790, 0xe1ddf2da, 0xa4cb7e33,
0x62fb1341, 0xcee4c6e8, 0xef20cada, 0x36774c01, 0xd07e9efe, 0x2bf11fb4,
0x95dbda4d, 0xae909198, 0xeaad8e71, 0x6b93d5a0, 0xd08ed1d0, 0xafc725e0,
0x8e3c5b2f, 0x8e7594b7, 0x8ff6e2fb, 0xf2122b64, 0x8888b812, 0x900df01c,
0x4fad5ea0, 0x688fc31c, 0xd1cff191, 0xb3a8c1ad, 0x2f2f2218, 0xbe0e1777,
0xea752dfe, 0x8b021fa1, 0xe5a0cc0f, 0xb56f74e8, 0x18acf3d6, 0xce89e299,
0xb4a84fe0, 0xfd13e0b7, 0x7cc43b81, 0xd2ada8d9, 0x165fa266, 0x80957705,
0x93cc7314, 0x211a1477, 0xe6ad2065, 0x77b5fa86, 0xc75442f5, 0xfb9d35cf,
0xebcdaf0c, 0x7b3e89a0, 0xd6411bd3, 0xae1e7e49, 0x00250e2d, 0x2071b35e,
0x226800bb, 0x57b8e0af, 0x2464369b, 0xf009b91e, 0x5563911d, 0x59dfa6aa,
0x78c14389, 0xd95a537f, 0x207d5ba2, 0x02e5b9c5, 0x83260376, 0x6295cfa9,
0x11c81968, 0x4e734a41, 0xb3472dca, 0x7b14a94a, 0x1b510052, 0x9a532915,
0xd60f573f, 0xbc9bc6e4, 0x2b60a476, 0x81e67400, 0x08ba6fb5, 0x571be91f,
0xf296ec6b, 0x2a0dd915, 0xb6636521, 0xe7b9f9b6, 0xff34052e, 0xc5855664,
0x53b02d5d, 0xa99f8fa1, 0x08ba4799, 0x6e85076a };

static const unsigned long blow1[] = {
0x4b7a70e9, 0xb5b32944, 0xdb75092e, 0xc4192623, 0xad6ea6b0, 0x49a7df7d,
0x9cee60b8, 0x8fedb266, 0xecaa8c71, 0x699a17ff, 0x5664526c, 0xc2b19ee1,
0x193602a5, 0x75094c29, 0xa0591340, 0xe4183a3e, 0x3f54989a, 0x5b429d65,
0x6b8fe4d6, 0x99f73fd6, 0xa1d29c07, 0xefe830f5, 0x4d2d38e6, 0xf0255dc1,
0x4cdd2086, 0x8470eb26, 0x6382e9c6, 0x021ecc5e, 0x09686b3f, 0x3ebaefc9,
0x3c971814, 0x6b6a70a1, 0x687f3584, 0x52a0e286, 0xb79c5305, 0xaa500737,
0x3e07841c, 0x7fdeae5c, 0x8e7d44ec, 0x5716f2b8, 0xb03ada37, 0xf0500c0d,
0xf01c1f04, 0x0200b3ff, 0xae0cf51a, 0x3cb574b2, 0x25837a58, 0xdc0921bd,
0xd19113f9, 0x7ca92ff6, 0x94324773, 0x22f54701, 0x3ae5e581, 0x37c2dadc,
0xc8b57634, 0x9af3dda7, 0xa9446146, 0x0fd0030e, 0xecc8c73e, 0xa4751e41,
0xe238cd99, 0x3bea0e2f, 0x3280bba1, 0x183eb331, 0x4e548b38, 0x4f6db908,
0x6f420d03, 0xf60a04bf, 0x2cb81290, 0x24977c79, 0x5679b072, 0xbcaf89af,
0xde9a771f, 0xd9930810, 0xb38bae12, 0xdccf3f2e, 0x5512721f, 0x2e6b7124,
0x501adde6, 0x9f84cd87, 0x7a584718, 0x7408da17, 0xbc9f9abc, 0xe94b7d8c,
0xec7aec3a, 0xdb851dfa, 0x63094366, 0xc464c3d2, 0xef1c1847, 0x3215d908,
0xdd433b37, 0x24c2ba16, 0x12a14d43, 0x2a65c451, 0x50940002, 0x133ae4dd,
0x71dff89e, 0x10314e55, 0x81ac77d6, 0x5f11199b, 0x043556f1, 0xd7a3c76b,
0x3c11183b, 0x5924a509, 0xf28fe6ed, 0x97f1fbfa, 0x9ebabf2c, 0x1e153c6e,
0x86e34570, 0xeae96fb1, 0x860e5e0a, 0x5a3e2ab3, 0x771fe71c, 0x4e3d06fa,
0x2965dcb9, 0x99e71d0f, 0x803e89d6, 0x5266c825, 0x2e4cc978, 0x9c10b36a,
0xc6150eba, 0x94e2ea78, 0xa5fc3c53, 0x1e0a2df4, 0xf2f74ea7, 0x361d2b3d,
0x1939260f, 0x19c27960, 0x5223a708, 0xf71312b6, 0xebadfe6e, 0xeac31f66,
0xe3bc4595, 0xa67bc883, 0xb17f37d1, 0x018cff28, 0xc332ddef, 0xbe6c5aa5,
0x65582185, 0x68ab9802, 0xeecea50f, 0xdb2f953b, 0x2aef7dad, 0x5b6e2f84,
0x1521b628, 0x29076170, 0xecdd4775, 0x619f1510, 0x13cca830, 0xeb61bd96,
0x0334fe1e, 0xaa0363cf, 0xb5735c90, 0x4c70a239, 0xd59e9e0b, 0xcbaade14,
0xeecc86bc, 0x60622ca7, 0x9cab5cab, 0xb2f3846e, 0x648b1eaf, 0x19bdf0ca,
0xa02369b9, 0x655abb50, 0x40685a32, 0x3c2ab4b3, 0x319ee9d5, 0xc021b8f7,
0x9b540b19, 0x875fa099, 0x95f7997e, 0x623d7da8, 0xf837889a, 0x97e32d77,
0x11ed935f, 0x16681281, 0x0e358829, 0xc7e61fd6, 0x96dedfa1, 0x7858ba99,
0x57f584a5, 0x1b227263, 0x9b83c3ff, 0x1ac24696, 0xcdb30aeb, 0x532e3054,
0x8fd948e4, 0x6dbc3128, 0x58ebf2ef, 0x34c6ffea, 0xfe28ed61, 0xee7c3c73,
0x5d4a14d9, 0xe864b7e3, 0x42105d14, 0x203e13e0, 0x45eee2b6, 0xa3aaabea,
0xdb6c4f15, 0xfacb4fd0, 0xc742f442, 0xef6abbb5, 0x654f3b1d, 0x41cd2105,
0xd81e799e, 0x86854dc7, 0xe44b476a, 0x3d816250, 0xcf62a1f2, 0x5b8d2646,
0xfc8883a0, 0xc1c7b6a3, 0x7f1524c3, 0x69cb7492, 0x47848a0b, 0x5692b285,
0x095bbf00, 0xad19489d, 0x1462b174, 0x23820e00, 0x58428d2a, 0x0c55f5ea,
0x1dadf43e, 0x233f7061, 0x3372f092, 0x8d937e41, 0xd65fecf1, 0x6c223bdb,
0x7cde3759, 0xcbee7460, 0x4085f2a7, 0xce77326e, 0xa6078084, 0x19f8509e,
0xe8efd855, 0x61d99735, 0xa969a7aa, 0xc50c06c2, 0x5a04abfc, 0x800bcadc,
0x9e447a2e, 0xc3453484, 0xfdd56705, 0x0e1e9ec9, 0xdb73dbd3, 0x105588cd,
0x675fda79, 0xe3674340, 0xc5c43465, 0x713e38d8, 0x3d28f89e, 0xf16dff20,
0x153e21e7, 0x8fb03d4a, 0xe6e39f2b, 0xdb83adf7 };

static const unsigned long blow2[] = {
0xe93d5a68, 0x948140f7, 0xf64c261c, 0x94692934, 0x411520f7, 0x7602d4f7,
0xbcf46b2e, 0xd4a20068, 0xd4082471, 0x3320f46a, 0x43b7d4b7, 0x500061af,
0x1e39f62e, 0x97244546, 0x14214f74, 0xbf8b8840, 0x4d95fc1d, 0x96b591af,
0x70f4ddd3, 0x66a02f45, 0xbfbc09ec, 0x03bd9785, 0x7fac6dd0, 0x31cb8504,
0x96eb27b3, 0x55fd3941, 0xda2547e6, 0xabca0a9a, 0x28507825, 0x530429f4,
0x0a2c86da, 0xe9b66dfb, 0x68dc1462, 0xd7486900, 0x680ec0a4, 0x27a18dee,
0x4f3ffea2, 0xe887ad8c, 0xb58ce006, 0x7af4d6b6, 0xaace1e7c, 0xd3375fec,
0xce78a399, 0x406b2a42, 0x20fe9e35, 0xd9f385b9, 0xee39d7ab, 0x3b124e8b,
0x1dc9faf7, 0x4b6d1856, 0x26a36631, 0xeae397b2, 0x3a6efa74, 0xdd5b4332,
0x6841e7f7, 0xca7820fb, 0xfb0af54e, 0xd8feb397, 0x454056ac, 0xba489527,
0x55533a3a, 0x20838d87, 0xfe6ba9b7, 0xd096954b, 0x55a867bc, 0xa1159a58,
0xcca92963, 0x99e1db33, 0xa62a4a56, 0x3f3125f9, 0x5ef47e1c, 0x9029317c,
0xfdf8e802, 0x04272f70, 0x80bb155c, 0x05282ce3, 0x95c11548, 0xe4c66d22,
0x48c1133f, 0xc70f86dc, 0x07f9c9ee, 0x41041f0f, 0x404779a4, 0x5d886e17,
0x325f51eb, 0xd59bc0d1, 0xf2bcc18f, 0x41113564, 0x257b7834, 0x602a9c60,
0xdff8e8a3, 0x1f636c1b, 0x0e12b4c2, 0x02e1329e, 0xaf664fd1, 0xcad18115,
0x6b2395e0, 0x333e92e1, 0x3b240b62, 0xeebeb922, 0x85b2a20e, 0xe6ba0d99,
0xde720c8c, 0x2da2f728, 0xd0127845, 0x95b794fd, 0x647d0862, 0xe7ccf5f0,
0x5449a36f, 0x877d48fa, 0xc39dfd27, 0xf33e8d1e, 0x0a476341, 0x992eff74,
0x3a6f6eab, 0xf4f8fd37, 0xa812dc60, 0xa1ebddf8, 0x991be14c, 0xdb6e6b0d,
0xc67b5510, 0x6d672c37, 0x2765d43b, 0xdcd0e804, 0xf1290dc7, 0xcc00ffa3,
0xb5390f92, 0x690fed0b, 0x667b9ffb, 0xcedb7d9c, 0xa091cf0b, 0xd9155ea3,
0xbb132f88, 0x515bad24, 0x7b9479bf, 0x763bd6eb, 0x37392eb3, 0xcc115979,
0x8026e297, 0xf42e312d, 0x6842ada7, 0xc66a2b3b, 0x12754ccc, 0x782ef11c,
0x6a124237, 0xb79251e7, 0x06a1bbe6, 0x4bfb6350, 0x1a6b1018, 0x11caedfa,
0x3d25bdd8, 0xe2e1c3c9, 0x44421659, 0x0a121386, 0xd90cec6e, 0xd5abea2a,
0x64af674e, 0xda86a85f, 0xbebfe988, 0x64e4c3fe, 0x9dbc8057, 0xf0f7c086,
0x60787bf8, 0x6003604d, 0xd1fd8346, 0xf6381fb0, 0x7745ae04, 0xd736fccc,
0x83426b33, 0xf01eab71, 0xb0804187, 0x3c005e5f, 0x77a057be, 0xbde8ae24,
0x55464299, 0xbf582e61, 0x4e58f48f, 0xf2ddfda2, 0xf474ef38, 0x8789bdc2,
0x5366f9c3, 0xc8b38e74, 0xb475f255, 0x46fcd9b9, 0x7aeb2661, 0x8b1ddf84,
0x846a0e79, 0x915f95e2, 0x466e598e, 0x20b45770, 0x8cd55591, 0xc902de4c,
0xb90bace1, 0xbb8205d0, 0x11a86248, 0x7574a99e, 0xb77f19b6, 0xe0a9dc09,
0x662d09a1, 0xc4324633, 0xe85a1f02, 0x09f0be8c, 0x4a99a025, 0x1d6efe10,
0x1ab93d1d, 0x0ba5a4df, 0xa186f20f, 0x2868f169, 0xdcb7da83, 0x573906fe,
0xa1e2ce9b, 0x4fcd7f52, 0x50115e01, 0xa70683fa, 0xa002b5c4, 0x0de6d027,
0x9af88c27, 0x773f8641, 0xc3604c06, 0x61a806b5, 0xf0177a28, 0xc0f586e0,
0x006058aa, 0x30dc7d62, 0x11e69ed7, 0x2338ea63, 0x53c2dd94, 0xc2c21634,
0xbbcbee56, 0x90bcb6de, 0xebfc7da1, 0xce591d76, 0x6f05e409, 0x4b7c0188,
0x39720a3d, 0x7c927c24, 0x86e3725f, 0x724d9db9, 0x1ac15bb4, 0xd39eb8fc,
0xed545578, 0x08fca5b5, 0xd83d7cd3, 0x4dad0fc4, 0x1e50ef5e, 0xb161e6f8,
0xa28514d9, 0x6c51133c, 0x6fd5c7e7, 0x56e14ec4, 0x362abfce, 0xddc6c837,
0xd79a3234, 0x92638212, 0x670efa8e, 0x406000e0 };

static const unsigned long blow3[] = {
0x3a39ce37, 0xd3faf5cf, 0xabc27737, 0x5ac52d1b, 0x5cb0679e, 0x4fa33742,
0xd3822740, 0x99bc9bbe, 0xd5118e9d, 0xbf0f7315, 0xd62d1c7e, 0xc700c47b,
0xb78c1b6b, 0x21a19045, 0xb26eb1be, 0x6a366eb4, 0x5748ab2f, 0xbc946e79,
0xc6a376d2, 0x6549c2c8, 0x530ff8ee, 0x468dde7d, 0xd5730a1d, 0x4cd04dc6,
0x2939bbdb, 0xa9ba4650, 0xac9526e8, 0xbe5ee304, 0xa1fad5f0, 0x6a2d519a,
0x63ef8ce2, 0x9a86ee22, 0xc089c2b8, 0x43242ef6, 0xa51e03aa, 0x9cf2d0a4,
0x83c061ba, 0x9be96a4d, 0x8fe51550, 0xba645bd6, 0x2826a2f9, 0xa73a3ae1,
0x4ba99586, 0xef5562e9, 0xc72fefd3, 0xf752f7da, 0x3f046f69, 0x77fa0a59,
0x80e4a915, 0x87b08601, 0x9b09e6ad, 0x3b3ee593, 0xe990fd5a, 0x9e34d797,
0x2cf0b7d9, 0x022b8b51, 0x96d5ac3a, 0x017da67d, 0xd1cf3ed6, 0x7c7d2d28,
0x1f9f25cf, 0xadf2b89b, 0x5ad6b472, 0x5a88f54c, 0xe029ac71, 0xe019a5e6,
0x47b0acfd, 0xed93fa9b, 0xe8d3c48d, 0x283b57cc, 0xf8d56629, 0x79132e28,
0x785f0191, 0xed756055, 0xf7960e44, 0xe3d35e8c, 0x15056dd4, 0x88f46dba,
0x03a16125, 0x0564f0bd, 0xc3eb9e15, 0x3c9057a2, 0x97271aec, 0xa93a072a,
0x1b3f6d9b, 0x1e6321f5, 0xf59c66fb, 0x26dcf319, 0x7533d928, 0xb155fdf5,
0x03563482, 0x8aba3cbb, 0x28517711, 0xc20ad9f8, 0xabcc5167, 0xccad925f,
0x4de81751, 0x3830dc8e, 0x379d5862, 0x9320f991, 0xea7a90c2, 0xfb3e7bce,
0x5121ce64, 0x774fbe32, 0xa8b6e37e, 0xc3293d46, 0x48de5369, 0x6413e680,
0xa2ae0810, 0xdd6db224, 0x69852dfd, 0x09072166, 0xb39a460a, 0x6445c0dd,
0x586cdecf, 0x1c20c8ae, 0x5bbef7dd, 0x1b588d40, 0xccd2017f, 0x6bb4e3bb,
0xdda26a7e, 0x3a59ff45, 0x3e350a44, 0xbcb4cdd5, 0x72eacea8, 0xfa6484bb,
0x8d6612ae, 0xbf3c6f47, 0xd29be463, 0x542f5d9e, 0xaec2771b, 0xf64e6370,
0x740e0d8d, 0xe75b1357, 0xf8721671, 0xaf537d5d, 0x4040cb08, 0x4eb4e2cc,
0x34d2466a, 0x0115af84, 0xe1b00428, 0x95983a1d, 0x06b89fb4, 0xce6ea048,
0x6f3f3b82, 0x3520ab82, 0x011a1d4b, 0x277227f8, 0x611560b1, 0xe7933fdc,
0xbb3a792b, 0x344525bd, 0xa08839e1, 0x51ce794b, 0x2f32c9b7, 0xa01fbac9,
0xe01cc87e, 0xbcc7d1f6, 0xcf0111c3, 0xa1e8aac7, 0x1a908749, 0xd44fbd9a,
0xd0dadecb, 0xd50ada38, 0x0339c32a, 0xc6913667, 0x8df9317c, 0xe0b12b4f,
0xf79e59b7, 0x43f5bb3a, 0xf2d519ff, 0x27d9459c, 0xbf97222c, 0x15e6fc2a,
0x0f91fc71, 0x9b941525, 0xfae59361, 0xceb69ceb, 0xc2a86459, 0x12baa8d1,
0xb6c1075e, 0xe3056a0c, 0x10d25065, 0xcb03a442, 0xe0ec6e0e, 0x1698db3b,
0x4c98a0be, 0x3278e964, 0x9f1f9532, 0xe0d392df, 0xd3a0342b, 0x8971f21e,
0x1b0a7441, 0x4ba3348c, 0xc5be7120, 0xc37632d8, 0xdf359f8d, 0x9b992f2e,
0xe60b6f47, 0x0fe3f11d, 0xe54cda54, 0x1edad891, 0xce6279cf, 0xcd3e7e6f,
0x1618b166, 0xfd2c1d05, 0x848fd2c5, 0xf6fb2299, 0xf523f357, 0xa6327623,
0x93a83531, 0x56cccd02, 0xacf08162, 0x5a75ebb5, 0x6e163697, 0x88d273cc,
0xde966292, 0x81b949d0, 0x4c50901b, 0x71c65614, 0xe6c6c7bd, 0x327a140a,
0x45e1d006, 0xc3f27b9a, 0xc9aa53fd, 0x62a80f00, 0xbb25bfe2, 0x35bdd2f6,
0x71126905, 0xb2040222, 0xb6cbcf7c, 0xcd769c2b, 0x53113ec0, 0x1640e3d3,
0x38abbd60, 0x2547adf0, 0xba38209c, 0xf746ce76, 0x77afa1c5, 0x20756060,
0x85cbfe4e, 0x8ae88dd8, 0x7aaaf9b0, 0x4cf9aa7e, 0x1948c25c, 0x02fb8a8c,
0x01c36ae4, 0xd6ebe1f9, 0x90d4f869, 0xa65cdea0, 0x3f09252d, 0xc208e69f,
0xb74e6132, 0xce77e25b, 0x578fdfe3, 0x3ac372e6 };

/* RMD hash function. This function is adopted from Antoon Bosselaers (1996).
   ftp://ftp.funet.fi/pub/crypt/hash/ripemd/ */

unsigned char *RMD(unsigned char *message)
/*
 * returns RMD(message)
 * message should be a string terminated by '\0'
 */
{
   unsigned long MDbuf[RMDsize/32];          /* contains (A, B, C, D(, E))   */
   static unsigned char hashcode[RMDsize/8]; /* for final hash-value         */
   dword X[16];                              /* current 16-word chunk        */
   int i;                                    /* counter                      */
   unsigned int length;                      /* length in bytes of message   */
   unsigned int nbytes;                      /* # of bytes not yet processed */

   /* initialize */
   MDinit(MDbuf);
   length = (dword)strlen((char *)message);

   /* process message in 16-word chunks */
   for (nbytes=length; nbytes > 63; nbytes-=64) {
      for (i=0; i<16; i++) {
         X[i] = BYTES_TO_DWORD(message);
         message += 4;
      }
      compress(MDbuf, X);
   }                                    /* length mod 64 bytes left */

   /* finish: */
   MDfinish(MDbuf, message, length, 0);
   for (i=0; i<RMDsize/8; i+=4) {
      hashcode[i]   =  MDbuf[i>>2];         /* implicit cast to byte  */
      hashcode[i+1] = (MDbuf[i>>2] >>  8);  /*  extracts the 8 least  */
      hashcode[i+2] = (MDbuf[i>>2] >> 16);  /*  significant bits.     */
      hashcode[i+3] = (MDbuf[i>>2] >> 24);
   }

   return (byte *)hashcode;
}
/* End code from Bosselaers */

/* feedback functions */
static unsigned char *crl(unsigned char *m1) /* rotate left */
 {
   register unsigned long bit1,bit2;
   int i;
   unsigned char *tm;
   tm=m1;
   bit1=BYTE_TO_UL(m1);
   bit2=BYTE_TO_UL(m1+4);
   ROL(bit1,key_schedule[iround][0]%32);
   ROL(bit2,key_schedule[iround][1]%32);
   m1=tm;
   for(i=0;i<=3;i++)
     {
       *(m1++)=bit1;
       bit1>>=8;
     }
   for(i=0;i<=3;i++)
     {
       *(m1++)=bit2;
       bit2>>=8;
     }
   m1=tm;
   return(m1);
 }

static unsigned char *crr(unsigned char *m1) /* rotate right */
 {
   register unsigned long bit1,bit2;
   int i;
   unsigned char *tm;
   tm=m1;
   bit1=BYTE_TO_UL(m1);
   bit2=BYTE_TO_UL(m1+4);
   ROR(bit1,key_schedule[iround][1]%32);
   ROR(bit2,key_schedule[iround][2]%32);
   m1=tm;
   for(i=0;i<=3;i++)
     {
       *(m1++)=bit1;
       bit1>>=B;
     }
   for(i=0;i<=3;i++)
     {
       *(m1++)=bit2;
       bit2>>=8;
     }
   m1=tm;
   return(m1);
 }

static unsigned char *hash1(unsigned char *m1) /* Benes-based hash */
 {
   register unsigned long tbit1,bit1,bit2;
   int i;
   unsigned char *tm;
   tm=m1;
   tbit1=bit1=BYTE_TO_UL(m1);
   bit2=BYTE_TO_UL(m1+4);
   bit1+=blow0[((unsigned long)key_schedule[iround][3]*bit2)%0xff];
   bit2+=blow2[((unsigned long)key_schedule[iround][4]*tbit1)%0xff];
   m1=tm;
   for(i=0;i<=3;i++)
     {
       *(m1++)=bit1;
       bit1>>=8;
     }
   for(i=0;i<=3;i++)
     {
       *(m1++)=bit2;
       bit2>>=8;
     }
   m1=tm;
   return(m1);
 }

static unsigned char *modmul(unsigned char *m1) /*multiplication mod(2^32)*/
 {
   register unsigned long bit1,bit2;
   int i;
   unsigned char *tm;
   tm=m1;
   bit1=BYTE_TO_UL(m1);
   bit2=BYTE_TO_UL(m1+4);
   bit1*=BYTE_TO_UL(&key_schedule[iround][5]);
   bit2*=BYTE_TO_UL(&key_schedule[iround][6]);
   m1=tm;
   for(i=0;i<=3;i++)
     {
       *(m1++)=bit1;
       bit1>>=8;
     }
   for(i=0;i<=3;i++)
     {
       *(m1++)=bit2;
       bit2>>=8;
     }
   m1=tm;
   return(m1);
 }

static unsigned char *modadd(unsigned char *m1) /* addition mod(2^32)*/
 {
   register unsigned long bit1,bit2;
   int i;
   unsigned char *tm;
   tm=m1;
   bit1=BYTE_TO_UL(m1);
   bit2=BYTE_TO_UL(m1+4);
   bit1+=BYTE_TO_UL(&key_schedule[iround][6]);
   bit2+=BYTE_TO_UL(&key_schedule[iround][7]);
   m1=tm;
   for(i=0;i<=3;i++)
     {
       *(m1++)=bit1;
       bit1>>=8;
     }
   for(i=0;i<=3;i++)
     {
       *(m1++)=bit2;
       bit2>>=8;
     }
   m1=tm;
   return(m1);
 }

/* encryption steps */

static unsigned char *permute1(unsigned char *m1)  /* permutation on byte level*/
{
  register int i;
  unsigned char *tm;
  unsigned char tmap[7];
  unsigned char map[] = {2, 0, 7, 3, 6, 4, 1, 5};
  tm=m1;
  for(i=0;i<=7;i++)
    {
      tmap[i]=*(m1+map[i]);
      m1=tm;
    }
  for(i=0;i<=7;i++)
    *(m1++)=tmap[i];
  m1=tm;
  return(m1);
}

static unsigned char *ipermute1(unsigned char *m1) /* decryption step */
{
  register int i;
  unsigned char *tm;
  unsigned char tmap[7];
  unsigned char map[] = {1, 6, 0, 3, 5, 7, 4, 2};
  tm=m1;
  for(i=0;i<=7;i++)
    {
      tmap[i]=*(m1+map[i]);
      m1=tm;
    }
  for(i=0;i<=7;i++)
    *(m1++)=tmap[i];
  m1=tm;
  return(m1);
}

static unsigned char *permute2(unsigned char *m1) /* another permutation */
{
  register int i;
  unsigned char *tm;
  unsigned char tmap[7];
  unsigned char map[] = {7, 0, 3, 1, 5, 6, 2, 4};
  tm=m1;
  for(i=0;i<=7;i++)
    {
      tmap[i]=*(m1+map[i]);
      m1=tm;
    }
  for(i=0;i<=7;i++)
    *(m1++)=tmap[i];
  m1=tm;
  return(m1);
}
```

```
static unsigned char *ipermute2(unsigned char *m1) /* decryption step */
{
  register int i;
  unsigned char *tm;
  unsigned char tmap[7];
  unsigned char map[] = {1, 3, 6, 2, 7, 4, 5, 0};
  tm=m1;
  for(i=0;i<=7;i++)
    {
      tmap[i]=*(m1+map[i]);
      m1=tm;
    }
  for(i=0;i<=7;i++)
    *(m1++)=tmap[i];
  m1=tm;
  return(m1);
}

static unsigned char *vigenere(unsigned char *m1) /* vigenere substitution */
{
  register int i;
  unsigned char *tm;
  tm=m1;
  for(i=0;i<=7;i++)
    {
      *m1=(*m1+key_schedule[iround][i])%256;
      m1++;
    }
    m1=tm;
  return(m1);
}

static unsigned char *ivigenere(unsigned char *m1) /* decryption step */
{
  register int i;
  unsigned char *tm;
  tm=m1;
  for(i=0;i<=7;i++)
    {
      *m1=(*m1-key_schedule[iround][i])%256;
      m1++;
    }
      m1=tm;
  return(m1);
}

static unsigned char *des1(unsigned char *m1) /* one-round DES step */
{
  register int i;
  register unsigned long w,u,t;
  static unsigned long *s;
  static unsigned char *tm;
  register unsigned long bit1,bit2;
  tm=m1;
  s=(unsigned long *)ks;
  bit2=BYTE_TO_UL(m1);
  bit1=BYTE_TO_UL(m1+4);
  D_ENCRYPT(bit2,bit1,iround);            /* call to Young's code*/
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit2;
      bit2>>=8;
    }
  m1=tm;
  return(m1);
}

static unsigned char *ddes1(unsigned char *m1) /* decryption step */
{
  register int i;
  register unsigned long w,*s,u,t;
  unsigned char *tm;
  register unsigned long bit1,bit2;
  tm=m1;
  s=(unsigned long *)ks;
  bit2=BYTE_TO_UL(m1);
  bit1=BYTE_TO_UL(m1+4);
  D_ENCRYPT(bit1,bit2,iround);
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit2;
      bit2>>=8;
    }
  m1=tm;
  return(m1);
}

static unsigned char *elr1(unsigned char *m1)   /* Feistel #1 */
{
  int i;
  unsigned long int w=0;
  static unsigned char *tm;
  unsigned long tbit1,tbit2,bit1,bit2;
  register unsigned long t1;

/*set up values according to RIPEM Hash function */
  register unsigned long aa,bb,cc,dd;
  register unsigned long aaa,bbb,ccc,ddd;
  unsigned long MDbuf[4];

  MDbuf[0] = 0x67452301UL;
  MDbuf[1] = 0xefcdab89UL;
  MDbuf[2] = 0x98badcfeUL;
  MDbuf[3] = 0x10325476UL;

  aa=aaa=MDbuf[0];
  bb=bbb=MDbuf[1];
  cc=ccc=MDbuf[2];
  dd=ddd=MDbuf[3];


  tm=m1;
  tbit2=bit1=BYTE_TO_UL(m1);
  t1=tbit1=bit2=BYTE_TO_UL(m1+4);

  /*round 1 of RMD */
  FF(aa, bb, cc, dd, t1, 11);
  FF(dd, aa, bb, cc, t1, 14);
  FF(cc, dd, aa, bb, t1, 15);
  FF(bb, cc, dd, aa, t1, 12);
  FF(aa, bb, cc, dd, t1,  5);
  FF(dd, aa, bb, cc, t1,  8);
  FF(cc, dd, aa, bb, t1,  7);
  FF(bb, cc, dd, aa, t1,  9);
  FF(aa, bb, cc, dd, t1, 11);
  FF(dd, aa, bb, cc, t1, 13);
  FF(cc, dd, aa, bb, t1, 14);
  FF(bb, cc, dd, aa, t1, 15);
  FF(aa, bb, cc, dd, t1,  6);
  FF(dd, aa, bb, cc, t1,  7);
  FF(cc, dd, aa, bb, t1,  9);
  FF(bb, cc, dd, aa, t1,  8);

  /* parallel round 3 of RMD */
  GGG(aaa, bbb, ccc, ddd, t1,  9);
  GGG(ddd, aaa, bbb, ccc, t1,  7);
  GGG(ccc, ddd, aaa, bbb, t1, 15);
  GGG(bbb, ccc, ddd, aaa, t1, 11);
  GGG(aaa, bbb, ccc, ddd, t1,  8);
  GGG(ddd, aaa, bbb, ccc, t1,  6);
  GGG(ccc, ddd, aaa, bbb, t1,  6);
  GGG(bbb, ccc, ddd, aaa, t1, 14);
  GGG(aaa, bbb, ccc, ddd, t1, 12);
  GGG(ddd, aaa, bbb, ccc, t1, 13);
  GGG(ccc, ddd, aaa, bbb, t1,  5);
  GGG(bbb, ccc, ddd, aaa, t1, 14);
  GGG(aaa, bbb, ccc, ddd, t1, 13);
  GGG(ddd, aaa, bbb, ccc, t1, 13);
  GGG(ccc, ddd, aaa, bbb, t1,  7);
  GGG(bbb, ccc, ddd, aaa, t1,  5);

  ddd += cc + MDbuf[1];               /* final result for MDbuf[0] */
  MDbuf[1] = MDbuf[2] + dd + aaa;
  MDbuf[2] = MDbuf[3] + aa + bbb;
  MDbuf[3] = MDbuf[0] + bb + ccc;
  MDbuf[0] = ddd;

  t1=MDbuf[0]+MDbuf[1]+MDbuf[2]+MDbuf[3];
  bit2=(t1)^tbit2;
  bit1=tbit1;
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit2;
      bit2>>=8;
    }
  m1=tm;
  return(m1);
}

static unsigned char *dlr1(unsigned char *m1) /* decryption step */
{
  int i;
  unsigned int w=0;
```

```
 unsigned char *tm;
 unsigned long tbit1,tbit2,bit1,bit2;
 register unsigned long t1;

/*set up values according to RIPEM Hash function */
 register unsigned long aa,bb,cc,dd;
 register unsigned long aaa,bbb,ccc,ddd;
 unsigned long MDbuf[4];

 MDbuf[0] = 0x67452301UL;
 MDbuf[1] = 0xefcdab89UL;
 MDbuf[2] = 0x98badcfeUL;
 MDbuf[3] = 0x10325476UL;

 aa=aaa=MDbuf[0];
 bb=bbb=MDbuf[1];
 cc=ccc=MDbuf[2];
 dd=ddd=MDbuf[3];
 tm=m1;
 t1=tbit2=bit1=BYTE_TO_UL(m1);
 tbit1=bit2=BYTE_TO_UL(m1+4);

 /*round 1 of RMD */
 FF(aa, bb, cc, dd, t1, 11);
 FF(dd, aa, bb, cc, t1, 14);
 FF(cc, dd, aa, bb, t1, 15);
 FF(bb, cc, dd, aa, t1, 12);
 FF(aa, bb, cc, dd, t1,  5);
 FF(dd, aa, bb, cc, t1,  8);
 FF(cc, dd, aa, bb, t1,  7);
 FF(bb, cc, dd, aa, t1,  9);
 FF(aa, bb, cc, dd, t1, 11);
 FF(dd, aa, bb, cc, t1, 13);
 FF(cc, dd, aa, bb, t1, 14);
 FF(bb, cc, dd, aa, t1, 15);
 FF(aa, bb, cc, dd, t1,  6);
 FF(dd, aa, bb, cc, t1,  7);
 FF(cc, dd, aa, bb, t1,  9);
 FF(bb, cc, dd, aa, t1,  8);

 /* parallel round 3 of RMD */
 GGG(aaa, bbb, ccc, ddd, t1,  9);
 GGG(ddd, aaa, bbb, ccc, t1,  7);
 GGG(ccc, ddd, aaa, bbb, t1, 15);
 GGG(bbb, ccc, ddd, aaa, t1, 11);
 GGG(aaa, bbb, ccc, ddd, t1,  8);
 GGG(ddd, aaa, bbb, ccc, t1,  6);
 GGG(ccc, ddd, aaa, bbb, t1,  6);
 GGG(bbb, ccc, ddd, aaa, t1, 14);
 GGG(aaa, bbb, ccc, ddd, t1, 12);
 GGG(ddd, aaa, bbb, ccc, t1, 13);
 GGG(ccc, ddd, aaa, bbb, t1,  5);
 GGG(bbb, ccc, ddd, aaa, t1, 14);
 GGG(aaa, bbb, ccc, ddd, t1, 13);
 GGG(ddd, aaa, bbb, ccc, t1, 13);
 GGG(ccc, ddd, aaa, bbb, t1,  7);
 GGG(bbb, ccc, ddd, aaa, t1,  5);

 ddd += cc + MDbuf[1];                  /* final result for MDbuf[0] */
 MDbuf[1] = MDbuf[2] + dd + aaa;
 MDbuf[2] = MDbuf[3] + aa + bbb;
 MDbuf[3] = MDbuf[0] + bb + ccc;
 MDbuf[0] = ddd;

 t1=MDbuf[0]+MDbuf[1]+MDbuf[2]+MDbuf[3];
 bit1=tbit1^(t1);
 bit2=tbit2;
 m1=tm;
 for(i=0;i<=3;i++)
   {
     *(m1++)=bit1;
     bit1>>=8;
   }
 for(i=0;i<=3;i++)
   {
     *(m1++)=bit2;
     bit2>>=8;
   }
 m1=tm;
 return(m1);
}

static unsigned char *elr2(unsigned char *m1)  /* Feistel #2 */
{
  int i;
  unsigned long int w=0;
  unsigned char *tm;
  unsigned long tbit1,tbit2,bit1,bit2;
  register unsigned long t1;

/* set up values according to RIPEM Hash function */
  register unsigned long aa,bb,cc,dd;
  register unsigned long aaa,bbb,ccc,ddd;
  unsigned long MDbuf[4];

  MDbuf[0] = 0x67452301UL;
```

```
MDbuf[1] = 0xefcdab89UL;
MDbuf[2] = 0x98badcfeUL;
MDbuf[3] = 0x10325476UL;

aa=aaa=MDbuf[0];
bb=bbb=MDbuf[1];
cc=ccc=MDbuf[2];
dd=ddd=MDbuf[3];

tm=m1;
t1=tbit2=bit1=BYTE_TO_UL(m1);
tbit1=bit2=BYTE_TO_UL(m1+4);

/*round 4 of RMD */
II(aa, bb, cc, dd, t1, 11);
II(dd, aa, bb, cc, t1, 12);
II(cc, dd, aa, bb, t1, 14);
II(bb, cc, dd, aa, t1, 15);
II(aa, bb, cc, dd, t1, 14);
II(dd, aa, bb, cc, t1, 15);
II(cc, dd, aa, bb, t1,  9);
II(bb, cc, dd, aa, t1,  8);
II(aa, bb, cc, dd, t1,  9);
II(dd, aa, bb, cc, t1, 14);
II(cc, dd, aa, bb, t1,  5);
II(bb, cc, dd, aa, t1,  6);
II(aa, bb, cc, dd, t1,  8);
II(dd, aa, bb, cc, t1,  6);
II(cc, dd, aa, bb, t1,  5);
II(bb, cc, dd, aa, t1, 12);

/* parallel round 2 of RMD */
HHH(aaa, bbb, ccc, ddd, t1,  9);
HHH(ddd, aaa, bbb, ccc, t1, 13);
HHH(ccc, ddd, aaa, bbb, t1, 15);
HHH(bbb, ccc, ddd, aaa, t1,  7);
HHH(aaa, bbb, ccc, ddd, t1, 12);
HHH(ddd, aaa, bbb, ccc, t1,  8);
HHH(ccc, ddd, aaa, bbb, t1,  9);
HHH(bbb, ccc, ddd, aaa, t1, 11);
HHH(aaa, bbb, ccc, ddd, t1,  7);
HHH(ddd, aaa, bbb, ccc, t1,  7);
HHH(ccc, ddd, aaa, bbb, t1, 12);
HHH(bbb, ccc, ddd, aaa, t1,  7);
HHH(aaa, bbb, ccc, ddd, t1,  6);
HHH(ddd, aaa, bbb, ccc, t1, 15);
HHH(ccc, ddd, aaa, bbb, t1, 13);
HHH(bbb, ccc, ddd, aaa, t1, 11);

ddd += cc + MDbuf[1];               /* final result for MDbuf[0] */
MDbuf[1] = MDbuf[2] + dd + aaa;
MDbuf[2] = MDbuf[3] + aa + bbb;
MDbuf[3] = MDbuf[0] + bb + ccc;
MDbuf[0] = ddd;

t1=MDbuf[0]+MDbuf[1]+MDbuf[2]+MDbuf[3];
bit1=(t1)^tbit1;
bit2=tbit2;
m1=tm;
for(i=0;i<=3;i++)
  {
    *(m1++)=bit1;
    bit1>>=8;
  }
for(i=0;i<=3;i++)
  {
    *(m1++)=bit2;
    bit2>>=8;
  }
m1=tm;
return(m1);
}

static unsigned char *dlr2(unsigned char *m1) /* decryption step */
{
  int i;
  unsigned int w=0;
  unsigned char *tm;
  unsigned long tbit1,tbit2,bit1,bit2;
  register unsigned long t1;

 /*set up values according to RIPEM Hash function */
  register unsigned long aa,bb,cc,dd;
  register unsigned long aaa,bbb,ccc,ddd;
  unsigned long MDbuf[4];

  MDbuf[0] = 0x67452301UL;
  MDbuf[1] = 0xefcdab89UL;
  MDbuf[2] = 0x98badcfeUL;
  MDbuf[3] = 0x10325476UL;

  aa=aaa=MDbuf[0];
  bb=bbb=MDbuf[1];
  cc=ccc=MDbuf[2];
  dd=ddd=MDbuf[3];
```

```
tm=m1;
tbit2=bit1=BYTE_TO_UL(m1);
t1=tbit1=bit2=BYTE_TO_UL(m1+4);

/* round 4 of RMD */
II(aa, bb, cc, dd, t1, 11);
II(dd, aa, bb, cc, t1, 12);
II(cc, dd, aa, bb, t1, 14);
II(bb, cc, dd, aa, t1, 15);
II(aa, bb, cc, dd, t1, 14);
II(dd, aa, bb, cc, t1, 15);
II(cc, dd, aa, bb, t1,  9);
II(bb, cc, dd, aa, t1,  8);
II(aa, bb, cc, dd, t1,  9);
II(dd, aa, bb, cc, t1, 14);
II(cc, dd, aa, bb, t1,  5);
II(bb, cc, dd, aa, t1,  6);
II(aa, bb, cc, dd, t1,  8);
II(dd, aa, bb, cc, t1,  6);
II(cc, dd, aa, bb, t1,  5);
II(bb, cc, dd, aa, t1, 12);

/* parallel round 2 of RMD */
HHH(aaa, bbb, ccc, ddd, t1,  9);
HHH(ddd, aaa, bbb, ccc, t1, 13);
HHH(ccc, ddd, aaa, bbb, t1, 15);
HHH(bbb, ccc, ddd, aaa, t1,  7);
HHH(aaa, bbb, ccc, ddd, t1, 12);
HHH(ddd, aaa, bbb, ccc, t1,  8);
HHH(ccc, ddd, aaa, bbb, t1,  9);
HHH(bbb, ccc, ddd, aaa, t1, 11);
HHH(aaa, bbb, ccc, ddd, t1,  7);
HHH(ddd, aaa, bbb, ccc, t1,  7);
HHH(ccc, ddd, aaa, bbb, t1, 12);
HHH(bbb, ccc, ddd, aaa, t1,  7);
HHH(aaa, bbb, ccc, ddd, t1,  6);
HHH(ddd, aaa, bbb, ccc, t1, 15);
HHH(ccc, ddd, aaa, bbb, t1, 13);
HHH(bbb, ccc, ddd, aaa, t1, 11);

ddd += cc + MDbuf[1];                /* final result for MDbuf[0] */
MDbuf[1] = MDbuf[2] + dd + aaa;
MDbuf[2] = MDbuf[3] + aa + bbb;
MDbuf[3] = MDbuf[0] + bb + ccc;
MDbuf[0] = ddd;

t1=MDbuf[0]+MDbuf[1]+MDbuf[2]+MDbuf[3];
bit2=tbit2^(t1);
bit1=tbit1;
m1=tm;
for(i=0;i<=3;i++)
  {
    *(m1++)=bit1;
    bit1>>=8;
  }
for(i=0;i<=3;i++)
  {
    *(m1++)=bit2;
    bit2>>=8;
  }
m1=tm;
return(m1);
}

static unsigned char *eblow_r(unsigned char *m1) /* Blowfish R */
{
  int i;
  unsigned char *tm;
  unsigned long tbit1,tbit2,bit1,bit2;
  unsigned long t1;


  tm=m1;
  tbit2=bit1=BYTE_TO_UL(m1);
  t1=tbit1=bit2=BYTE_TO_UL(m1+4);

  t1=(blow0[key_schedule[iround][0]^(*(m1+4))]+blow1[key_schedule[iround][1]^(*(m1+5))])^
    blow2[key_schedule[iround][2]^(*(m1+6))]+blow3[key_schedule[iround][3]^(*(m1+7))];

  bit2=(t1)^tbit2;
  bit1=tbit1;
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit2;
      bit2>>=8;
    }
  m1=tm;
  return(m1);
}
```

```
static unsigned char *dblow_r(unsigned char *m1) /* decryption step */
{
  int i;
  unsigned char *tm;
  unsigned long tbit1,tbit2,bit1,bit2;
  unsigned long t1;

  tm=m1;
  t1=tbit2=bit1=BYTE_TO_UL(m1);
  tbit1=bit2=BYTE_TO_UL(m1+4);

  t1=(blow0[key_schedule[iround][0]^(*m1)]+blow1[key_schedule[iround][1]^(*(m1+1))])^
    blow2[key_schedule[iround][2]^(*(m1+2))]+blow3[key_schedule[iround][3]^(*(m1+3))];

  bit1=tbit1^(t1);
  bit2=tbit2;
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit2;
      bit2>>=8;
    }
  m1=tm;
  return(m1);
}

static unsigned char *eufn(unsigned char *m1) /* encryption of UFN 16:48 */
{
  int i;
  unsigned int w=0;
  unsigned char *tm;
  unsigned int source,tbit1,bit2;  /* the target would be bit1 o bit2 */
  unsigned long tbit2,bit1;
  unsigned long t1,t2;
  unsigned int s1,s2;

  tm=m1;                         /*store address of m1 */
  tbit2=bit1=8YTE_TO_UL(m1);
  tbit1=bit2=BYTE_TO_UI(m1+4);
  source=BYTE_TO_UI(m1+6);

  s1=ROL16(source,6);            /* the two rotations by 6 and 8 bits increase */
  s2=ROL16(s1,8);                /* diffusion of the input to the round function */
  t1=(blow0[key_schedule[iround][2]^(unsigned char)s1])^(blow1[key_schedule[iround][4]^(unsigned char)s2]);
  t2=(blow2[key_schedule[iround][3]^(unsigned char)(s1>>8)])^(blow3[key_schedule[iround][5]^(unsigned char)(s2>>8)]);
  bit1^=t1;
  bit2^=t2;
  m1=tm;
  for(i=0;i<=1;i++)
    {
      *(m1++)=source;
      source>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  for(i=0;i<=1;i++)
    {
      *(m1++)=bit2;
      bit2>>=8;
    }
  m1=tm;
  return(m1);
}

static unsigned char *dufn(unsigned char *m1) /* decryption step */
{
  int i;
  unsigned int w=0;
  unsigned char *tm;
  unsigned int source,tbit1,bit2;  /* the target would be bit1 o bit2 */
  unsigned long tbit2,bit1;
  unsigned long t1,t2;
  unsigned int s1,s2;

  tm=m1;                         /*store address of m1 */
  bit1=BYTE_TO_UL(m1+2);
  bit2=BYTE_TO_UI(m1+6);
  source=BYTE_TO_UI(m1);
  s1=ROL16(source,6);
  s2=ROL16(s1,8);
  t1=(blow0[key_schedule[iround][2]^(unsigned char)s1])^(blow1[key_schedule[iround][4]^(unsigned char)s2]);
  t2=(blow2[key_schedule[iround][3]^(unsigned char)(s1>>8)])^(blow3[key_schedule[iround][5]^(unsigned char)(s2>>8)]);
  bit1^=t1;
  bit2^=t2;
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  for(i=0;i<=1;i++)
    {
      *(m1++)=bit2;
      bit2>>=8;
    }
  for(i=0;i<=1;i++)
    {
      *(m1++)=source;
      source>>=8;
    }
  m1=tm;
  return(m1);
}

static unsigned char *eufn40_24(unsigned char *m1)/* encryption of UFN 40:24 */
{
  int i;
  unsigned int bit1;
  unsigned char *tm;
  unsigned char bit2,bit3;
  unsigned long bit4;
  unsigned long t1;
  unsigned char hash_input[5];

  tm=m1;
  bit1=BYTE_TO_UI(m1);
  bit2=*(m1+2);
  bit3=*(m1+3);
  bit4=BYTE_TO_UL(m1+4);
    hash_input[0]=bit3;
  for(i=0;i<4;i++)
    hash_input[i+1]=bit4>>(8*i);
  t1=(blow0[key_schedule[iround][0]^hash_input[0]]+
    blow1[key_schedule[iround][1]^hash_input[1]])^
    (blow2[key_schedule[iround][2]^hash_input[2]]+
    blow3[key_schedule[iround][3]^hash_input[3]])+
    blow2[key_schedule[iround][4]^hash_input[4]];

  bit2^=(unsigned char) t1;
  t1>>=8;
  bit1^=(unsigned int) t1;
  m1=tm;
  *(m1++)=bit3;
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit4;
      bit4>>=8;
    }
  for(i=0;i<=1;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  *(m1++)=bit2;
  m1=tm;
  return(m1);
}

static unsigned char *dufn40_24(unsigned char *m1) /* decryption step */
{
  int i;
  unsigned int bit1;
  unsigned char *tm;
  unsigned char bit2,bit3;
  unsigned long bit4;
  unsigned long t1;
  unsigned char hash_input[5];

  tm=m1;
  bit1=BYTE_TO_UI(m1+5);
  bit2=*(m1+7);
  bit3=*(m1);
  bit4=BYTE_TO_UL(m1+1);

  hash_input[0]=bit3;
  for(i=0;i<4;i++)
    hash_input[i+1]=bit4>>(8*i);
   t1=(blow0[key_schedule[iround][0]^hash_input[0]]+
    blow1[key_schedule[iround][1]^hash_input[1]])^
    (blow2[key_schedule[iround][2]^hash_input[2]]+
    blow3[key_schedule[iround][3]^hash_input[3]])+
    blow2[key_schedule[iround][4]^hash_input[4]];

  bit2^=(unsigned char) t1;
  t1>>=8;
  bit1^=(unsigned int) t1;
  m1=tm;
  for(i=0;i<=1;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  *(m1++)=bit2;
  *(m1++)=bit3;
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit4;
      bit4>>=8;
    }
  m1=tm;
  return(m1);
}

static unsigned char *eblow_l(unsigned char *m1) /* Blowfish L*/
{
  int i;
  unsigned long w=0;
  unsigned char *tm;
  unsigned long tbit1,tbit2,bit1,bit2;
  unsigned long t1;

  tm=m1;
  tbit2=bit1=BYTE_TO_UL(m1);
  t1=tbit1=bit2=BYTE_TO_UL(m1+4);
  t1=(blow0[key_schedule[iround][0]^(*m1)]+blow1[key_schedule[iround][1]^(*(m1+1))])^
    blow2[key_schedule[iround][2]^(*(m1+2))]+blow3[key_schedule[iround][3]^(*(m1+3))];

  bit1=(t1)^tbit1;
  bit2=tbit2;
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit2;
      bit2>>=8;
    }
  m1=tm;
  return(m1);
}

static unsigned char *dblow_l(unsigned char *m1)/* encryption of UFN 16:48 */
{
  int i;
  unsigned int w=0;
  unsigned char *tm;
  unsigned long tbit1,tbit2,bit1,bit2;
  unsigned long t1;

  tm=m1;
  t1=tbit2=bit1=BYTE_TO_UL(m1);
  tbit1=bit2=BYTE_TO_UL(m1+4);
  t1=(blow0[key_schedule[iround][0]^(*(m1+4))]+blow1[key_schedule[iround][1]^(*(m1+5))])^
    blow2[key_schedule[iround][2]^(*(m1+6))]+blow3[key_schedule[iround][3]^(*(m1+7))];
  bit2=tbit2^(t1);
  bit1=tbit1;
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit2;
      bit2>>=8;
    }
  m1=tm;
  return(m1);
}

static unsigned char *lfsr1(unsigned char *m1) /* LFSR #1 */
{
  register int i;
  unsigned char *tm;
  unsigned long tbit2,bit1,bit2;
  register unsigned long shift_register;
  unsigned int poly[4];
  tm=m1;
  bit1=BYTE_TO_UL(m1);
  shift_register=key_schedule[iround][7]^*(m1+3);
  for(i=3;i<=5;i++)
    {
      shift_register<<=8;
      shift_register^=key_schedule[iround][i]^*(m1+3);
    }
  bit2=BYTE_TO_UL(m1+4);
  tbit2=0;
  for(i=0;i<4;i++)               /* select a primitive polynomial for LFSR instance */
    poly[i]=primitive_poly[4*((8+key_schedule[iround][1]&key_schedule[iround][2]|key_schedule[iround][6])%prim_pol_num)+i];

  /* perform 32 shifts and discard them */
  for(i=0;i<32;i++)
    shift_register=((((shift_register >> poly[0])
      ^(shift_register >> poly[1])
      ^(shift_register >> poly[2])
      ^(shift_register >> poly[3])
      ^shift_register)
     & 0x00000001)
    << 31) |(shift_register >> 1);
  /* perform 32 shifts and apply to right part of the message */
  for(i=0;i<32;i++)
    {
      shift_register=((((shift_register >> poly[0])
^(shift_register >> poly[1])
^(shift_register >> poly[2])
^(shift_register >> poly[3])
^shift_register)
       & 0x00000001)
      << 31) |(shift_register >> 1);
      tbit2|=(shift_register & 0x00000001) << i;
    }
  bit2^=tbit2;
  tbit2=0;
  for(i=0;i<24;i++)    /* obtain the remaining 24 bits required */
    {                  /* to complete the sequence */
      shift_register=((((shift_register >> poly[0])
^(shift_register >> poly[1])
^(shift_register >> poly[2])
^(shift_register >> poly[3])
^shift_register)
       & 0x00000001)
      << 31) |(shift_register >> 1);
      tbit2|=(shift_register & 0x00000001) << i;
    }
  bit1^=tbit2;
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit2;
      bit2>>=8;
    }
  m1=tm;
  return(m1);
}

unsigned char *lfsr2(unsigned char *m1)   /* LFSR #2 */
{
  register int i;
  unsigned char *tm;
  unsigned long tbit1,bit1,bit2;
  register unsigned long shift_register;
  unsigned int poly[4];
  tm=m1;
  bit1=BYTE_TO_UL(m1);
  bit2=BYTE_TO_UL(m1+4);
  shift_register=*(m1+6)^key_schedule[iround][2];
  shift_register<<=B;
  shift_register^=*(m1+7)^key_schedule[iround][4];
  shift_register<<=8;
  tbit1=0;
  for(i=0;i<4;i++)
    poly[i]=primitive_poly[4*((16+key_schedule[iround][0]&key_schedule[iround][3]|key_schedule[iround][5])%prim_pol_num)+i];

  /* perform 32 shifts and discard them */
  for(i=0;i<32;i++)
    shift_register=((((shift_register >> poly[0])
      ^(shift_register >> poly[1])
      ^(shift_register >> poly[2])
      ^(shift_register >> poly[3])
      ^shift_register)
     & 0x00000001)
    << 31) |(shift_register >> 1);
  /* perform 32 shifts and apply to left part of the message */
  for(i=0;i<32;i++)
    {
      shift_register=((((shift_register >> poly[0])
^(shift_register >> poly[1])
^(shift_register >> poly[2])
^(shift_register >> poly[3])
^shift_register)
       & 0x00000001)
      << 31) |(shift_register >> 1);
      tbit1|=(shift_register & 0x00000001) << i;
    }
  tbit1*=tbit1*tbit1;  /* x^3 mod(2^32) */
  bit1^=tbit1;         /* apply sequence to left input block */
  tbit1=0;
  for(i=0;i<16;i++)
    {
      shift_register=((((shift_register >> poly[0])
^(shift_register >> poly[1])
^(shift_register >> poly[2])
^(shift_register >> poly[3])
^shift_register)
       & 0x00000001)
      << 31) |(shift_register >> 1);
      tbit1|=(shift_register & 0x00000001) << i;
    }
  bit2^=(tbit1*tbit1*tbit1)>>16;      /* first two bytes of right input block */
  m1=tm;              /* update m1... */
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit1;
      bit1>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=bit2;
      bit2>>=8;
    }
  m1=tm;
  return(m1);
}

unsigned char *des_ip(unsigned char *m1)
{
  unsigned char *tm;
  int i;
  register unsigned long l,r,t,u;

  tm=m1;
  l=BYTE_TO_UL(m1);
  r=BYTE_TO_UL(m1+4);
  /* code from Eric Young */
  PERM_OP(r,l,t, 4,0x0f0f0f0f);
  PERM_OP(l,r,t,16,0x0000ffff);
  PERM_OP(r,l,t, 2,0x33333333);
  PERM_OP(l,r,t, 8,0x00ff00ff);
  PERM_OP(r,l,t, 1,0x55555555);
  /* r and l are reversed - remember that :-) - fix
   * it in the next step */
  /* Things have been modified so that the initial rotate is
   * done outside the loop.  This required the
   * des_SPtrans values in sp.h to be rotated 1 bit to the right.
   * One perl script later and things have a 5% speed up on a sparc2.
   * Thanks to Richard Outerbridge <71755.204@CompuServe.COM>
   * for pointing this out. */
  r=(r<<1)|(r>>31);
  l=(l<<1)|(l>>31);
 /* end code from Eric Young */
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=l;
      l>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=r;
      r>>=8;
    }
  m1=tm;
  return(m1);
}

unsigned char *des_ip_1(unsigned char *m1) /* DES IP^(-1) */
{
  unsigned char *tm;
  int i;
  register unsigned long l,r,t;

  tm=m1;
  l=BYTE_TO_UL(m1);
  r=BYTE_TO_UL(m1+4);
  /* code from Eric Young */
  l=(l>>1)|(l<<31);
  r=(r>>1)|(r<<31);
  /* swap l and r
   * we will not do the swap so just remember they are
   * reversed for the rest of the subroutine
   * luckily FP fixes this problem :-) */

  PERM_OP(r,l,t, 1,0x55555555);
  PERM_OP(l,r,t, 8,0x00ff00ff);
  PERM_OP(r,l,t, 2,0x33333333);
  PERM_OP(l,r,t,16,0x0000ffff);
  PERM_OP(r,l,t, 4,0x0f0f0f0f);
  /* end code from Eric Young */
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=l;
      l>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=r;
      r>>=8;
    }
  m1=tm;
  return(m1);
}
```

```
unsigned char *des2(unsigned char *m1) /* 2-round DES */
{
  unsigned char *tm;
  register unsigned long *s;
  int i;
  register unsigned long l,r,t,u;

  tm=m1;
  l=BYTE_TO_UL(m1);
  r=BYTE_TO_UL(m1+4);
  s=(unsigned long *) ks;
  /* code from Eric Young */
  D_ENCRYPT(l,r,(iround-1)*4);
  D_ENCRYPT(r,l,(iround-1)*4);
  /* end code from Eric Young */
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=l;
      l>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=r;
      r>>=8;
    }
  m1=tm;
  return(m1);
}

unsigned char *ddes2(unsigned char *m1) /* decryption step */
{
  unsigned char *tm;
  register unsigned long *s;
  int i;
  register unsigned long l,r,t,u;

  tm=m1;
  l=BYTE_TO_UL(m1);
  r=BYTE_TO_UL(m1+4);
  key_sched((C_Block *)(key),ks);
  s=(unsigned long *) ks;
  D_ENCRYPT(r,l,(iround-1)*4);
  D_ENCRYPT(l,r,(iround-1)*4);
  m1=tm;
  for(i=0;i<=3;i++)
    {
      *(m1++)=l;
      l>>=8;
    }
  for(i=0;i<=3;i++)
    {
      *(m1++)=r;
      r>>=8;
    }
  m1=tm;
  return(m1);
}
/* -------------------------- */

void generate_round_keys(cipher_length)  /* key schedule */
const int cipher_length;
{

 int i,j;
 unsigned char kss[12];
 unsigned char k2[8];

 key_sched((C_Block *)(key),ks); /* generate DES round keys */
 for(i=1;i<=cipher_length;i++)   /* generate other round keys */
   {
     key[8]='\0';
     kss[0]='\0';
     strcat(kss,key);
     kss[8]=i;
     kss[9]='\0';
     hashkey= RMD((byte *)kss);
     for(j=0;j<8;j++)
       key_schedule[i][j]=hashkey[j];
   }
}

void cag(fi, cfi, length)

unsigned int fi[MAX_HEIGHT][MAX_LAYERS];
unsigned int cfi[MAX_LAYERS];
int *length;

{
  int i,j,k;
  int nf,nl;                    /* for the random */
  unsigned short seed[3];       /* function       */
  unsigned short lseed[3];
  struct timeval tp;
  struct timezone tzp;
  /* Determine the size and indexes for the crypto algorithm */

/*  printf("seed:");
  i = getchar();
  getchar();*/
  for(i=0;i<4;i++)
    do
      {
gettimeofday(&tp,&tzp);
seed[i]=(unsigned char)(tp.tv_usec & 0x00FF);
gettimeofday(&tp,&tzp);
      }while(tp.tv_usec & 0x0081);
  seed48(seed);
  *length=nl=MIN_LAYERS+((int) (MAX_LAYERS-MIN_LAYERS)*drand48());
  for (i=0;i<=MAX_LAYERS;i++)
    {
      cfi[i]=0;
      for (j=0;j<=MAX_HEIGHT;j++)
fi[j][i]=0;
    }
  for (i=1;i<=nl;i++)
    {
      cfi[i]= (int) ((NUM_CFUNCT)*drand48())+1;
      nf=(MIN_HEIGHT)+((int)(MAX_HEIGHT-MIN_HEIGHT)*drand48());
      for (j=1;j<=nf;j++)
do
  {
    int l;
    k=0;
    fi[j][i]= (int) ((NUM_FUNCT)*drand48())+1;
    for(l=1;l<j;l++)
      if(fi[j][i] == fi[l][i] )
k=1;
  }
  while(k);
    }
}

void copyarr(unsigned char *a1,unsigned char *a2,const unsigned int nbl)
{                               /* copy array a2 to array a1 */
  int i;
  for(i=0;i<=nbl;i++)
    *(a2++)=*(a1++);
}


void encrypt(fi, cfi, mst, length, message_length)  /* encryption cipher */

const unsigned int fi[MAX_HEIGHT][MAX_LAYERS];
const unsigned int cfi[MAX_LAYERS];
unsigned char mst[];
const int length;
int *message_length;

{
  int i=0;
  int l,d;
  register int k,j;
  unsigned char block[12*BUFSIZ/MSG][MSG];
  unsigned char opblock[MSG],sum[length][MSG];
  unsigned int num_blocks;

 generate_round_keys(length);
  for(k=0;k<=BUFSIZ/MSG;k++)
    for(l=0;l<MSG;l++)
      block[k][l]=0;
  if(*message_length==0)
    {
      while(mst[i] !=0)
block[i/MSG][i%MSG]=mst[i++];
      *message_length=i;
      num_blocks=((i%MSG==0) ? i/MSG : i/MSG+1);
    }
  else
    {
      for(i=0;i<*message_length;i++)
block[i/MSG][i%MSG]=mst[i];
      num_blocks=((*message_length%MSG==0) ? *message_length/MSG : *message_length/MSG+1);
    }
  for(i=0;i<=length;i++)
    for(k=0;k<=MSG;k++)
      sum[i][k]=0;
  for(l=0;l<num_blocks;l++)
    {
      for(iround=1;iround<=length;iround++)
 {
  for(k=0;k<MSG;k++)
    block[l][k]^=sum[iround][k];
  (*nefunct[cfi[iround]])(block[l]);
  for(k=0;k<=MSG;k++)
    sum[iround][k]=0;
  for(j=1;j<=MAX_HEIGHT;j++)
    if (fi[j][iround]!=0)
      {
 for(k=0;k<MSG;k++)
```

```
  opblock[k]=block[l][k];
(*nfunct[fi[j][iround]])(opblock);
for(k=0;k<MSG;k++)
  sum[iround][k]^=opblock[k];
      }
    else
      break;
}
    }

  copyarr(block,mst,num_blocks*MSG);
}

void decrypt(fi, cfi, mst, length, message_length) /* decryption */

const unsigned int fi[MAX_HEIGHT][MAX_LAYERS];
const unsigned int cfi[MAX_LAYERS];
unsigned char mst[];
const int length;
const int message_length;

{
  int i=0;
  int k,j,l,d;
  unsigned char block[12*BUFSIZ/MSG][MSG];
  unsigned char opblock[MSG],sum[length][MSG],tsum[MSG];
  unsigned int num_blocks;

  generate_round_keys(length);
  for(i=0;i<=message_length-1;i++)
    block[i/MSG][i%MSG]=mst[i];
  i=message_length;
  num_blocks=((i%MSG==0) ? i/MSG : i/MSG+1);
  for(i=0;i<=length;i++)
    for(k=0;k<=MSG;k++)
      sum[i][k]=0;
  for(l=0;l<num_blocks;l++)
    {
      for(iround=length;iround>0;iround--)
{
  for(d=0;d<MSG;d++)
    tsum[d]=sum[iround][d];
  for(k=0;k<=MSG;k++)
     sum[iround][k]=0;
  for(j=1;j<=MAX_HEIGHT;j++)
    if (fi[j][iround]!=0)
      {
for(k=0;k<MSG;k++)
  opblock[k]=block[l][k];
(*nfunct[fi[j][iround]])(opblock);
for(k=0;k<MSG;k++)
  sum[iround][k]^=opblock[k];
      }
    else
      break;
   (*ndfunct[cfi[iround]])(block[l]);
   for(k=0;k<MSG;k++)
     block[l][k]^=tsum[k];
}
    }
  copyarr(block,mst,num_blocks*MSG);
}


void initcag()  /* initialize arrays to point to crypto functions */
{
  hashkey=malloc(128*sizeof(char));

  nfunct[1]  = crl;
  nfunct[2]  = crr;
  nfunct[3]  = hash1;
  nfunct[4]  = modmul;
  nfunct[5]  = modadd;
  nfunct[6]  = permute1;
  nfunct[7]  = permute2;
  nfunct[8]  = ipermute1;
  nfunct[9]  = vigenere;
  nfunct[10]= ivigenere;
  nfunct[11]= ipermute2;
  nfunct[12]= des1;
  nfunct[13]= ddes1;
  nfunct[14]= elr1;
  nfunct[15]= dlr1;
  nfunct[16]= elr2;
  nfunct[17]= dlr2;
  nfunct[18]= lfsr1;
  nfunct[19]= lfsr2;
  nfunct[20]= eblow_l;
  nfunct[21]= eblow_r;

  nefunct[1]  = permute1;
  ndfunct[1]  = ipermute1;
  nefunct[2]  = vigenere;
  ndfunct[2]  = ivigenere;
  nefunct[3]  = permute2;
  ndfunct[3]  = ipermute2;
  nefunct[4]  = des1;
  ndfunct[4]  = ddes1;
  nefunct[5]  = elr1;
  ndfunct[5]  = dlr1;
  nefunct[6]  = elr2;
  ndfunct[6]  = dlr2;
  nefunct[7]  = lfsr1;
  ndfunct[7]  = lfsr1;
  nefunct[8]  = lfsr2;
  ndfunct[8]  = lfsr2;
  nefunct[9]= eblow_l;
  ndfunct[9]= dblow_l;
  nefunct[10]= eblow_r;
  ndfunct[10]= dblow_r;
  nefunct[11]= des_ip;
  ndfunct[11]= des_ip_1;
  nefunct[12]= des_ip_1;
  ndfunct[12]= des_ip;
  nefunct[13]= des2;
  ndfunct[13]= ddes2;
  nefunct[14]= eufn;
  ndfunct[14]= dufn;
  nefunct[15]= eufn40_24;
  ndfunct[15]= dufn40_24;

/* Name of the functions for the use in the test suite */
/* NOTE: the namef strings must be 15 characters long, */
/*       whereas the namec 13.                         */
  namef[1]  = "rotate left    ";
  namef[2]  = "rotate right   ";
  namef[3]  = "hash #1        ";
  namef[4]  = "times mod 2^32 ";
  namef[5]  = "plus  mod 2^32 ";
  namef[6]  = "permute #1     ";
  namef[7]  = "permute #2     ";
  namef[8]  = "inv. permute #1";
  namef[9]  = "vigenere       ";
  namef[10]= "inv. vigenere  ";
  namef[11]= "inv. permute2  ";
  namef[12]= "1-round DES    ";
  namef[13]= "inv.1-round DES";
  namef[14]= "Feistel #1     ";
  namef[15]= "inv. Feistel #1";
  namef[16]= "Feistel #2     ";
  namef[17]= "inv. Feistel #2";
  namef[18]= "LFSR #1        ";
  namef[19]= "LFSR #2        ";
  namef[20]= "Blowfish L     ";
  namef[21]= "Blowfish R     ";

  namec[1]  = "permute #1   ";
  namec[2]  = "vigenere     ";
  namec[3]  = "permute #2   ";
  namec[4]  = "1-round DES  ";
  namec[5]  = "Feistel #1   ";
  namec[6]  = "Feistel #2   ";
  namec[7]  = "LFSR #1      ";
  namec[8]  = "LFSR #2      ";
  namec[9]= "Blowfish L   ";
  namec[10]= "Blowfish R   ";
  namec[11]= "Des.IP       ";
  namec[12]= "Des.IP^-1    ";
  namec[13]= "2-round DES  ";
  namec[14]= "UFN 16:48    ";
  namec[15]= "UFN 40:24    ";
}
```

D.4 cagscr2.c

```
/*
  cagscr2.c, Vasilios Katos, 1998.
  This module implements the UNIX talk-like utility, without the talk daemon.
  A server runs instead waiting for connection request from a client.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include "fstuff.h"
#include "rmd128.h"
#include "LargeOp.h"
#include "PrimeTools.h"
#include "rsa.h"
#include <memory.h>
#include <math.h>
#include <curses.h>

extern void encrypt();
extern void decrypt();
extern void cag();
extern void copyarr();
extern void initcag();
extern void init_display();
extern void display(WINDOW *,const int);
extern void display_options();
extern void status(const int);
extern void tserver();
extern void tclient();

unsigned char *(*nfunct[NUM_FUNCT+1])();   /* array of pointers to feedback functions */
unsigned char *(*nefunct[NUM_CFUNCT+1])(); /* array of pointers to encryption functions */
unsigned char *(*ndfunct[NUM_CFUNCT+1])(); /* array of pointers to decryption functions */

extern WINDOW *win_a,*win_b;
extern char *num[];
extern char *myname;

main()
{
  unsigned int findex[MAX_HEIGHT][MAX_LAYERS];  /* feedback blocks */
  unsigned int cfindex[MAX_LAYERS];             /* encryption blocks */
  int i,j,dc;
  unsigned char c1[BUFSIZ-1],c[BUFSIZ-1],*pc,*tpc,*enc,*tenc;
  unsigned char li[BUFSIZ-1], lj[MSG-1];
  unsigned char outbuf[BUFSIZ-1],dca[BUFSIZ-1];
  int pos;
  byte *hashcode;
  LARGE_INT pp,pq,e,d,n;          /* for the RSA */
  FILE *fp;
  char *remhost;                  /* hostname of client tcp request */
  char *callname;                 /* name of user running the server */
  int len,mlen,nblocks;

  initcag();                      /* initialize cryptographic parameters */
  InitLarge();                    /* initialize large number arrays */
  init_display();                 /* initialize curses */
  display_options();              /* Options menu */
  while((dc=getch())!='5')
    switch(dc)
      {
      case '1':                   /* generate/refresh public/private key pair */
status(1);
waddstr(win_a,"\n  Computing p...");
wrefresh(win_a);
LargePrimeHunt(PSIZE_MIN,PSIZE_MAX,&pp);
waddstr(win_a,"\n  Computing q...");
wrefresh(win_a);
LargePrimeHunt(QSIZE_MIN,QSIZE_MAX,&pq);
waddstr(win_a,"\n  Computing public key...");
wrefresh(win_a);
LargePrimeHunt(ESIZE_MIN,ESIZE_MAX,&e);
Mul(&pp,&pq,&n);
waddstr(win_a,"\n  Computing secret key...");
wrefresh(win_a);
GenerateRSAPair(&pp,&pq,&e,&d);
waddstr(win_a,"\n  Pair generated.\n");
wrefresh(win_a);
fp=fopen("RSA","w");                        /* write keys to file RSA */
i=fwrite(&pp.Size,sizeof(int),1,fp);
i=fwrite(pp.Digits,sizeof(BYTE),pp.Size,fp);
i=fwrite(&pq.Size,sizeof(int),1,fp);
i=fwrite(pq.Digits,sizeof(BYTE),pq.Size,fp);
i=fwrite(&e.Size,sizeof(int),1,fp);
i=fwrite(e.Digits,sizeof(BYTE),e.Size,fp);
i=fwrite(&d.Size,sizeof(int),1,fp);
i=fwrite(d.Digits,sizeof(BYTE),d.Size,fp);
fclose(fp);
status(0);
break;
      case '2':                             /* construct a symmetric block cipher */
status(4);
cag(findex,cfindex,&len);
waddstr(win_a,"Layers selected:");      /* display cipher ...*/
waddstr(win_a,num[len]);
waddstr(win_a,"\n");
for(i=1;i<=len;i++)
  {
    waddstr(win_a,num[cfindex[i]]);
    waddstr(win_a,":");
    for(j=1;j<MAX_HEIGHT;j++)
      if(findex[j][i] != 0 )
{
  waddstr(win_a,num[findex[j][i]]);
  waddstr(win_a,"-");
}
    waddstr(win_a,"\n");
  }
wrefresh(win_a);
status(0);
break;
       case '3':                            /* call client */
remhost=malloc(32);
wprintw(win_b,"\n\nCall user:");
wrefresh(win_b);
remhost=malloc(32);
callname=malloc(16);
i=j=0;
do
  {
    if((j=getch())!=8)
      {
callname[i]=j;
if(callname[i]!='\n')
  display(win_b,callname[i]);
else
  display(win_b,'@');
i++;
      }
    else
      if(i>0)
{
  display(win_b,8);
  display(win_b,' ');
  display(win_b,8);
  i--;
}
  }while(j!='@' && j!='\n');
callname[i-1]='\0';
i=0;
do
  {
    if((j=getch())!=8)
      {
remhost[i]=j;
display(win_b,remhost[i]);
i++;
      }
    else
      if(i>0)
{
  display(win_b,8);
  display(win_b,' ');
  display(win_b,8);
  i--;
}
  }while(j!='\n');
remhost[i-1]='\0';
tclient(remhost,callname,findex,cfindex,len);
display_options();
break;
      case '4':                     /* call server */
status(10);
tserver();
status(0);
display_options();
break;
      }
  endwin();
  printf("\n\n");
}
```

D.5 rsa.h

```
/* sizes of primes p,q and public exponent e*/
#define PSIZE_MIN 5
#define PSIZE_MAX 6
#define QSIZE_MIN 6
#define QSIZE_MAX 7
#define ESIZE_MIN 7
#define ESIZE_MAX 7

extern void GenerateRSAPair(LARGE_INT *, LARGE_INT *, LARGE_INT *, LARGE_INT *);
extern void RSA(LARGE_INT *,const LARGE_INT *, const LARGE_INT *);
```

D.6 rsa.c

```
/*
  rsa.c, Vasilios Katos, 1998.
  Extensions on Cooke's (1995) functions to perform RSA operations. */
#include "LargeOp.h"
#include "PrimeTools.h"
#include "rsa.h"
#include <memory.h>
#include <stdio.h>

static void Inverse(const LARGE_INT *, const LARGE_INT *, LARGE_INT *);
static void gcd(const LARGE_INT *, const LARGE_INT *, LARGE_INT *);

static void Inverse(const LARGE_INT *Val, const LARGE_INT *Mod, LARGE_INT *Inv)
{
  LARGE_INT c1,c2,c3,t,b1,b2,b3,zero;
  int sign=0;

  Reset(Inv);
  LargeCopy(Mod,&c1);
  LargeCopy(Val,&c2);
  Reset(&b1);
  LargeCopy(&One,&b2);
  Reset(&zero);
  do
    {
      Div(&c1,&c2,&t,&c3);
      Mul(&t,&b2,&b3);
      Add(&b1,&b3,&b3);
      LargeCopy(&c2,&c1);
      LargeCopy(&c3,&c2);
      LargeCopy(&b2,&b1);
      LargeCopy(&b3,&b2);
      sign++;
    }while(Comp(&c2,&zero));
  if(sign%2==0)
    Sub(Mod,&b1,&b1);
  LargeCopy(&b1,Inv);
}

static void gcd(const LARGE_INT *op1, const LARGE_INT *op2, LARGE_INT *opt)
{
  LARGE_INT m,a,b,c,zero;
  Reset(&zero);
  if(Comp(op1,op2)==1)
    {
      LargeCopy(op1,&a);
      LargeCopy(op2,&b);
    }
  else
    {
      LargeCopy(op2,&a);
      LargeCopy(op1,&b);
    }
  do
    {
      Div(&a,&b,&m,&c);
      LargeCopy(&b,&a);
      LargeCopy(&c,&b);
    }while(Comp(&c,&zero)!=0);
  LargeCopy(&a,opt);
}


void GenerateRSAPair(LARGE_INT *p,LARGE_INT *q,LARGE_INT *e,LARGE_INT *d)                   /* Input  : primes p,q and public e          */
             /* Output: primes p,q public e, secret d    */
{
  LARGE_INT n,phi_n,p1,q1,gcdpq1,bin,testmsg,tmp;
  int i,foundpair=0;
  Reset(d);
  do
    {
      Mul(p,q,&n);
      Sub(p,&One,&p1);
      Sub(q,&One,&q1);
      Mul(&p1,&q1,&phi_n);
      gcd(&q1,&p1,&gcdpq1);
      Div(&phi_n,&gcdpq1,&phi_n,&bin);
      Inverse(e,&phi_n,d);
      do
{
  Add(e,&One,e);
  Inverse(e,&phi_n,d);
#ifndef Quiet
  printf("Updating e...\n");
#endif
}while(IsGCD1(e,&p1)!=1 || IsGCD1(e,&q1)!=1 || IsGCD1(d,&p1)!=1 || IsGCD1(d,&q1)!=1);
      for(i=0;i<5;i++)
{
  GenerateRandom(&testmsg,n.Size-1,n.Size-1);
  LargeCopy(&testmsg,&bin);
#ifndef Quiet
  Display(&testmsg);
#endif
  RSA(&testmsg,e,&n);
#ifndef Quiet
  Display(&testmsg);
#endif
  RSA(&testmsg,d,&n);
#ifndef Quiet
  Display(&testmsg);
  printf("--------------\n");
#endif
  if(Comp(&testmsg,&bin)==0)
    foundpair++;
  else
    break;
}
      if(foundpair!=5)
{
  foundpair=0;
  LargePrimeHunt(p->Size,p->Size,p);
  LargePrimeHunt(q->Size,q->Size,q);
}
    }while(foundpair==0);
}
```

```
void RSA(LARGE_INT *message, const LARGE_INT *key, const LARGE_INT *mod)        /* (message)=(message)^(key) MOD(mod) */
{
  LARGE_INT res;
  PowerMod(message,key,mod,&res);
  LargeCopy(&res,message);
}
```

D.7 server.c

```
/*
  server.c, Vasilios Katos, 1998.
  The tserver() function call.
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
#include <stdio.h>
#include <ctype.h>
#include <curses.h>
#include <memory.h>
#include <math.h>
#include "LargeOp.h"
#include "PrimeTools.h"
#include "rsa.h"
#include "fstuff.h"
#include <sys/stat.h>
#define MYPORT 5000

extern errno;
extern void display(WINDOW *,const int);
extern void encrypt();
extern void decrypt();
extern WINDOW *win_a,*win_b,*win_c;
extern char *num[];
extern void tsrver();
extern char *myname;
extern char key[];

void tserver()
{
  unsigned int fin[MAX_HEIGHT][MAX_LAYERS];
  unsigned int cfin[MAX_LAYERS];
  int sock;  /* TCP socket */
  int msgsock; /* TCP connection */
  int length; /* length */
  struct sockaddr_in server; /* socket address */
  unsigned char inbuf[BUFSIZ],outbuf[BUFSIZ]; /* data buffer */
  int rval, running = TRUE;
  int on = 1; /* reuse socket address */
  int pid,dc[BUFSIZ],i,pos=0;
  unsigned char hisname[16];
  int len,mlen,nblocks;
  int pipefd;
  /* RSA variables */
  LARGE_INT pp,pq,e,d,n,m,em;
  LARGE_INT li,lj;
  int dg,j,nomore;
  FILE *fp;

  for(i=0;i<9;i++)
    key[i]='a'+i;
  sock = socket( AF_INET, SOCK_STREAM, 0); /* create a socket */
  setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,(char *)&on, sizeof(on));
  if (sock == -1) {
    ("opening stream socket");
    exit(1);
  }
  /* bind socket */
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = htonl(INADDR_ANY);
  server.sin_port = htons(MYPORT);
  if (bind(sock, (struct sockaddr *) &server, sizeof(server)) == -1) {
    waddstr(win_a,"\nerror: binding stream socket");
    wrefresh(win_a);
    exit(1);
  }
  /* find out assigned port number and print it out */
  length = sizeof(server);
  if (getsockname(sock, (struct sockaddr *) &server, &length) == -1) {
    waddstr(win_a,"\nerror: getting socket name");
    wrefresh(win_a);
    exit(1);
  }
  /* start accepting connections */
  listen(sock, 2);
  do {
    wmove(win_c,0,0);
    wrefresh(win_c);
    leaveok(win_a,TRUE);
    leaveok(win_b,TRUE);
    immedok(win_a,TRUE);
    immedok(win_b,TRUE);
 nl();
    wclear(win_b);
    wrefresh(win_b);
    msgsock = accept(sock, (struct sockaddr *) 0, (int *) 0);
    if (msgsock == -1) {
      waddstr(win_a,"\nerror: accept");
      wrefresh(win_a);
    }
    else
      {
/* Get RSA parameters */
InitLarge();
Reset(&pp);
Reset(&pq);
Reset(&e);
Reset(&d);
fp=fopen("RSA","r");
i=fread(&pp.Size,sizeof(int),1,fp);
wprintw(win_a,"\nsize %d",pp.Size);
i=fread(pp.Digits,sizeof(BYTE),pp.Size,fp);
wprintw(win_a,"\nbytes:%d",i);
i=fread(&pq.Size,sizeof(int),1,fp);
wprintw(win_a,"\nsize %d",pq.Size);
i=fread(pq.Digits,sizeof(BYTE),pq.Size,fp);
wprintw(win_a, "\nbytes:%d",i);
i=fread(&e.Size,sizeof(int),1,fp);
i=fread(e.Digits,sizeof(BYTE),e.Size,fp);
wprintw(win_a,"\nbytes:%d",i);
i=fread(&d.Size,sizeof(int),1,fp);
i=fread(d.Digits,sizeof(BYTE),d.Size,fp);
wprintw(win_a,"\nbytes:%d",i);
fclose(fp);
Mul(&pp,&pq,&n);
/*End Get RSA parameters */
/* get myname, hisname, his public key */
status(3);
memset(hisname,'\0',sizeof(hisname));
if ((rval = read(msgsock, hisname, sizeof(hisname))) == -1)
  {
    waddstr(win_a,"\nerror: reading socket stream");
  }
wprintw(win_a,"\nhisname transfered:%s",hisname);
memset(outbuf,'\0',sizeof(outbuf));
outbuf[0]=1;
if (write(msgsock, outbuf, 1) < 0)
  {
    waddstr(win_a,"\n error: write to socket");
    return;
  }
memset(inbuf,'\0',sizeof(inbuf));
if ((rval = read(msgsock, inbuf, sizeof(inbuf))) == -1) {
    waddstr(win_a,"\nerror: reading socket stream");
}
wprintw(win_a,"\nReceived myname:%s",inbuf);
if(!strncmp(inbuf,myname,strlen(myname)))
  {
    waddstr(win_a,"\nUser confirmed.\n");
  }
else
  {
    outbuf[0]=2; /* Send client "no user" */
    if (write(msgsock, outbuf, 1) < 0)
      {
waddstr(win_a,"\n error: write to socket");
return;
      }
    ungetch('4');
    close(msgsock);
    close(sock);
    return;
  }
status(9);
wprintw(win_c,"%s,accept ?",hisname);
wrefresh(win_c);
do
  {
    i=getchar();
  }while(i!='y' && i!='Y' && i!='n' && i!='N');
if (i=='n' || i=='N')
  {
    outbuf[0]=3; /* Send client "reject" */
    if (write(msgsock, outbuf, 1) < 0)
      {
waddstr(win_a,"\n error: write to socket");
return;
      }
    ungetch('4');
    close(msgsock);
    close(sock);
    return;
  }
if (write(msgsock, outbuf, 1) < 0)
```

```
  {
    waddstr(win_a,"\n error: write to socket");
    return;
  }
memset(inbuf,'\0',sizeof(inbuf));
if ((rval = read(msgsock, inbuf, sizeof(inbuf))) == -1) {
    waddstr(win_a,"\nerror: reading socket stream");
}
Reset(&m);
m.Size=rval;
wprintw(win_a,"\nReceived modulus:");
for(i=0;i<m.Size;i++)
  m.Digits[i]=inbuf[i];
/*DISPLAY*/
    for(j=m.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",m.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
if (write(msgsock, outbuf, 1) < 0)
  {
    waddstr(win_a,"\n error: write to socket");
    return;
  }
if ((rval = read(msgsock, inbuf, sizeof(inbuf))) == -1) {
    waddstr(win_a,"\nerror: reading socket stream");
}
Reset(&em);
em.Size=rval;
wprintw(win_a,"Received public:");
for(i=0;i<em.Size;i++)
  em.Digits[i]=inbuf[i];
/*DISPLAY*/
    for(j=em.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",em.Digits[j]);
    wprintw(win_a,"\n");
/*ENDDISPLAY*/
 /* send myname, my public key             */
    status(2);
memset(outbuf,'\0',sizeof(outbuf));
strcat(outbuf,myname);
if (write(msgsock, outbuf, strlen(outbuf)) < 0)
  {
    waddstr(win_a,"\n error: write to socket");
    return;
  }
wprintw(win_a,"\nsending myname:%s!",outbuf);
if ((rval = read(msgsock, inbuf, sizeof(inbuf))) == -1) {
  waddstr(win_a,"\nerror: reading socket stream");
}
wprintw(win_a,"\nSending public modulus:");
memset(outbuf,'\0',sizeof(outbuf));
for(i=0;i<n.Size;i++)
  outbuf[i]=n.Digits[i];
/*DISPLAY*/
    for(j=n.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",n.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
if (write(msgsock, outbuf, n.Size) < 0)
  {
    waddstr(win_a,"\n error: write to socket");
    return;
  }
if ((rval = read(msgsock, inbuf, sizeof(inbuf))) == -1) {
  waddstr(win_a,"\nerror: reading socket stream");
}
wprintw(win_a,"Sending public key:");
memset(outbuf,'\0',sizeof(outbuf));
for(i=0;i<e.Size;i++)
        outbuf[i]=e.Digits[i];
/*DISPLAY*/
    for(j=e.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",e.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
if (write(msgsock, outbuf, e.Size) < 0)
  {
    waddstr(win_a,"\n error: write to socket");
    return;
  }
/* Receive symmetric algorithm */
status(6);
len=1;
nomore=FALSE;
while(!nomore)
  {
    if((rval = read(msgsock, inbuf, sizeof(inbuf)))<0)
      {
waddstr(win_a,"\n error: write to socket");
return;
      }
    if(rval==1 && inbuf[0]==0xFF)
      break;
    wprintw(win_a,"Receiving layer:%d\n",len);
    if (write(msgsock,outbuf,1) < 0)
      {
```

```
waddstr(win_a,"\n error: write to socket");
return;
      }
    Reset(&lj);
    lj.Size=rval;
    wprintw(win_a,"\nReceived layer:");
    for(i=0;i<lj.Size;i++)
      lj.Digits[i]=inbuf[i];
/*DISPLAY*/
    for(j=lj.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",lj.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
    PowerMod(&lj,&d,&n,&li);
/*DISPLAY*/
    for(j=li.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",li.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
/*     RSA(&li,&d,&n);*/
    cfin[len]=(unsigned int)li.Digits[0];
    for(i=1;i<MAX_HEIGHT;i++)
      fin[i][len]=(unsigned int) li.Digits[li.Size-i];
    len++;
  }
len--;
wprintw(win_a,"Layers selected:%d\n",len);
for(i=1;i<=len;i++)
  {
    waddstr(win_a,num[cfin[i]]);
    waddstr(win_a,":");
    for(j=1;j<MAX_HEIGHT;j++)
      if(fin[j][i] != 0 )
{
  waddstr(win_a,num[fin[j][i]]);
  waddstr(win_a,"-");
}
    waddstr(win_a,"\n");
  }
/* evaluate the block cipher and re-sent */
        status(5);
evaluate(fin,cfin,&len)
wprintw(win_a,"\nSending no of layers:%d",len);
for(i=1;i<=len;i++)
  {
    Reset(&lj);
    lj.Digits[0]=(unsigned char)cfin[i];
    wprintw(win_a,"\n%i:",(int)lj.Digits[MAX_HEIGHT]);
    for(j=MAX_HEIGHT-1;j>0;j--)
      lj.Digits[j]=(unsigned char)fin[MAX_HEIGHT-j][i];
    lj.Size=MAX_HEIGHT;
/*DISPLAY*/
    for(j=lj.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",lj.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
    wprintw(win_a,"\nRSA:");
    RSA(&li,&em,&m);
/*DISPLAY*/
    for(j=li.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",li.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
    wprintw(win_a,"\nSending layer:%d\n",i);
    memset(outbuf,'\0',sizeof(outbuf));
    for(j=0;j<li.Size;j++)
      outbuf[j]=li.Digits[j];
    if (write(msgsock, outbuf, li.Size) < 0)
      {
waddstr(win_a,"\n error: write to socket");
return;
      }
    if ((rval = read(msgsock, inbuf, sizeof(inbuf))) == -1) {
      waddstr(win_a,"\nerror: reading socket stream");
    }
  }
outbuf[0]=END_OF_CIPHER;      /* send terminator */
if (write(msgsock, outbuf, 1) < 0)
  {
    waddstr(win_a,"\n error: write to socket");
    return;
  }
/*------------------*/
status(0);
/* Create a named pipe for Eve */
/*if(mknod("eve",S_IFIFO | 0666,0)<0)
  wprintw(win_a,"Error: Cannot create pipe\n");
else
  if((pipefd=open("eve",1))<0)
    wprintw(win_a,"Error: Cannot open pipe\n");
    */
status(13);
wprintw(win_c,"%s",hisname);
wrefresh(win_c);
if((pid=fork())>=0)
  do {
    if(pid==0)
      {
memset(inbuf, '\0', sizeof(inbuf));
if ((rval = read(msgsock, inbuf, BUFSIZ)) == -1) {
  waddstr(win_a,"\nerror: reading socket stream");
}
if (rval == 0)
  {
    waddstr(win_a,"Ending connection\n");
    close(msgsock,0);
    close(sock,0);
    i=getch();
    return;
  }
else {
  if (!strncmp(inbuf,"quit",4)) {
    exit(0);
  }
  else {
    mlen=rval-8;
    if(mlen==-8)
      {
close(msgsock,0);
close(sock,0);
return;
      }
    if(mlen)
      {
/* Eve's lines */
/* write(pipefd,inbuf,mlen);*/
/*--------------*/
decrypt(fin,cfin,inbuf,len,mlen);
for(i=8;i<mlen;i++)
  wprintw(win_a,"%c",inbuf[i]);
      }
    wprintw(win_a,"\n");
  }
}
      }
    else
      {
memset(outbuf,0,sizeof(outbuf));
pos=0;
strcat(outbuf,key);
pos=8;
do
  {
    outbuf[pos]=getch();
    display(win_b,outbuf[pos]);
  }while(outbuf[pos++]!='\n');
/* for(i=pos-1;i>=0;i--)
  ungetch(dc[i]);
wgetstr(win_b,outbuf);*/
if((pos-1)%MSG==0)
  outbuf[pos++]=' ';
for(i=pos-1;i<MSG*(((pos-1)%MSG==0) ? (pos-1)/MSG : (pos-1)/MSG+1);i++)
  outbuf[i]=' ';
mlen=0;
encrypt(fin,cfin,outbuf,len,&mlen);
if (write(msgsock, outbuf, mlen+8) < 0) {
  waddstr(win_a,"writing socket stream");
  close(msgsock,0);
  close(sock);
  exit(1);
}
      }
  } while (running == TRUE && rval != 0);
      }
    close(msgsock);
  } while (running == TRUE);
  close(sock,0);
  exit(0);
}
```

D.8 client.c

```
/*
  client.c, Vasilios Katos, 1998.
  The function call to tclient().
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <ctype.h>
#include <string.h>
#include <curses.h>
#include <memory.h>
#include "LargeOp.h"
#include "PrimeTools.h"
```

```
#include "rsa.h"
#include "fstuff.h"

extern int h_error;
#define MYPORT 5000
extern void encrypt();
extern void decrypt();
extern void tclient();
extern WINDOW *win_a,*win_b,*win_c;
extern char *num[];
extern char *myname;
char key[]="abcdefgh";

void tclient(char *remuser,char *hisname, unsigned int fin[MAX_HEIGHT][MAX_LAYERS], unsigned int cfin[MAX_LAYERS], int len)
{
  struct sockaddr_in sin; /* socket address */
  struct servent *ps;   /* server entry */
  struct hostent *ph;   /* host entry */
  int s;  /* TCP socket */
  int rlen; /* length */
  int quitting = FALSE; /* flag to tell when to quit */
  long address; /* IP address */
  unsigned char inbuf[BUFSIZ],outbuf[BUFSIZ]; /* data buffer */
  char *host; /* remote host */
  int on = 1; /* reuse socket address */
  int nomore;                   /* flag for block cipher exchange*/
  int pid,dc[BUFSIZ],i,pos=0;
  int mlen,nblocks;

/* RSA variables */
  LARGE_INT pp,pq,e,d,n,m,em;
  LARGE_INT li,lj;
  int dg,j;
  FILE *fp;

/*
 * Find Internet address of host
 */
  host=remuser;
  waddstr(win_a,hisname);
  waddstr(win_a,"\n host:");
  waddstr(win_a,host);
  wrefresh(win_a);
      if (isdigit(host[0])) { /* dotted IP Address */
    if ((address = inet_addr(host)) == -1) {
      waddstr(win_a,"\n invalid host name");
      wrefresh(win_a);
      return;
    }
    sin.sin_addr.s_addr = address;
    sin.sin_family = AF_INET;
  }
  else
    if ((ph = gethostbyname(host)) == NULL) {
      switch (h_errno) {
      case HOST_NOT_FOUND:
waddstr(win_a,"\nNo such host ");
wrefresh(win_a);
return;
      case TRY_AGAIN:
waddstr(win_a, "\nhost busy, try again later");
wrefresh(win_a);
return;
      case NO_RECOVERY:
waddstr(win_a, "\nDNS error");
wrefresh(win_a);
return;
      case NO_ADDRESS:
waddstr(win_a, "\nno IP address for required host");
wrefresh(win_a);
return;
      default:
waddstr(win_a, "\nclient: uknown error");
wrefresh(win_a);
return;
      }
    }
    else {
      sin.sin_family = ph->h_addrtype;
      memcpy((char *) & sin.sin_addr, ph->h_addr, ph->h_length);
    }
  sin.sin_port = htons(MYPORT);
  /* open a socket */
  if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    waddstr(win_a,"\n error: socket");
    wrefresh(win_a);
    return;
  }
  setsockopt(s,SOL_SOCKET,SO_REUSEADDR,(char *)&on, sizeof(on));
  /* connect to the remote echo server */
  if (connect(s, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
    waddstr(win_a,"\n error: connect");
    wrefresh(win_a);
    return;
  }
  /* until end of file, copy data from standard input to echo server
```

```
     and echo copy echo server's response to standard out */
  wmove(win_c,0,0);
  wrefresh(win_c);
  leaveok(win_a,TRUE);
  leaveok(win_b,TRUE);
  immedok(win_a,TRUE);
  immedok(win_b,TRUE);
  nl();
  wclear(win_b);
  wrefresh(win_b);
  /* Get RSA parameters */
  InitLarge();
  Reset(&pp);
  Reset(&pq);
  Reset(&e);
  Reset(&d);
  Reset(&n);

  fp=fopen("RSA","r");
  i=fread(&pp.Size,sizeof(int),1,fp);
  wprintw(win_a,"\nsize %d",pp.Size);
  i=fread(pp.Digits,sizeof(BYTE),pp.Size,fp);
  wprintw(win_a,"\nbytes:%d",i);
  i=fread(&pq.Size,sizeof(int),1,fp);
  wprintw(win_a,"\nsize %d",pq.Size);
  i=fread(pq.Digits,sizeof(BYTE),pq.Size,fp);
  wprintw(win_a, "\nbytes:%d",i);
  i=fread(&e.Size,sizeof(int),1,fp);
  i=fread(e.Digits,sizeof(BYTE),e.Size,fp);
  wprintw(win_a,"\nbytes:%d",i);
  i=fread(&d.Size,sizeof(int),1,fp);
  i=fread(d.Digits,sizeof(BYTE),d.Size,fp);
  wprintw(win_a,"\nbytes:%d",i);
  fclose(fp);
  Mul(&pp,&pq,&n);
  /*End Get RSA parameters */
  status(8);
  /* send myname, hisname, my public key */
  memset(outbuf,'\0',sizeof(outbuf));
  strcat(outbuf,myname);
  if (write(s, outbuf, strlen(outbuf)) < 0)
    {
      waddstr(win_a,"\n error: write to socket");
      return;
    }
  wprintw(win_a,"\nsending myname:%s!",outbuf);
  if ((rlen = read(s, inbuf, sizeof(inbuf))) == -1) {
    waddstr(win_a,"\nerror: reading socket stream");
  }
  memset(outbuf,'\0',sizeof(outbuf));
  strcat(outbuf,hisname);
  wprintw(win_a,"\nsending hisname:%s!\n",outbuf);
  if (write(s, outbuf, strlen(outbuf)) < 0)
    {
      waddstr(win_a,"\n error: write to socket");
      return;
    }
  if ((rlen = read(s, inbuf, sizeof(inbuf))) == -1) {
    waddstr(win_a,"\nerror: reading socket stream");
  }
  if (inbuf[0]==2 || inbuf[0]==3)
    {
      status(9+inbuf[0]);
      close(s);
      return;
    }
  status(2);
  memset(outbuf,'\0',sizeof(outbuf));
  for(i=0;i<n.Size;i++)
    outbuf[i]=n.Digits[i];

  wprintw(win_a,"\nSending modulus:");
/*DISPLAY*/
    for(j=n.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",n.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
  if (write(s, outbuf, n.Size) < 0)
    {
      waddstr(win_a,"\n error: write to socket");
      return;
    }
  if ((rlen = read(s, inbuf, sizeof(inbuf))) == -1) {
    waddstr(win_a,"\nerror: reading socket stream");
  }
  wprintw(win_a,"\nSending public key:");
/*DISPLAY*/
    for(j=e.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",e.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
  memset(outbuf,'\0',sizeof(outbuf));
  for(i=0;i<e.Size;i++)
    outbuf[i]=e.Digits[i];
  if (write(s, outbuf, e.Size) < 0)
    {
      waddstr(win_a,"\n error: write to socket");
      return;
    }
  /* get hisname, his public key */
  status(3);
    memset(inbuf,'\0',sizeof(inbuf));
    if ((rlen = read(s, inbuf, sizeof(inbuf))) == -1) {
      waddstr(win_a,"\nerror: reading socket stream");
    }
    wprintw(win_a,"\nReceived hisname:%s",inbuf);
    if(!strncmp(inbuf,hisname,strlen(myname)))
      {
waddstr(win_a,"\nUser confirmed.\n");
      }
    if (write(s, outbuf, 1) < 0)
      {
waddstr(win_a,"\n error: write to socket");
return;
      }
memset(inbuf,'\0',sizeof(inbuf));
if ((rlen = read(s, inbuf, sizeof(inbuf))) == -1) {
    waddstr(win_a,"\nerror: reading socket stream");
}
Reset(&m);
m.Size=rlen;
wprintw(win_a,"Received modulus:");

for(i=0;i<m.Size;i++)
  m.Digits[i]=inbuf[i];
/*DISPLAY*/
    for(j=m.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",m.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
if (write(s, outbuf, 1) < 0)
  {
    waddstr(win_a,"\n error: write to socket");
    return;
  }
memset(inbuf,'\0',sizeof(inbuf));
if ((rlen = read(s, inbuf, sizeof(inbuf))) == -1) {
  waddstr(win_a,"\nerror: reading socket stream");
}
Reset(&em);
em.Size=rlen;
wprintw(win_a,"Received public:");
for(i=0;i<em.Size;i++)
  em.Digits[i]=inbuf[i];
/*DISPLAY*/
    for(j=em.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",em.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
/* Encrypt symmetric algorithm with his public key and send */
  status(5);
wprintw(win_a,"\nSending no of layers:%d",len);
for(i=1;i<=len;i++)
  {
    Reset(&lj);
    lj.Digits[0]=(unsigned char)cfin[i];
    wprintw(win_a,"\n%i:",(int)lj.Digits[MAX_HEIGHT]);
    for(j=MAX_HEIGHT-1;j>0;j--)
      lj.Digits[j]=(unsigned char)fin[MAX_HEIGHT-j][i];
    lj.Size=MAX_HEIGHT;
/*DISPLAY*/
    for(j=lj.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",lj.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
    wprintw(win_a,"\nRSA:");
    RSA(&li,&em,&m);
/*DISPLAY*/
    for(j=li.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",li.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
    wprintw(win_a,"\nSending layer:%d\n",i);
    memset(outbuf,'\0',sizeof(outbuf));
    for(j=0;j<li.Size;j++)
      outbuf[j]=li.Digits[j];
    if (write(s, outbuf, li.Size) < 0)
      {
waddstr(win_a,"\n error: write to socket");
return;
      }
    if ((rlen = read(s, inbuf, sizeof(inbuf))) == -1) {
      waddstr(win_a,"\nerror: reading socket stream");
    }
  }
outbuf[0]=0xFF;
if (write(s, outbuf, 1) < 0)
  {
    waddstr(win_a,"\n error: write to socket");
    return;
  }
status(0);
```

```
/* recieve revised algorithm */
  status(6);
len=1;
nomore=FALSE;
while(!nomore)
  {
    if((rlen = read(s, inbuf, sizeof(inbuf)))<0)
      {
waddstr(win_a,"\n error: write to socket");
return;
      }
    if(rlen==1 && inbuf[0]==END_OF_CIPHER)
      break;
    wprintw(win_a,"Receiving layer:%d\n",len);
    if (write(s,outbuf,1) < 0)
      {
waddstr(win_a,"\n error: write to socket");
return;
      }

    Reset(&lj);
    lj.Size=rlen;
    wprintw(win_a,"\nReceived layer:");
    for(i=0;i<lj.Size;i++)
      lj.Digits[i]=inbuf[i];
/*DISPLAY*/
    for(j=lj.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",lj.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
    PowerMod(&lj,&d,&n,&li);
/*DISPLAY*/
    for(j=li.Size-1;j>=0;j--)
      wprintw(win_a,"%02X",li.Digits[j]);
  wprintw(win_a,"\n");
/*ENDDISPLAY*/
    cfin[len]=(unsigned int)li.Digits[0];
    for(i=1;i<MAX_HEIGHT;i++)
      fin[i][len]=(unsigned int) li.Digits[li.Size-i];
    len++;
  }
len--;
wprintw(win_a,"Layers selected:%d\n",len);
for(i=1;i<=len;i++)
  {
    waddstr(win_a,num[cfin[i]]);
    waddstr(win_a,":");
    for(j=1;j<MAX_HEIGHT;j++)
      if(fin[j][i] != 0 )
{
  waddstr(win_a,num[fin[j][i]]);
  waddstr(win_a,"-");
}
    waddstr(win_a,"\n");
  }
/*---------------------*/
status(0);
status(13);
wprintw(win_c,"%s",hisname);
wrefresh(win_c);
  pid=fork();            /* create parallel process */
  if(pid>=0)
     {
       while (!quitting) {
 if(pid>0)
   {
     memset(outbuf,'\0',sizeof(outbuf));
     pos=0;
     strcat(outbuf,key);
     pos=8;
     do
       {
 outbuf[pos]=getch();
 display(win_b,outbuf[pos]);
       }while(outbuf[pos++]!='\n');
     if((pos-1)%MSG==0)
       outbuf[pos++]=' ';
     for(i=pos-1;i<MSG*(((pos-1)%MSG==0) ? (pos-1)/MSG : (pos-1)/MSG+1);i++)
       outbuf[i]=' ';
     mlen=0;
     encrypt(fin,cfin,outbuf,len,&mlen);
     if (write(s, outbuf, mlen+8) < 0) {
       waddstr(win_a,"\n error: write to socket");
       wrefresh(win_a);
       return;
     }
     if (!strncmp(outbuf,"quit",4)) {
       quitting = TRUE;
     }
   }
 if(pid==0)
   {
     memset(inbuf,'\0',sizeof(inbuf));
     if ((mlen = read(s, inbuf, sizeof(inbuf))) < 0) {
       waddstr(win_a,"\n error: read");
       wrefresh(win_a);
      return;
    }
    else
      {
mlen-=8;
if(mlen==-8)
  {
    close(s);
    return;
  }
if(mlen)
  {
    decrypt(fin,cfin,inbuf,len,mlen);
    for(i=8;i<mlen;i++)
      wprintw(win_a,"%c",inbuf[i]);
    wrefresh(win_a);
  }
wprintw(win_a,"\n");
      }
  }
      }
    }
  close(s);
  return;
}
```

D.9 cbp.h

```
/*
 cbp.h, Vasilios Katos, 1998.
 The Cryptographic Block Profile. */

#define SEC_SPEC 1  /* level of crypto strength */
```

D.10 cbp.c

```
/*
  cbp.c, Vasilios Katos, 1998.
  Definition of CBP, values, and evaluation() function call.
*/
#include<stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include"fstuff.h"
#include"cbp.h"

extern char *namef[NUM_FUNCT+1];
extern char *namec[NUM_CFUNCT+1];
extern void cag();
enum cipher_type {permutation, substitution, feistel, benes, misty, other};

typedef struct cbp_struct {
  char *name;                  /* name of the encryption step */
  enum cipher_type f_type;     /* type of the encryption step  */
  unsigned long source_msb;    /* 32 msb of the source */
  unsigned long source_lsb;    /* 32 lsb of the source */
  unsigned long target_msb;    /* 32 msb of the target */
  unsigned long target_lsb;    /* 32 lsb of the target */
  float totdiff;               /* total diffusion */
  float margdiff;              /* marginal diffusion */
  float conf;                  /* confusion */
}cbp;

struct cbp_struct cbparr[NUM_CFUNCT+1];

static getcbp()
{
  int i;
  initcag();  /* get the names of the encryption steps */

  for(i=1;i<=NUM_CFUNCT;i++)
    cbparr[i].name=namec[i];

  cbparr[1].f_type=permutation;
  cbparr[2].f_type=substitution;
  cbparr[3].f_type=permutation;
  cbparr[4].f_type=feistel;
  cbparr[5].f_type=feistel;
  cbparr[6].f_type=feistel;
  cbparr[7].f_type=other;
  cbparr[8].f_type=other;
  cbparr[9].f_type=feistel;
  cbparr[10].f_type=feistel;
  cbparr[11].f_type=permutation;
  cbparr[12].f_type=permutation;
  cbparr[13].f_type=feistel;
  cbparr[14].f_type=feistel;
  cbparr[15].f_type=feistel;

  /* update source and target fields */
  cbparr[4].source_msb =0x00000000L;
  cbparr[4].source_lsb =0xffffffffL;
  cbparr[4].target_msb =0x00000000L;
  cbparr[4].target_lsb =0xffffffffL;
  cbparr[5].source_msb =0x00000000L;
  cbparr[5].source_lsb =0xffffffffL;
  cbparr[5].target_msb =0x00000000L;
  cbparr[5].target_lsb =0xffffffffL;
  cbparr[6].source_msb =0xffffffffL;
  cbparr[6].source_lsb =0x00000000L;
  cbparr[6].target_msb =0xffffffffL;
  cbparr[6].target_lsb =0x00000000L;
  cbparr[7].source_msb =0x000000ffL;
  cbparr[7].source_lsb =0x00000000L;
  cbparr[7].target_msb =0xffffff00L;
  cbparr[7].target_lsb =0xffffffffL;
  cbparr[8].source_msb =0x00000000L;
  cbparr[8].source_lsb =0x0000ffffL;
  cbparr[8].target_msb =0xffffffffL;
  cbparr[8].target_lsb =0xffff0000L;
  cbparr[9].source_msb =0xffffffffL;
  cbparr[9].source_lsb =0x00000000L;
  cbparr[9].target_msb =0xffffffffL;
  cbparr[9].target_lsb =0x00000000L;
  cbparr[10].source_msb=0x00000000L;
  cbparr[10].source_lsb=0xffffffffL;
  cbparr[10].target_msb=0x00000000L;
  cbparr[10].target_lsb=0xffffffffL;
  cbparr[13].source_msb=0x00000000L;
  cbparr[13].source_lsb=0xffffffffL;
  cbparr[13].target_msb=0x00000000L;
  cbparr[13].target_lsb=0xffffffffL;
  cbparr[14].source_msb=0x00000000L;
  cbparr[14].source_lsb=0x0000ffffL;
  cbparr[14].target_msb=0x0000ffffL;
  cbparr[14].target_lsb=0xffffffffL;
  cbparr[15].source_msb=0x000000ffL;
  cbparr[15].source_lsb=0xffffffffL;
  cbparr[15].target_msb=0x00000000L;
  cbparr[15].target_lsb=0x00ffffffL;

/* update total and marginal diffusion fields */
  cbparr[1].totdiff  =0.0156;
  cbparr[1].margdiff =0.0;
  cbparr[2].totdiff  =0.0588;
  cbparr[2].margdiff =0.0054;
  cbparr[3].totdiff  =0.0156;
  cbparr[3].margdiff =0.0;
  cbparr[4].totdiff  =0.0625;
  cbparr[4].margdiff =0.0104;
  cbparr[5].totdiff  =0.2656;
  cbparr[5].margdiff =0.1876;
  cbparr[6].totdiff  =0.2656;
  cbparr[6].margdiff =0.0613;
  cbparr[7].totdiff  =0.0293;
  cbparr[7].margdiff =0.0;
  cbparr[8].totdiff  =0.0457;
  cbparr[8].margdiff =0.0061;
  cbparr[9].totdiff  =0.2656;
  cbparr[9].margdiff =0.0552;
  cbparr[10].totdiff =0.2656;
  cbparr[10].margdiff=0.0645;
  cbparr[11].totdiff =0.0156;
  cbparr[11].margdiff=0.0;
  cbparr[12].totdiff =0.0156;
  cbparr[12].margdiff=0.0;
  cbparr[13].totdiff =0.3206;
  cbparr[13].margdiff=0.0671;
  cbparr[14].totdiff =0.2031;
  cbparr[14].margdiff=0.0498;
  cbparr[15].totdiff =0.2500;
  cbparr[15].margdiff=0.0530;

  /* confusion */
  cbparr[1].conf =0.0;
  cbparr[2].conf =0.0219;
  cbparr[3].conf =0.0;
  cbparr[4].conf =0.0415;
  cbparr[5].conf =0.2499;
  cbparr[6].conf =0.2499;
  cbparr[7].conf =0.0;
  cbparr[8].conf =0.1621;
  cbparr[9].conf =0.2492;
  cbparr[10].conf=0.2492;
  cbparr[11].conf=0.0;
  cbparr[12].conf=0.0;
  cbparr[13].conf=0.2905;
  cbparr[14].conf=0.1755;
  cbparr[15].conf=0.2336;
}

static unsigned int Hamming(x,y)         /* get Hamming distance between x and y */
```

```
unsigned long x,y;
{
  unsigned int h=0;
  int i;
    for(i=0;i<32;i++)
    h+=((x>>i)^(y>>i))&0x00000001;
  return h;
}

static int compare_st(s,t)      /* compare source and target and return */
                                /* the weight of their difference */
const int s,t;
{
  int w=0;
  if(cbparr[s].source_msb==cbparr[t].target_msb)
    if(cbparr[s].source_lsb==cbparr[t].target_lsb)
      return w;                 /* source==target, return 0 */
    else
      {
w=Hamming(cbparr[s].source_lsb,cbparr[t].target_lsb);
if(cbparr[s].source_lsb<cbparr[t].target_lsb)
  w*=-1;
return w;               /* source<target, return -(weight) */
      }
  else
    {
      w=Hamming(cbparr[s].source_msb,cbparr[t].target_msb);
      if(cbparr[s].source_msb<cbparr[t].target_msb)
w*=-1;
      if(cbparr[s].source_lsb==cbparr[t].target_lsb)
return w;
      else
{
  if(cbparr[s].source_msb<cbparr[t].target_msb)
    w-=Hamming(cbparr[s].source_lsb,cbparr[t].target_lsb);
  else
    w+=Hamming(cbparr[s].source_lsb,cbparr[t].target_lsb);
}
      return w;
    }
}

int evaluate(fi, cfi, length)   /* evaluate the cipher and accept or suggest */
                                /* a new one */
unsigned int fi[MAX_HEIGHT][MAX_LAYERS];
unsigned int cfi[MAX_LAYERS];
int *length;
{
  int i;
  float conf=0;       /* confusion estimation variable */
  float extra=0;      /* security specification */
  float p1,m1,mf1,d1;
  short int feistelflag=0;

  for(i=1;i<=*length;i++)
    {
      if(cbparr[cfi[i]].f_type==feistel)
{
  if(feistelflag==1) /* if not the first feistel block */
    switch(abs(compare_st(cfi[i],cfi[i-1])))
      {
      case 0:
conf*=1-146.19*cbparr[cfi[i]].conf-m1+585*cbparr[cfi[i]].margdiff-5.4*mf1;
conf+=-0.09+cbparr[cfi[i]].conf*(23.6*mf1-13.76+173.67*cbparr[cfi[i]].margdiff)+0.27*d1/m1;
      case 8:
conf*=1-146.19*cbparr[cfi[i]].conf-m1+585*cbparr[cfi[i]].margdiff-5.4*mf1;
conf+=-0.09+cbparr[cfi[i]].conf*(23.6*mf1-13.76+173.67*cbparr[cfi[i]].margdiff)+0.27*d1/m1;
      case 16:
  conf*=-5.98-408.5*cbparr[cfi[i]].conf-3.25*m1+1796*cbparr[cfi[i]].margdiff+mf1*(168.4-880*cbparr[cfi[i]].conf);
  conf+=-54.44+cbparr[cfi[i]].conf*(176.8-1486*cbparr[cfi[i]].margdiff-14.8*mf1)+7.94*cbparr[cfi[i]].totdiff/cbparr[cfi[i]].margdiff;
      default :
printf("\nCOMP:%d",abs(compare_st(cfi[i],cfi[i-1])));
if(conf>1)
  {
    conf=0.9999;
    extra++;
  }
m1=0.24722+0.22469*((float)log(conf));
if(m1<0)
  m1=0.0111; /* minimum value of marginal diff. */
mf1=cbparr[cfi[i]].margdiff;
d1=conf;
      }
  else
    {
      feistelflag=1;
      conf=cbparr[cfi[i]].conf;
      m1=cbparr[cfi[i]].margdiff;
      mf1=m1;
      d1=cbparr[cfi[i]].totdiff;
    }
}
      else
{
  /* check if diffusion is not zero or one. If false, replace the non-feistel type with the neighbour feistel one.*/
  if(conf>0 && extra<SEC_SPEC)
```

```
      {
        cfi[i]=cfi[i-1];
        switch(abs(compare_st(cfi[i],cfi[i-1])))
{
case 0:
  conf*=1-146.19*cbparr[cfi[i]].conf-m1+585*cbparr[cfi[i]].margdiff-5.4*mf1;
  conf+=-0.09+cbparr[cfi[i]].conf*(23.6*mf1-13.76+173.67*cbparr[cfi[i]].margdiff)+0.27*d1/m1;
case 8:
  conf*=1-146.19*cbparr[cfi[i]].conf-m1+585*cbparr[cfi[i]].margdiff-5.4*mf1;
  conf+=-0.09+cbparr[cfi[i]].conf*(23.6*mf1-13.76+173.67*cbparr[cfi[i]].margdiff)+0.27*d1/m1;
case 16:
  conf*=-5.98-408.5*cbparr[cfi[i]].conf-3.25*m1+1796*cbparr[cfi[i]].margdiff+mf1*(168.4-880*cbparr[cfi[i]].conf);
  conf+=-54.44+cbparr[cfi[i]].conf*(176.8-1486*cbparr[cfi[i]].margdiff-14.8*mf1)+7.94*cbparr[cfi[i]].totdiff/cbparr[cfi[i]].margdiff;
default :
  if(conf>1)
    {
      conf=0.9999;
      extra++;
    }
  m1=0.24722+0.22469*((float)log(conf));
  if(m1<0)
    m1=0.0111; /* minimum value of marginal diff. */
  mf1=cbparr[cfi[i]].margdiff;
  d1=conf;
}
      }
}
        printf("\nconf:%f marg:%f",conf,m1);
      }
}
```

D.11 testsuite.c

```
/*
   testsuite.c, Vasilios Katos, 1998.
   The test suite environment to measure the CBP values and cipher instances.
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <curses.h>
#include <time.h>
#include <math.h>
#include "fstuff.h"
#include "rmd128.h"
/*#include "des_locl.h"*/

#define PROMPT printf("\nABSENT>")
#define CHI_SQUARE 3.84          /* for a 5% significance level */
#define CHI_SQUARE63 82          /* for a 5% significance level, 63 d.f. */
#define CHI_SQUARE63_025 85      /* for a 2.5% significance level, 63 d.f.*/
#define CHI_SQUARE63_010 91       /* for a 1% significance level, 63 d.f.*/
#define CHI_SQUARE63_005 95       /* for a 0.5% significance level, 63 d.f.*/
#define CHI_SQUARE63_001 103      /* for a 0.1% significance level, 63 d.f.*/
#define CHI_SQUARE_SERIAL 5.99  /* for a 5% significance level, 2 d.f. */
#define Z_TEST 1.96             /* for a 5% confidence interval on a normal distribution test */
#define PLOT_HEIGHT 20 /* height of plots (in characters) */

extern void encrypt();
extern void decrypt();
extern void cag();
extern void copyarr();
extern void initcag();

unsigned char *(*nfunct[NUM_FUNCT+1])();
unsigned char *(*nefunct[NUM_CFUNCT+1])();
unsigned char *(*ndfunct[NUM_CFUNCT+1])();

extern char *namef[NUM_FUNCT+1];
extern char *namec[NUM_CFUNCT+1];
char key[]="abcdefgh";     /* define master key and assign a default value */

FILE *outfile,*datafile; /* the outfile is for the script and the other for the test results of (3), in order to be read by matlab */
char *matfile;            /* name of the data file for matlab */
int outfileflag=0;
int scriptflag=0;
int sel,inp,nb,incf,ni;      /* global variables for the tests */
void bin8(unsigned char i) /* display a byte in a binary format */
{
  int j=8;
  while(j--)
    {
      printf("%d",(i & 0x80)>>7);
      i <<=1;
    }
  printf(" ");
}

void fbin8(unsigned char i) /* display a byte in a binary format @ outfile */
{
  int j=8;
  while(j--)
    {
      fprintf(outfile,"%d",(i & 0x80)>>7);
      i <<=1;
    }
  fprintf(outfile," ");
}

void bin16(unsigned int i) /* display 2 bytes in a binary format */
{
  int j=16;
  while(j--)
    {
      printf("%d",(i & 0x8000)>>15);
      i <<=1;
    }
  printf(" ");
}

void bin32(unsigned long int i) /* display 4 bytes in a binary format */
{
  int j=32;
  while(j--)
    {
      printf("%d",(i & 0x80000000)>>31);
      i <<=1;
    }
  printf(" ");
}

void cagseed(fi, cfi, length, iseed)
unsigned int fi[MAX_HEIGHT][MAX_LAYERS];
unsigned int cfi[MAX_LAYERS];
int *length;
const int iseed;
{
  int i,j,k;
  int nf,nl;                    /* for the random */
  unsigned short seed[3];       /* function       */
  unsigned short lseed[3];
  /* Determine the size and indeces for the crypto algorithm */
  seed[0]=seed[1]=seed[2]=seed[3]=iseed;
  seed48(seed);
  *length=nl=MIN_LAYERS+((int) (MAX_LAYERS-MIN_LAYERS)*drand48());
  for (i=0;i<=MAX_LAYERS;i++)
    {
      cfi[i]=0;
      for (j=0;j<=MAX_HEIGHT;j++)
fi[j][i]=0;
    }
  for (i=1;i<=nl;i++)
    {
      cfi[i]= (int) ((NUM_CFUNCT)*drand48())+1;
      nf=(MIN_HEIGHT)+((int)(MAX_HEIGHT-MIN_HEIGHT)*drand48());
      for (j=1;j<=nf;j++)
do
  {
    int l;
    k=0;
    fi[j][i]= (int) ((NUM_FUNCT)*drand48())+1;
    for(l=1;l<j;l++)
      if(fi[j][i] == fi[l][i] )
k=1;
  }
  while(k);
    }
}

void noalg()
{
  printf("\nNo cryptographic algorithm defined.\nUse \"seed\", \"random\", or \"define\".");
}

void list()
{
  int i,j;
  printf("\nCryptographic primitives:\n");
  printf("-------------------------\n");
  for(i=1;i<NUM_CFUNCT-1;i++)
    if(i%2)
      printf("%d.%s\t%d.%s\n",i,namec[i],i+1,namec[i+1]);
  if(NUM_CFUNCT%2)
    printf("%d.%s\n",NUM_CFUNCT,namec[NUM_CFUNCT]);
  else
    printf("%d.%s\t%d.%s\n",NUM_CFUNCT-1,namec[NUM_CFUNCT-1],NUM_CFUNCT,namec[NUM_CFUNCT]);
  printf("\nFeedback blocks:\n");
  printf("----------------\n");
  for(i=1;i<NUM_FUNCT-1;i++)
    if(i%2)
      printf("%d.%s\t%d.%s\n",i,namef[i],i+1,namef[i+1]);
  if(NUM_FUNCT%2)
    printf("%d.%s\n",NUM_FUNCT,namef[NUM_FUNCT]);
}

void display(fi, cfi, length)   /* Display the indeces of the encryption and feedback blocks */
const unsigned int fi[MAX_HEIGHT][MAX_LAYERS];
const unsigned int cfi[MAX_LAYERS];
const int length;
{
  int i,j;

  printf("Layers selected:%i\n",length);
  for(i=1;i<=length;i++)
    {
      printf("%d. %d:",i,cfi[i]);
      for(j=1;j<MAX_HEIGHT;j++)
if(fi[j][i] != 0 )
  printf("%d-",fi[j][i]);
      printf("\n");
    }
}

void fdisplay(fi, cfi, length)                  /* print the algorithm to the output file */
const unsigned int fi[MAX_HEIGHT][MAX_LAYERS];
const unsigned int cfi[MAX_LAYERS];
const int length;
{
  int i,j;

  for(i=1;i<=length;i++)
    {
      fprintf(outfile,"%d:",cfi[i]);
      for(j=1;j<MAX_HEIGHT;j++)
if(fi[j][i] != 0 )
  fprintf(outfile,"%d-",fi[j][i]);
      fprintf(outfile,"\n");
    }
}

void graph(fi, cfi, length)   /* draw the cipher in terms of blocks */
const unsigned int fi[MAX_HEIGHT][MAX_LAYERS];
const unsigned int cfi[MAX_LAYERS];
const int length;
{
  char *feedbackfirst[]={"       /---------------\\   ",
 "       |               |  ",
 "   ___|",
 "   |   |               |  |",
 "   V   \\_______________/  |"};
  char *feedbackblock[]={"  |   /---------------\\  -",
 "   |   |               |  |",
 "   |___|",
 "   |   |               |  |",
 "   V   \\_______________/  |"};
  char *encryptblock[] ={"  |   /---------------\\  -",
 "   V   |               |  |",
 "->(+)->| ",
 "        |               |  ",
 "        \\_______________/  "};

  char *encryptfirst[] ={"       /---------------\\   ",
 "       |               |  ",
 "->---->| ",
 "        |               |  ",
 "        \\_______________/  "};
  int nozero[length+1];
  int i,j,k;

  for(j=MAX_HEIGHT-1;j>0;j--)
    {
      for(k=0;k<=4;k++)
 {
  printf("\n ");
  for(i=1;i<=length;i++)
    if(fi[j][i] !=0 )
      {
 if(j==MAX_HEIGHT-1)
   {
     printf("%s",feedbackfirst[k]);
     if(k==2)
       printf("%s|<- ",namef[fi[j][i]]);
   }
 else
   if(fi[j+1][i])
     {
       printf("%s",feedbackblock[k]);
       if(k==2)
 printf("%s|<-|",namef[fi[j][i]]);
     }
   else
     {
       printf("%s",feedbackfirst[k]);
       if(k==2)
 printf("%s|<- ",namef[fi[j][i]]);
     }
      }
    else
      printf("                          ");
 }
    }
  for(k=0;k<=4;k++)
    {
      printf("\n");
      if(k<2)
printf(" ");
      for(i=1;i<=length;i++)
{
  if(fi[1][i])
    {
      printf("%s",encryptblock[k]);
      if(k==2)
{
  printf("%s |--",namec[cfi[i]]);
  if(i==length)
    printf("-->ciphertext");
}
    }
  else
    {
      printf("%s",encryptfirst[k]);
      if(k==2)
{
  printf("%s |--",namec[cfi[i]]);
  if(i==length)
    printf("-->ciphertext");
}
    }
}
    }
}

void show(const int si)   /* demonstrate an encryption step transformation */
{
  unsigned int sfi[MAX_HEIGHT][MAX_LAYERS];
  unsigned int scfi[MAX_LAYERS];
  unsigned char sc[8]={'a','b','c','d','e','f','g','h'};

  int i,j;
  /*initialise the array of the cryptographic algorithm */
  for (i=0;i<=MAX_LAYERS;i++)
    {
      scfi[i]=0;
      for (j=0;j<=MAX_HEIGHT;j++)
sfi[j][i]=0;
    }
  scfi[1]=si;
  printf("Demonstrating cryptographic primitive %s",namec[si]);
  printf("\nInput:%s",sc);
  printf("\n\t\t\tKey:%s",key);
  i=16;
  encrypt(sfi,scfi,sc,1,&i);
  printf("\nOuput:%s",sc);
}

void speed(fi, cfi, length)  /* estimate speed of encryption/decryption */
const unsigned int fi[MAX_HEIGHT][MAX_LAYERS];
const unsigned int cfi[MAX_LAYERS];
const int length;
{
  clock_t t1,t0;
  static double dte=-1, dtd=-1;
  int ml,i;

  unsigned char speedc[10009];
  printf("Processing 10,000 characters...\n");
  memset(speedc,'\0',sizeof(speedc));
  srand(time(NULL));
  strcat(speedc,key);
  for(i=7;i<10008;i++)
    speedc[i]=(char)(rand() >> 7);
  t0=clock();
  ml=10008;
  encrypt(fi,cfi,speedc,length,&ml);
  t1=clock();
  printf("\nEncryption:\n-----------\nTest input processed in %g seconds.\n",
 (double)(t1-t0)/(double)CLOCKS_PER_SEC);
  printf("Characters processed per second: %g\n",
 (double)CLOCKS_PER_SEC*10000/((double)t1-t0));
  if(dte>=0)
    printf("Change in speed since last measure:%g%%\n",(double)(dte-t1+t0)/(double)(t1-t0)*100.0);
  if(outfileflag)
    {
      fprintf(outfile,"Encryption speed:%g characters/sec. (%g secs)\n",
        (double)CLOCKS_PER_SEC*10000/((double)t1-t0),(double)(t1-t0)/(double)CLOCKS_PER_SEC);
      if(dte>=0)
fprintf(outfile,"Change in speed:%g%%\n",(double)(dte-t1+t0)/(double)(t1-t0)*100.0);
    }
  dte=(double)(t1-t0);
  t0=clock();
  decrypt(fi,cfi,speedc,length,ml);
  t1=clock();
  printf("\nDecryption:\n------------\nTest input processed in %g seconds.\n",
 (double)(t1-t0)/(double)CLOCKS_PER_SEC);
  printf("Characters processed per second: %g\n",
 (double)CLOCKS_PER_SEC*10000/((double)t1-t0));
  if(dtd>=0)
    printf("Change in speed since last mesure:%g%%\n",(double)(dtd-t1+t0)/(double)(t1-t0)*100.0);
  if(outfileflag)
```

```
    {
      fprintf(outfile,"Decryption speed:%g characters/sec. (%g secs)\n",
       (double)CLOCKS_PER_SEC*10000/((double)t1-t0),(double)(t1-t0)/(double)CLOCKS_PER_SEC);
      if(dtd>=0)
fprintf(outfile,"Change in speed:%g%%\n",(double)(dtd-t1+t0)/(double)(t1-t0)*100.0);
    }
  dtd=(double)(t1-t0);
}

void plot(arr,range)  /* Graphs for frequency and weight of output bits */
const unsigned long arr[65];
const int range;
{
  int i,j;
  unsigned long max=0;

  if(range==64)
    {
      printf("\nFrequency\n ^");
      max=ni;
    }
  else
    {
      printf("\nWeight\n ^");
      for(i=0;i<range;i++)
if(arr[i]>max)
  max=arr[i];
    }
  for(j=0;j<PLOT_HEIGHT;j++)
    {
      if(j==PLOT_HEIGHT/2)
printf("\n-|");
      else
printf("\n |");
      for(i=0;i<range;i++)
{
  if((PLOT_HEIGHT-j)==(PLOT_HEIGHT*arr[i]/max))
    printf("*");
  else
    printf(" ");
}
    }
  printf("\n +");
  for(i=0;i<range;i++)
    printf("-");
}

void matmul64(a,b,c)      /* 64x64 matrix multiplication: c=axb */
const float a[64][64];
const float b[64][64];
float c[64][64];
{
  int i,j,k;

    for(i=0;i<64;i++)
for(j=0;j<64;j++)
  {
    c[i][j]=0;
    for(k=0;k<64;k++)
      c[i][j]+=a[i][k]*b[k][j];
  }
}

void matmul32(a,b,c)      /* 32x32 matrix multiplication: c=axb */
const float a[32][32];
const float b[32][32];
float c[32][32];
{
  int i,j,k;

    for(i=0;i<32;i++)
for(j=0;j<32;j++)
  {
    c[i][j]=0;
    for(k=0;k<32;k++)
      c[i][j]+=a[i][k]*b[k][j];
  }
}

int zeros(a,n)           /* number of zeros in a */
const float a[4096];     /* max dimension of matrix: 64x64 */
const int n;
{
  int i,z=0;

  for(i=0;i<n*n;i++)
    if(a[i]==0)
      z++;
  return z;
}


void test(fi, cfi, length)  /* tests on the encryption steps and cipher instances */
const unsigned int fi[MAX_HEIGHT][MAX_LAYERS];
const unsigned int cfi[MAX_LAYERS];
const int length;
```

```
{
  int unsigned long n,n0[64],n1[64],n10[64],n01[65],h[64][64]; /* n01 is also used to store histogram of weights */
  int d,di,dj,k,ml;
  register int i,j,ii,jj;
  int unsigned short window[64][64];
  int unsigned long uli,ulj,auto_of_d[64],sac;
  float x[64],mean,mean_d[64],chi_square,conf[64][64],pr_conf[64][64],prod[64][64],temq;
  struct{
   float m[32][32];
  }quad[5];              /* four quadrants of matrix */
  int dr[11]={0,0,0,0,0,0,0,0,0,0,0};  /* distinguisher results */
  float diff_rate,old_diff_rate,mdiff; /* diffusion rate, marginal diffusion */
  double var_d;
  unsigned char t[BUFSIZ];
  unsigned char tb[16],tc[16];
  int diff[64],weight;
  unsigned rand_offset;
  const unsigned char struct_in[]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,    /* structured input */
   0xff,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
   0x00,0xff,0x00,0x00,0x00,0x00,0x00,0x00,
   0x00,0x00,0xff,0x00,0x00,0x00,0x00,0x00,
   0x00,0x00,0x00,0xff,0x00,0x00,0x00,0x00,
   0x00,0x00,0x00,0x00,0xff,0x00,0x00,0x00,
   0x00,0x00,0x00,0x00,0x00,0xff,0x00,0x00,
   0x00,0x00,0x00,0x00,0x00,0x00,0xff,0x00,
   0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff,
   0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
   0x00,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
   0xff,0x00,0xff,0xff,0xff,0xff,0xff,0xff,
   0xff,0xff,0x00,0xff,0xff,0xff,0xff,0xff,
   0xff,0xff,0xff,0x00,0xff,0xff,0xff,0xff,
   0xff,0xff,0xff,0xff,0x00,0xff,0xff,0xff,
   0xff,0xff,0xff,0xff,0xff,0x00,0xff,0xff,
   0xff,0xff,0xff,0xff,0xff,0xff,0x00,0xff,
   0xff,0xff,0xff,0xff,0xff,0xff,0xff,0x00,
   0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00,
   0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff,
   0x00,0xff,0x00,0xff,0x00,0xff,0x00,0xff,
   0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa,0xaa};
  char *bike[]={" \\  |  /  -"," |  /  -  \\"," /  -  \\  |"," -  \\  |  /"};
  int bikeind=0,bikecount=0;       /* some animation on progress */
  char s[32];
  s[0]='\0';
  if(scriptflag==0)
    {
      inp=0;
      printf("Test to run:\n");
      printf("            1. Frequency\n");
      printf("            2. Serial\n");
      printf("            3. Confusion-diffusion\n");
      printf("            4. Autocorrelation\n");
      printf("            5. Diffusion matrix\n");
      printf("            6. Confusion matrix/depth analysis\n");
      printf("            7. Total diffusion/marginal diffusion\n");
      printf("            8. Diffusion distinguisher\n");
      printf("Select test:");
      scanf("%d",&sel);
      if(sel<3)
{
  printf("Input:\n");
  printf("      1. Linear (whole input)\n");
  printf("      2. Linear per byte\n");
  printf("      3. Random\n");
  printf("      4. Linear byte - constant others\n");
  printf("      5. Structured input\n");
  printf("Select input:");
  scanf("%d",&inp);
  if(inp==2 || inp==4)
    {
      printf("Byte to be changed linearly [0-7]:");
      scanf("%d",&nb);
    }
  if(inp!=3 && inp!=5 && sel!=4 && sel!=3)
    {
      printf("Increment factor:");
      scanf("%d",&incf);
    }
}
      if(sel==3 || (sel>=6 && sel<=8))
{
  printf("Test:\n");
  printf("     1. Plaintext\n");
  printf("     2. Key\n");
  printf("Select variable:");
  scanf("%d",&inp);
}
      if(inp!=5 && sel!=4 && sel!=7)
{
  printf("Loops :");
  scanf("%u",&ni);
}
      else
ni=sizeof(struct_in)/8;
    }
  else
    if(strlen(matfile)>1)             /* if a script is running and requests an */
```

```
      {
strcat(s,matfile);             /* output for raw data */
switch(sel)
  {
  case 3:
    strcat(s,".cd");  /* confusion/diffusion (block cipher test)*/
    break;
  case 6:
    strcat(s,".depth"); /* depth matrix */
    break;
  case 7:
    strcat(s,".mt");     /* marginal/total diff */
    break;
  }
      }
  if(sel==7)
    ni=100;
  if(sel==4)
    ni=sizeof(struct_in)/8;
  for(i=0;i<64;i++)
    n0[i]=n1[i]=n10[i]=n01[i]=0;
  strcat(tb,key);
  for(i=8;i<=15;i++)
    tc[i-8]=tb[i]=(char)(rand() >> 7 );
  memset(t,'\0',sizeof(t));
  switch(sel)       /* tests */
    {
    case 1:
      ulj=0;
      for(uli=0;uli<ni;uli++)
{
  strncat(t,tb,16);
  ml=16;
  encrypt(fi,cfi,t,length,&ml);
  k=0;
  for(i=8;i<16;i++)
    {
      j=8;
      while(j--)
{
  n1[k++]+=(t[i] & 0x80)>>7;
  t[i] <<=1;
}
    }
  if(inp==1)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
tb[i]+=incf;
ulj++;
if(ulj>(int)(256/incf))
  {
    for(i=8;i<=15;i++)
      tb[i]=(char)(rand() >> 7 );
    ulj=0;
  }
      }
  if(inp==2)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
if(i-8==nb)
  tb[i]+=incf;
ulj++;
if(ulj>(int)(256/incf))
  {
    for(i=8;i<=15;i++)
      if((i-8)!=nb)
tb[i]=(char)(rand() >> 7 );
    ulj=0;
  }
      }
  if(inp==3)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
tb[i]=(char)(rand() >> 7 );
      }
  if(inp==4)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
if(i-8==nb)
  tb[i]+=incf;
      }
  if(inp==5)
    for(i=0;i<8;i++)
      {
tb[i+8]=struct_in[uli*8+i];
      }
  if((bikecount++)>30)                /* progress animation */
    {
      printf("%s\r",bike[bikeind++%4]);
      fflush(stdout);
      bikecount=0;
    }
```

```
}
      for(i=0;i<64;i++)
x[i]=(float)(((ni-2.0*n1[i]))*(ni-2.0*n1[i]))/(float)(ni);
      for(j=0;j<8;j++)
{
  printf("\n\nOutput bit |");
  for(i=8*j;i<8*j+8;i++)
    printf("%2d |",i);
  printf("\n");
  for(i=0;i<44;i++)
    printf("-");
  printf("\n   PASS    |");
  for(i=8*j;i<8*j+8;i++)
    if(x[i]<=CHI_SQUARE)
      printf(" X |");
    else
      printf("   |");
  printf("\n   FAIL    |");
  for(i=8*j;i<8*j+8;i++)
    if(x[i]>CHI_SQUARE)
      printf(" X |");
    else
      printf("   |");
}
      printf("\n");
  if(outfileflag)              /* store results to output file */
    {
      fprintf(outfile,"Frequency:\n");
      for(i=0;i<64;i++)
if(x[i]<=CHI_SQUARE)
  fprintf(outfile,"P");
else
  fprintf(outfile,"F");
      fprintf(outfile,"\n");
    }

      break;
    case 2:
      ulj=0;
      for(uli=0;uli<ni;uli++)
{
  strncat(t,tb,16);
  ml=16;
  encrypt(fi,cfi,t,length,&ml);
  for(i=8;i<16;i++)
    tc[i]=t[i];
  k=0;
  for(i=8;i<16;i++)
    {
      j=8;
      if(uli)
while(j--)
  {
    n1[k]+=((t[i] & 0x80)>>7)&((tc[i-8] & 0x80)>>7);
    n0[k]+=!((t[i] & 0x80)>>7)&!((tc[i-8] & 0x80)>>7);
    n10[k]+=((t[i] & 0x80)>>7)&!((tc[i-8] & 0x80)>>7);
    n01[k++]+=!((t[i] & 0x80)>>7)&((tc[i-8] & 0x80)>>7);
    t[i] <<=1;
    tc[i-8] <<=1;
  }
      tc[i-8]=tc[i];
    }
  if(inp==1)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
tb[i]+=incf;
ulj++;
if(ulj>(int)(256/incf))
  {
    for(i=8;i<=15;i++)
      tb[i]=(char)(rand() >> 7 );
    ulj=0;
  }
      }
  if(inp==2)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
if(i-8==nb)
  tb[i]+=incf;
ulj++;
if(ulj>(int)(256/incf))
  {
    for(i=8;i<=15;i++)
      if((i-8)!=nb)
tb[i]=(char)(rand() >> 7 );
    ulj=0;
  }
      }
  if(inp==3)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
tb[i]=(char)(rand() >> 7 );
      }
  if(inp==4)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
if(i-8==nb)
  tb[i]+=incf;
      }
  if(inp==5)
    for(i=0;i<8;i++)
      {
tb[i+8]=struct_in[uli*8+i];
      }
  if((bikecount++)>30)
    {
      printf("%s\r",bike[bikeind++%4]);
      fflush(stdout);
      bikecount=0;
    }
}
      for(i=0;i<64;i++)
x[i]=(float)(4.0/(ni-1.0)*(n0[i]*n0[i]+n01[i]*n01[i]+n10[i]*n10[i]+n1[i]*n1[i])-
     2.0/ni*((n01[i]+n0[i])*(n01[i]+n0[i])+(n10[i]+n1[i])*(n10[i]+n1[i]))-1);
      for(j=0;j<8;j++)
{
  printf("\n\nOutput bit |");
  for(i=8*j;i<8*j+8;i++)
    printf("%2d |",i);
  printf("\n");
  for(i=0;i<44;i++)
    printf("-");
  printf("\n   PASS    |");
  for(i=8*j;i<8*j+8;i++)
    if(x[i]<=CHI_SQUARE_SERIAL)
      printf(" X |");
    else
      printf("   |");
  printf("\n   FAIL    |");
  for(i=8*j;i<8*j+8;i++)
    if(x[i]>CHI_SQUARE_SERIAL)
      printf(" X |");
    else
      printf("   |");
}
      printf("\n");
  if(outfileflag)             /* store results to output file */
    {
      fprintf(outfile,"Serial:\n");
      for(i=0;i<64;i++)
if(x[i]<=CHI_SQUARE_SERIAL)
  fprintf(outfile,"P");
else
  fprintf(outfile,"F");
      fprintf(outfile,"\n");
    }
      break;
    case 3:
      if(scriptflag==0)
{
  printf("Results:");
  gets(s);
  gets(s);
}
      if(strlen(s)>0)
datafile=fopen(s,"w");
      dj=-1;
      if(inp==2)
for(i=8;i<16;i++)
  t[i]=tb[i]=(char)(rand() >> 7 );
      for(di=0;di<64;di++)
{
  dj++;
  if(dj>7)
    dj=0;
  memset(n01,0,sizeof(n01));
  memset(diff,0,sizeof(diff));
  for(uli=0;uli<ni;uli++)
    {
      if(inp==1)
{
  for(i=0;i<8;i++)
    t[i]=tb[i]=key[i];
  for(i=8;i<16;i++)
    t[i]=tb[i]=(char)(rand() >> 7 );
}
      else      /* we want to keep input constant when examining key bits */
{
  for(i=0;i<8;i++)
    t[i]=tb[i]=key[i]=(char)(rand() >> 7 );
  for(i=8;i<16;i++)
    t[i]=tb[i];
}
      ml=16;
      encrypt(fi,cfi,t,length,&ml);
      k=0;
      for(i=8;i<16;i++)
{
  j=8;
  while(j--)
    {
      n1[k++]=(t[i] & 0x80)>>7;
      t[i] <<= 1;
    }
}
      if(inp==1)
for(i=0;i<16;i++)
  if(i==(int)di/8+8)
    t[(int)di/8+8]=tb[(int)di/8+8]^=0x80>>dj;
  else
    t[i]=tb[i];
      else
for(i=0;i<16;i++)
  if(i==(int)di/8)
    key[(int)di/8]=t[(int)di/8]=tb[(int)di/8]^=0x80>>dj;
  else
    t[i]=tb[i];
      ml=16;
      encrypt(fi,cfi,t,length,&ml);
      k=0;
      for(i=8;i<16;i++)
{
  j=8;
  while(j--)
    {
      n0[k++]=(t[i] & 0x80)>>7;
      t[i] <<= 1;
    }
}
      weight=0;
      for(i=0;i<64;i++)
{
  diff[i]+=n0[i]^n1[i];
  weight+=n0[i]^n1[i];
}
      n01[weight]++;
    }
  plot(diff,64);
  plot(n01,65);
  if(strlen(s)>0)
    {
      for(i=0;i<65;i++)
fprintf(datafile,"%d\n",n01[i]);
              for(i=0;i<64;i++)
fprintf(datafile,"%d\n",diff[i]);
    }
}
      printf("\n");
      if(strlen(s)>0)
fclose(datafile);
      if(outfileflag)
fprintf(outfile,"\n");
      break;
    case 4:
      ulj=sac=0;
      ni=sizeof(struct_in)/8;
      for(uli=0;uli<ni;uli++)
{
  strncat(t,tb,8);
  for(i=0;i<8;i++)
    t[i+8]=struct_in[uli*8+i];
  printf("\n");
  ml=16;
  encrypt(fi,cfi,t,length,&ml);
  n=k=0;
  for(i=8;i<16;i++)
    {
      bin8(t[i]);
      j=8;
      while(j--)
{
  n1[k++]=(t[i] & 0x80)>>7;
  n+=(t[i] & 0x80)>>7;          /* number of 1's */
  t[i] <<=1;
}
    }
  for(d=0;d<64;d++)
    {
      auto_of_d[d]=0;
      mean_d[d]=(float)(n*n*(64.0-d)/4096.0);
      for(i=0;i<64-d;i++)
auto_of_d[d]+=n1[i]*n1[i+d];
    }
  chi_square=0;
  for(i=0;i<64;i++)
    chi_square+=((float)auto_of_d[i]-mean_d[i])*((float)auto_of_d[i]-mean_d[i])/mean_d[i];
  if(chi_square<=CHI_SQUARE63)
    sac++;
  if(inp==1)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
tb[i]+=incf;
ulj++;
```

```
if(ulj>(int)(256/incf))
  {
    for(i=8;i<=15;i++)
      tb[i]=(char)(rand() >> 7 );
    ulj=0;
  }
      }
  if(inp==2)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
if(i-8==nb)
  tb[i]+=incf;
ulj++;
if(ulj>(int)(256/incf))
  {
    for(i=8;i<=15;i++)
      if((i-8)!=nb)
tb[i]=(char)(rand() >> 7 );
    ulj=0;
  }
      }
  if(inp==3)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
tb[i]=(char)(rand() >> 7 );
      }
  if(inp==4)
    for(i=8;i<16;i++)
      {
tb[i-8]=tb[i];
if(i-8==nb)
  tb[i]+=incf;
      }
  if((bikecount++)>30)
    {
      printf("%s\r",bike[bikeind++%4]);
      fflush(stdout);
      bikecount=0;
    }
}
      printf("\nPassed:%d of %d (%3.1f%%)\n",sac,ni,(float) sac/(float) ni*100.0);
      if(outfileflag)             /* store results to output file */
{
  fprintf(outfile,"Autocorrelation: Passed:%d of %d (%3.1f%%)\n",sac,ni,(float) sac/(float) ni*100.0);
}
      break;
    case 5:
      printf("\n      ");
      for(i=1;i<7;i++)
printf("            %d",i);
      printf("\nOut: ");
      for(i=0;i<64;i++)
printf("%d|",i%10);
      printf("\n");
      if(outfileflag)
{
  fprintf(outfile,"\nOut:");
  for(i=0;i<64;i++)
    if(i%8==0)
      fprintf(outfile,"|%d",i%10);
    else
      fprintf(outfile,"%d",i%10);
}
      dj=-1;
      memset(window,0,sizeof(window));
      for(di=0;di<64;di++)
{
  dj++;
  if(dj>7)
    dj=0;
/*   memset(diff,0,sizeof(diff));*/
  for(i=0;i<64;i++)
    diff[i]=0;
  for(uli=0;uli<ni;uli++)
    {
      for(i=0;i<8;i++)
t[i]=tb[i]=key[i];
      for(i=8;i<16;i++)
t[i]=tb[i]=(char)(rand() >> 7 );
      ml=16;
      encrypt(fi,cfi,t,length,&ml);
      k=0;
      for(i=8;i<16;i++)
{
  j=8;
  while(j--)
    {
      n1[k++]=(t[i] & 0x80)>>7;
      t[i] <<= 1;
    }
}
      for(i=0;i<16;i++)
if(i==(int)di/8+8)
  t[(int)di/8+8]=tb[(int)di/8+8]^=0x80>>dj;
```

```
else
  t[i]=tb[i];
      ml=16;
      encrypt(fi,cfi,t,length,&ml);
      k=0;
      for(i=8;i<16;i++)
{
  j=8;
  while(j--)
    {
      n0[k++]=(t[i] & 0x80)>>7;
      t[i] <<= 1;
    }
}
      for(i=0;i<64;i++)
{
if(diff[i]==0)
  diff[i]=n0[i]^n1[i];
if(uli==0)                    /* store the first matrix */
  window[di][i]=diff[i];      /* for autocorrelation    */
}
    }
  if(di%8==0)
    {
      printf("\n");
      for(i=0;i<132;i++)
printf("-");
    }
  if(di)
    {
      printf("\n%3d:",di);
      if(outfileflag)
fprintf(outfile,"\n%3d:",di);
    }
  else
    {
      printf("\nIn0:");
      if(outfileflag)
fprintf(outfile,"\nIn0:");
    }
  for(i=0;i<64;i++)
    {
      if(i%8==0)
{
  printf("|%d",diff[i]);
  if(outfileflag)
    fprintf(outfile,"|%d",diff[i]);
  continue;
}
      printf("%2d",diff[i]);
      if(outfileflag)
fprintf(outfile,"%1d",diff[i]);
    }
}
      printf("\n");
      /* autocorrelation */
      memset(x,0.0,sizeof(x));
      for(i=0;i<64;i++)
  {
  for(j=0;j<32;j++)
    {
      x[i]+=window[i][j]*window[i][j+32];
    }
}
      mean=0.5*0.5*(64.0-32.0)/(64.0*64.0);  /* expected value of x[i] */
      printf("\nautocorrelation of rows...");
      if(outfileflag)
fprintf(outfile,"\ninput autocorrelation:");
      for(j=0;j<8;j++)
{
  printf("\n\nInput bit |");
  for(i=8*j;i<8*j+8;i++)
    printf("%2d |",i);
  printf("\n");
  for(i=0;i<44;i++)
    printf("-");
  printf("\n   PASS    |");
  for(i=8*j;i<8*j+8;i++)
    if(((float)x[i]-mean)*((float)x[i]-mean)/mean>CHI_SQUARE63_001)
      {
printf(" X |");
if(outfileflag)
  fprintf(outfile,"P");
      }
    else
      {
printf("   |");
if(outfileflag)
  fprintf(outfile,"F");
      }
  printf("\n   FAIL    |");
  for(i=8*j;i<8*j+8;i++)
    if(((float)x[i]-mean)*((float)x[i]-mean)/mean<=CHI_SQUARE63_001)
      printf(" X |");
    else
      printf("   |");
```

```
}
      if(outfileflag)
fprintf(outfile,"\n");
      memset(x,0.0,sizeof(x));
      for(i=0;i<64;i++)
  {
  for(j=0;j<32;j++)
    {
      x[i]+=window[j][i]*window[j+32][i];
    }
}
      mean=0.5*0.5*(64.0-32.0)/(64.0*64.0);  /* expected value of x[i] */
      printf("\nautocorrelation of columns...");
      if(outfileflag)
fprintf(outfile,"\noutput autocorrelation:");
      for(j=0;j<8;j++)
{
  printf("\n\nOutput bit |");
  for(i=8*j;i<8*j+8;i++)
    printf("%2d |",i);
  printf("\n");
  for(i=0;i<44;i++)
    printf("-");
  printf("\n   PASS    |");
  for(i=8*j;i<8*j+8;i++)
    if(((float)x[i]-mean)*((float)x[i]-mean)/mean>CHI_SQUARE63_001)
      {
printf(" X |");
if(outfileflag)
  fprintf(outfile,"P");
      }
    else
      {
printf("   |");
if(outfileflag)
  fprintf(outfile,"F");
      }
  printf("\n   FAIL    |");
  for(i=8*j;i<8*j+8;i++)
    if(((float)x[i]-mean)*((float)x[i]-mean)/mean<=CHI_SQUARE63_001)
      printf(" X |");
    else
      printf("   |");
}
      if(outfileflag)
fprintf(outfile,"\n");
      break;
    case 6:
      if(scriptflag==0)
{
  printf("Results:");
  gets(s);
  gets(s);
}
      if(strlen(s)>0)
datafile=fopen(s,"w");
      dj=-1;
      rand_offset=rand();
      if(inp==2)
for(i=8;i<16;i++)
  t[i]=tb[i]=(char)(rand() >> 7 );
      memset(conf,0,sizeof(conf));
      memset(window,0,sizeof(window));
      for(i=0;i<64;i++)
for(j=0;j<64;j++)
  h[i][j]=ni;
      for(uli=0;uli<ni;uli++){
if(inp==1)
  {
    for(i=0;i<8;i++)
      t[i]=tb[i]=key[i];
    for(i=8;i<16;i++)
      t[i]=tb[i]=(char)(rand() >> 7 );
  }
else       /* we want to keep input constant when examining key bits */
  {
    for(i=0;i<8;i++)
      t[i]=tb[i]=key[i]=(char)(rand() >> 7 );
    for(i=8;i<16;i++)
      t[i]=tb[i];
  }
for(di=0;di<64;di++){
  dj++;
  if(dj>7)
    dj=0;
  memset(n01,0,sizeof(n01));
  memset(diff,0,sizeof(diff));
  for(i=0;i<16;i++)
    t[i]=tb[i];
  ml=16;
  encrypt(fi,cfi,t,length,&ml);
  k=0;
  for(i=8;i<16;i++)
    {
      j=8;
      while(j--)
{
  n1[k++]=(t[i] & 0x80)>>7;
  t[i] <<= 1;
}
    }
  if(inp==1)
    for(i=0;i<16;i++)
      if(i==(int)di/8+8)
t[(int)di/8+8]=tb[(int)di/8+8]^0x80>>dj;
      else
t[i]=tb[i];
  else
    for(i=0;i<16;i++)
      if(i==(int)di/8)
key[(int)di/8]=t[(int)di/8]=tb[(int)di/8]^0x80>>dj;
      else
t[i]=tb[i];
  ml=16;
  encrypt(fi,cfi,t,length,&ml);
  k=0;
  for(i=8;i<16;i++)
    {
      j=8;
      while(j--)
{
  n0[k++]=(t[i] & 0x80)>>7;
  t[i] <<= 1;
}
    }
  for(i=0;i<64;i++)
    {
      diff[i]+=n0[i]^n1[i];
      conf[di][i]+=diff[i];     /* confusion for one loop */
      window[di][i]<<=1;
      window[di][i]|=diff[i];
    }
  if((bikecount++)>30)                  /* progress animation */
    {
      printf("%s\r",bike[bikeind++%4]);
      fflush(stdout);
      bikecount=0;
    }
}
if(uli>2)
    for(i=0;i<64;i++)
      for(j=0;j<64;j++)
{
  ulj=0;
  for(ii=0;ii<64;ii++)
    for(jj=0;jj<64;jj++)
      if(h[i][j]==ni) /* if already updated, not necessary to go further*/
if(window[i][j]!=window[ii][jj])
  ulj++;
  if(ulj>4032)
    h[i][j]=uli;
}
}
      printf("\nFirst quadrant\n");
      if(outfileflag)
fprintf(outfile,"\nFirst quadrant\n");
      for(i=0;i<32;i++)
{
  for(j=0;j<32;j++)
    {
      conf[i][j]=conf[i][j]/ni;      /* normalise */
      printf("%1.2g ",conf[i][j]);
      if(outfileflag)
fprintf(outfile,"%1.2g ",conf[i][j]);
    }
  printf("\n");
  if(outfileflag)
    fprintf(outfile,"\n");
}
      printf("Second quadrant\n");
     if(outfileflag)
fprintf(outfile,"Second quadrant\n");
      for(i=32;i<64;i++)
{
  for(j=0;j<32;j++)
    {
      conf[i][j]=conf[i][j]/ni;      /* normalise */
      printf("%1.2g ",conf[i][j]);
      if(outfileflag)
fprintf(outfile,"%1.2g ",conf[i][j]);
    }
  printf("\n");
  if(outfileflag)
    fprintf(outfile,"\n");
}
      printf("Third quadrant\n");
     if(outfileflag)
fprintf(outfile,"Third quadrant\n");
      for(i=0;i<32;i++)
{
  for(j=32;j<64;j++)
    {
      conf[i][j]=conf[i][j]/ni;      /* normalise */
      printf("%0.2g ",conf[i][j]);
      if(outfileflag)
fprintf(outfile,"%1.2g ",conf[i][j]);
    }
  printf("\n");
  if(outfileflag)
    fprintf(outfile,"\n");
}
      printf("Fourth quadrant\n");
     if(outfileflag)
fprintf(outfile,"Fourth quadrant\n");
      for(i=32;i<64;i++)
{
  for(j=32;j<64;j++)
    {
      conf[i][j]=conf[i][j]/ni;      /* normalise */
      printf("%1.2g ",conf[i][j]);
      if(outfileflag)
fprintf(outfile,"%1.2g ",conf[i][j]);
    }
  printf("\n");
  if(outfileflag)
    fprintf(outfile,"\n");
}
      printf("Depth matrix:\n");
      if(outfileflag)
fprintf(outfile,"Depth matrix:\n");
      for(i=0;i<64;i++)
{
  for(j=0;j<64;j++)
    {
      printf("%2d ",h[i][j]);
      if(strlen(s)>0)
fprintf(datafile,"%2d ",h[i][j]);
      if(outfileflag)
fprintf(outfile,"%2d ",h[i][j]);
    }
  if(strlen(s)>0)
    fprintf(datafile,"\n");
  if(outfileflag)
    fprintf(outfile,"\n");
  printf("\n");
}
      if(strlen(s)>0)
fclose(datafile);
      if(outfileflag)
fprintf(outfile,"\n");
      break;
    case 7:
      if(scriptflag==0)
{
  printf("Results:");
  gets(s);
  gets(s);
}
      if(strlen(s)>0)
datafile=fopen(s,"w");
      printf("Diffusion | M. diffusion\n");
      printf("------------------------\n");
      if(outfileflag)
{
  if(inp==2)
    fprintf(outfile,"variable=KEY\n");
  else
    fprintf(outfile,"variable=PLAINTEXT\n");
  fprintf(outfile,"Diffusion | M. diffusion\n------------------------\n");
}
dj=-1;
      if(inp==2)                         /* assign a random input */
for(i=8;i<16;i++)
  t[i]=tb[i]=(char)(rand() >> 7 );
      memset(conf,0,sizeof(conf));      /* initialise confusion matrix */
      old_diff_rate=0;
      ii=0;
      for(uli=0;uli<ni;uli++){          /* repeat test "ni" times */
if(inp==1)
  {
    for(i=0;i<8;i++)
      t[i]=tb[i]=key[i];
    for(i=8;i<16;i++)
      t[i]=tb[i]=(char)(rand() >> 7 );
  }
else       /* we want to keep input constant when examining key bits */
  {
    for(i=0;i<8;i++)
      t[i]=tb[i]=key[i]=(char)(rand() >> 7 );
    for(i=8;i<16;i++)
      t[i]=tb[i];
  }
for(di=0;di<64;di++){           /* loop to test all input (or key) bits */
  dj++;                         /* control bits within the current byte */
  if(dj>7)
    dj=0;
  memset(n01,0,sizeof(n01));    /* initialise array of computations */
  for(i=0;i<16;i++)             /* backup of input */
```

```
      t[i]=tb[i];
    ml=16;                          /* message length=1 key block & one plaintext block */
    encrypt(fi,cfi,t,length,&ml);
    k=0;
    for(i=8;i<16;i++)               /* check only the ciphertext excluding the key */
      {
        j=8;
        while(j--)                  /* extract bits from the current byte */
{
  n1[k++]=(t[i] & 0x80)>>7;
  t[i] <<= 1;
}
      }
    if(inp==1)                      /* invert one input plaintext bit */
      for(i=0;i<16;i++)
        if(i==(int)di/8+8)
t[(int)di/8+8]=tb[(int)di/8+8]^0x80>>dj;
        else
t[i]=tb[i];
    else                            /* invert one key bit */
      for(i=0;i<16;i++)
        if(i==(int)di/8)
key[(int)di/8]=t[(int)di/8]=tb[(int)di/8]^0x80>>dj;
        else
t[i]=tb[i];
    ml=16;
    encrypt(fi,cfi,t,length,&ml); /* perform second encryption */
    k=0;
    for(i=8;i<16;i++)
      {
        j=8;
        while(j--)
{
  n0[k++]=(t[i] & 0x80)>>7;
  t[i] <<= 1;
}
      }
                                          /* end loop for input bits */
    for(i=0;i<64;i++)                     /* update confusion matrix */
      conf[di][i]+=n0[i]^n1[i];
}
diff_rate=0;
for(i=0;i<64;i++)
  for(j=0;j<64;j++)
    if(conf[j][i])
      diff_rate++;
diff_rate/=(64.0*64.0);
if(uli)
  mdiff=(diff_rate-old_diff_rate); /* compute marginal diffusion */
else
  mdiff=0;
if(strlen(s)>0 && uli==1)
  fprintf(datafile,"%9.4f ",mdiff);  /* write marginal diffusion */
old_diff_rate=diff_rate;
printf("%6.4f    %9.4f\n",diff_rate,mdiff);        /* output diffusion & marginal diffusion */
if(outfileflag)
  fprintf(outfile,"%6.4f    %9.4f\n",diff_rate,mdiff);
if(diff_rate==1.0)                               /* exit test if diffusion==1 */
  {
    printf("Complete diffusion in %d tests.\n",uli);
    if(outfileflag)
      fprintf(outfile,"Complete diffusion in %d tests.",uli);
    break;
  }
if(mdiff==0 && (ii++)>4)                         /* exit test if diffusion is constant for more than 5 tests */
  {
    printf("Maximum diffusion achieved after %d tests.\n",uli-4);
    if(outfileflag)
      fprintf(outfile,"Maximum diffusion achieved after %d tests.",uli-4);
    break;
  }
else
  if(mdiff)
    ii=0;
      }
      if(strlen(s)>0)
        {
  fprintf(datafile,"%6.4f\n",diff_rate);
  fclose(datafile);
        }
        if(outfileflag)
 fprintf(outfile,"\n");
        break;
      case 8:
        dj=-1;
        rand_offset=rand();
        if(inp==2)
 for(i=8;i<16;i++)
   t[i]=tb[i]=(char)(rand() >> 7 );
        memset(conf,0,sizeof(conf));
        for(i=0;i<64;i++)
 for(j=0;j<64;j++)
   h[i][j]=ni;
        for(uli=0;uli<ni;uli++){
        memset(conf,0,sizeof(conf));
 if(inp==1)
```

```
  {
    for(i=0;i<8;i++)
      t[i]=tb[i]=key[i];
    for(i=8;i<16;i++)
      t[i]=tb[i]=(char)(rand() >> 7 );
  }
else       /* we want to keep input constant when examining key bits */
  {
    for(i=0;i<8;i++)
      t[i]=tb[i]=key[i]=(char)(rand() >> 7 );
    for(i=8;i<16;i++)
      t[i]=tb[i];
  }
for(di=0;di<64;di++){
  dj++;
  if(dj>7)
    dj=0;
  memset(n01,0,sizeof(n01));
  memset(diff,0,sizeof(diff));
  for(i=0;i<16;i++)
    t[i]=tb[i];
  ml=16;
  encrypt(fi,cfi,t,length,&ml);
  k=0;
  for(i=8;i<16;i++)
    {
      j=8;
      while(j--)
{
  n1[k++]=(t[i] & 0x80)>>7;
  t[i] <<= 1;
}
    }
  if(inp==1)
    for(i=0;i<16;i++)
      if(i==(int)di/8+8)
t[(int)di/8+8]=tb[(int)di/8+8]^0x80>>dj;
      else
t[i]=tb[i];
  else
    for(i=0;i<16;i++)
      if(i==(int)di/8)
key[(int)di/8]=t[(int)di/8]=tb[(int)di/8]^0x80>>dj;
      else
t[i]=tb[i];
  ml=16;
  encrypt(fi,cfi,t,length,&ml);
  k=0;
  for(i=8;i<16;i++)
    {
      j=8;
      while(j--)
{
  n0[k++]=(t[i] & 0x80)>>7;
  t[i] <<= 1;
}
    }
  for(i=0;i<64;i++)
    {
      diff[i]+=n0[i]^n1[i];
      conf[di][i]+=diff[i];    /* confusion for one loop */
    }
}
          matmul64(conf,conf,prod);
  printf("\nZeros in product:%f",(float) zeros(prod,64)/(64.0*64.0));
  printf("\nexpected zeros in prod:%f\n",(float) pow((2.0*64*64-zeros(prod,64)-1)*zeros(prod,64)/(64*64*(64*64-1)),64));
  if(outfileflag)
    {
      fprintf(outfile,"\nZeros in product:%f",(float) zeros(prod,64)/(64.0*64.0));
      fprintf(outfile,"\nexpected zeros in prod:%f\n",(float) pow((2.0*64*64-zeros(prod,64)-1)*zeros(prod,64)/(64*64*(64*64-1)),64));
    }
  memcpy(pr_conf,conf,sizeof(conf));
  matmul64(pr_conf,pr_conf,prod);
  /*  break conf. matrix into its four quadrants */
  for(i=0;i<32;i++)
    for(j=0;j<32;j++)
      {
quad[1].m[i][j]=conf[i][j];
quad[2].m[i][j]=conf[i][j+32];
quad[3].m[i][j]=conf[i+32][j];
quad[4].m[i][j]=conf[i+32][j+32];
      }
  printf("\n                 Expected                                 Actual\n");
  printf("        Q1        Q2        Q3        Q4            Q1        Q2        Q3        Q4 \n");
  printf(" ----------------------------------------    -----------------------------------");
  if(outfileflag)
    {
  fprintf(outfile,"\n                 Expected                                 Actual\n");
  fprintf(outfile,"        Q1        Q2        Q3        Q4            Q1        Q2        Q3        Q4 \n");
  fprintf(outfile," ----------------------------------------    -----------------------------------");
    }
  for(i=1;i<=4;i++)
    {
      printf("\nQ%d ",i);
      if(outfileflag)
fprintf(outfile,"\nQ%d ",i);
```

```
      for(j=1;j<=4;j++)
{
  matmul32(quad[i].m,quad[j].m,prod);
if(i==j)
  {
    printf("%1.6f ",(float) pow((2.0*32*32-zeros(prod,32)-1)*zeros(prod,32)/(32*32*(32*32-1)),32));
    if(outfileflag)
      fprintf(outfile,"%1.6f ",(float) pow((2.0*32*32-zeros(prod,32)-1)*zeros(prod,32)/(32*32*(32*32-1)),32));
  }
else
  {
    temq=(float)zeros(quad[i].m,32)/(1024.0)+(float)zeros(quad[j].m,32)/(32.0*32.0);
    temq-=(float)zeros(quad[i].m,32)*zeros(quad[j].m,32)/(float)pow(32.0,4);
    printf("%1.6f ",(float) pow(temq,32));
    if(outfileflag)
      fprintf(outfile,"%1.6f ",(float) pow(temq,32));
  }
}
      printf("  Q%d ",i);
      if(outfileflag)
fprintf(outfile,"  Q%d ",i);
      for(j=1;j<=4;j++)
{
  matmul32(quad[i].m,quad[j].m,prod);
  printf("%1.6f ",(float) zeros(prod,32)/(32*32));
  if(outfileflag)
    fprintf(outfile,"%1.6f ",(float) zeros(prod,32)/(32*32));
}
     }
       }
       if(outfileflag)
fprintf(outfile,"\n");
       break;
     }
}

int load(fi,cfi,length,filename)     /* returns 0 on success, 1 otherwise */
unsigned int fi[MAX_HEIGHT][MAX_LAYERS];
unsigned int cfi[MAX_LAYERS];
int *length;
char *filename;

{
  FILE *fd;
  char *s;
  int i,j,pos,dc;

  s=malloc(128*sizeof(char));
  if((fd=fopen(filename,"r"))==NULL)
    {
      printf("Error opening file.");
      return 1;
    }
  else
    {
      fgets(s,80,fd);
      if(!strncmp(s,"ABSENT",6))
 {
   /* Delete any current cryptographic algorithms */
   for (i=0;i<=MAX_LAYERS;i++)
     {
       pos=cfi[i]=0;
       for (j=0;j<=MAX_HEIGHT;j++)
 fi[j][i]=0;
     }

   *length=0;
   while(fgets(s,80,fd)!=NULL)
     {
       (*length)++;
       j=atoi(s);
       if(j<1 || j>NUM_CFUNCT)
 j=1;
       cfi[*length]=j;
       pos=1;
       for(j=0;j<strlen(s);j++)
 {
   if(isspace(s[j]) && pos < MAX_HEIGHT)
     {
       dc=atoi(&s[j]);
       if(dc>NUM_FUNCT)
 dc=1;
       if(dc<1)
 continue;
       fi[pos++][*length]=dc;
     }
 }
     }
   display(fi,cfi,*length);
 }
      else
 {
   printf("Not an ABSENT algorithm file.");
   fclose(fd);
   return 1;
 }
```

```
fclose(fd);
return 0;
    }
}

main()
{
  unsigned int findex[MAX_HEIGHT][MAX_LAYERS];
  unsigned int cfindex[MAX_LAYERS];
  unsigned int tfindex[MAX_HEIGHT][MAX_LAYERS];   /* temp algorithm buffers */
  unsigned int tcfindex[MAX_LAYERS];              /* temp algorithm buffers */
  int neighbourtype=0;                            /* specifies the type of neighbourhood search */
  int repeatcount=0;                            /* number of neighbours */
  int i,j,k,dc,ndc;
  unsigned char c1[BUFSIZ-1],c[BUFSIZ-1],*pc,*tpc,*enc,*tenc;
  unsigned char li[BUFSIZ-1], lj[MSG-1];
  unsigned char outbuf[BUFSIZ-1],dca[BUFSIZ-1];
  int pos;
  int len,mlen,nblocks,original_len;
  char *select,*s,*ss,*sp,*ssp;
  FILE *plain,*cipher;
  unsigned long filepos;

  initcag();
 /*initialise the array of the cryptographic algorithm */
  for (i=0;i<=MAX_LAYERS;i++)
    {
      pos=cfindex[i]=0;
      for (j=0;j<=MAX_HEIGHT;j++)
 findex[j][i]=0;
    }
  len=0;

  for(i=0;i<60;i++)
    printf("\n");
  printf("Welcome to ABSOLUTE ENCRYPTION test suite\n\n");
  printf("Type ? for help.\n");
  select=malloc(64*sizeof(char));
  s=malloc(128*sizeof(char));
  sp=s;
  ss=malloc(32*sizeof(char));
  ssp=ss;
  matfile=malloc(64*sizeof(char));
  do
    {
      PROMPT;
      gets(select);
      if(!strncmp(select,"?",1) && (strncmp(select,"? script",8)))
 {
   printf("seed         - create an algorithm by giving a seed\n");
   printf("random       - create automatically a random algorithm\n");
   printf("define       - construct an algorithm manually\n");
   printf("edit         - edit the current algorithm\n");
   printf("list         - list of cryptographic primitives\n");
   printf("display      - display the current algorithm\n");
   printf("graph        - draw the current algorithm\n");
   printf("show         - demonstrate the transformation of a cryptographic primitive\n");
   printf("key[bin]     - display or change the key [in binary format]\n");
   printf("encrypt      - encrypt a string\n");
   printf("run          - run encryption/decryption sequences\n");
   printf("speed        - perform a time trial on the current algorithm\n");
   printf("test         - mesure confusion/diffusion\n");
   printf("ciphertext   - encrypt a file\n");
   printf("plaintext    - decrypt a file\n");
   printf("save         - save the current algorithm\n");
   printf("load         - load an algorithm\n");
   printf("script       - execute an ABSENT script (? script for help).\n");
   printf("quit         - quit ABSOLUTE ENCRYPTION\n");
 }
      if(!strncmp(select,"? script" ,8))
 {
   printf("Available commands for an ABSENT script file: \n");
   printf("outfile=<filename>      - output file\n");
   printf("matlab=<filename>       - output file for matlab\n");
   printf("repeat= <n>             - begins a loop to run <n> tests. \n");
   printf("next                    - closes the repeat loop\n");
   printf("random                  - create automatically a random algorithm\n");
   printf("seed=<s>                - create an algorithm based on seed <s>\n");
   printf("load <filename>         - load an algorithm\n");
   printf("edit <l> <new>          - edit layer <l> and replace with <new> \n");
   printf("speed                   - perform a time trial on the current algorithm\n");
   printf("key=<k>                 - run tests with key <k>\n");
   printf("keybin=<k>              - as above, but keys should be in binary format\n");
   printf("loops=<n>               - number of loops\n");
   printf("input=lin  [<b>] |      - linear input [byte <b>] OR\n");
   printf("       linct <b>  |     - one byte linear and others constant OR\n");
   printf("       rand             - random input\n");
   printf("       struct           - structured input\n");
   printf("       plain            - select plaintext as variable for diffusion/confusion related tests\n");
   printf("       key              - select key as variable for diffusion/confusion related tests\n");
   printf("freq                    - run the frequency test\n");
   printf("serial                  - run the serial test\n");
   printf("autocor                 - run the autocorrelation test\n");
   printf("block                   - run the block cipher test\n");
   printf("diffusion               - calculate diffusion\n");
   printf("margdif                 - calculate marginal diffusion\n");
  printf("depth                        - calculate depth matrix\n");
  printf("zero                         - zero-density distinguisher on diffusion\n");
  printf("newtest                      - start a new test\n");
}
      s=sp;
     if(!strncmp(select,"ls" ,2))
{
  system("ls");
}
      if(!strncmp(select,"seed",4))
{
  if(strlen(select)>4)
    i=(int)(select[5]);
  else
    {
      printf("seed:");
      gets(s);
      i=(int)(s[0]);
    }
  cagseed(findex,cfindex,&len,i);
  display(findex,cfindex,len);
}
      if(!strncmp(select,"random",6))
{
  cag(findex,cfindex,&len);
  display(findex,cfindex,len);
}
      if(!strncmp(select,"display",7))
{
  if(cfindex[1])
    display(findex,cfindex,len);
  else
    noalg();
}

      if(!strncmp(select,"graph",5))
{
  if(cfindex[1])
    graph(findex,cfindex,len);
  else
    noalg();
}
      if(!strncmp(select,"key",3) && strncmp(select,"keybin",6))
{
  printf("Old key=%s\n",key);
  printf("New key=");
  gets(s);
  if(strlen(s))
    for(i=0;i<8;i++)
      if(i<strlen(s))
key[i]=s[i];
      else
key[i]=' ';
}
      if(!strncmp(select,"keybin",6))
{
  printf("Old key=");
  for(i=0;i<8;i++)
    bin8(key[i]);
  printf("\nNew key=");
  memset(s,'\0',sizeof(s));
  gets(s);
  pos=0;
  if(strlen(s))
    {
      for(i=0;i<8;i++)
key[i]=0;
      for(i=0;i<8;i++)
for(j=7;j>=0;j--)
  if((pos<strlen(s)) && (s[pos]=='1') && (!isspace(s[pos])))
    {
      key[i]+=(0x01 << j);
      pos++;
    }
  else
    {
      while(isspace(s[pos]))
pos++;
      pos++;
    }
    }
}
      if(!strncmp(select,"show",4))
{
  if((j=strlen(select))>4)
    {
      if((j=atoi(&select[5]))>NUM_CFUNCT)
j=0;
      if(j<=0)
j=0;
    }
  else
    {
      list();
      printf("Cryptographic primitive(1-%d):",NUM_CFUNCT);
      gets(s);
      if((j=atoi(s))>NUM_CFUNCT)
j=0;
      if(j<=0)
j=0;
    }
  if(j)
    show(j);
}
      if(!strncmp(select,"ciphertext",10))
{
  if(cfindex[1])
    {
      if((j=strlen(select))>10)
{
  s=&select[11];
  for(i=0;i<strlen(s);i++)
    if(isspace(s[i]))
      s[i]='\0';
}
      else
{
  printf("Source file:");
  gets(s);
  for(i=0;i<strlen(s);i++)
    if(isspace(s[i]))
      s[i]='\0';
}
      printf("Source file:%s ",s);
      if((plain=fopen(s,"r"))==NULL)
printf("does not exist.");
      else
{
  printf("\nDestination file:");
  gets(ss);
  for(i=0;i<strlen(ss);i++)
    if(isspace(ss[i]))
      ss[i]='\0';
  if((cipher=fopen(ss,"wb"))==NULL)
    {
      printf("Error opening destination file");
      close(plain);
    }
  else
    {
      memset(c1,'\0',sizeof(c1));
      memset(c,'\0',sizeof(c));
      while((pos=fread(c1,sizeof(char),BUFSIZ-16,plain))>0)
{
  strcat(c,key);
  strcat(c,c1);
  pos=strlen(c)+1;
  if((pos-1)%MSG==0)
    c[pos++]=' ';
  j=0;
  for(i=pos-1;i<MSG*(((pos-1)%MSG==0) ? (pos-1)/MSG : (pos-1)/MSG+1);i++,j++)
    c[i]=' ';
  mlen=pos+j-9;
  encrypt(findex,cfindex,c,len,&mlen);
  for(i=8;i<mlen;i++)
    if(c[i]>31 && c[i]!=127)
      printf("%c",c[i]);
  if(j=fwrite(c,sizeof(char),mlen,cipher)<0)
    printf("\nError writing file");
  memset(c1,'\0',sizeof(c1));
  memset(c,'\0',sizeof(c));
}
      fclose(cipher);
      fclose(plain);
    }
}
    }
  else
    noalg();
}
      if(!strncmp(select,"plaintext",9))
{
  if(cfindex[1])
    {
      if((j=strlen(select))>9)
{
  s=&select[10];
  for(i=0;i<strlen(s);i++)
    if(isspace(s[i]))
      s[i]='\0';
}
      else
{
  printf("Source file:");
  gets(s);
  for(i=0;i<strlen(s);i++)
    if(isspace(s[i]))
      s[i]='\0';
}
      printf("Source file:%s ",s);
      if((cipher=fopen(s,"rb"))==NULL)
printf("does not exist.");
```

```
      else
{
  printf("\nDestination file:");
  gets(s);
  for(i=0;i<strlen(s);i++)
    if(isspace(s[i]))
      s[i]='\0';
  if((plain=fopen(s,"w"))==NULL)
    {
      printf("Error opening destination file");
      close(cipher);
    }
  else
    {
      memset(c,'\0',sizeof(c));
      while((mlen=fread(c,sizeof(char),BUFSIZ-8,cipher))>0)
{
  decrypt(findex,cfindex,c,len,mlen);
  for(i=8;i<mlen;i++)
    printf("%c",c[i]);
  if(fwrite(&c[8],sizeof(char),mlen-8,plain)<0)
    printf("\nError writing file");
  memset(c,'\0',sizeof(c));
}
      fclose(plain);
      fclose(cipher);
    }
}
    }
  else
    noalg();
}
      if(!strncmp(select,"define",6))
{
  printf("\nNumber of layers (max %d):",MAX_LAYERS);
  scanf("%d",&len);
  if (len>MAX_LAYERS)
    len=MAX_LAYERS;
  i=getchar();
  /*initialise the array of the cryptographic algorithm */
  for (i=0;i<=MAX_LAYERS;i++)
    {
      pos=cfindex[i]=0;
      for (j=0;j<=MAX_HEIGHT;j++)
findex[j][i]=0;
    }
  for(i=1;i<=len;i++)
    {
      list();
      printf("Layer %d:",i);
      gets(s);
      j=atoi(s);
      if(j<1 || j>NUM_CFUNCT)
j=1;
      cfindex[i]=j;
      pos=1;
      for(j=1;j<strlen(s);j++)
{
  if(isspace(s[j]) && pos < MAX_HEIGHT)
    {
      dc=atoi(&s[j]);
      if(dc<1 || dc>NUM_FUNCT)
dc=1;
      findex[pos++][i]=dc;
    }
}
    }
  display(findex,cfindex,len);
}
      if(!strncmp(select,"list",4))
{
  list();
}
      if(!strncmp(select,"edit",4))
{
  if(cfindex[1])
    {
      printf("\nEdit layer:");
      gets(s);
      i=atoi(s);
      if(i>0 && i<=len && s[0]!='+')
{
  printf("Old layer=%d ",cfindex[i]);
  for(j=1;j<MAX_HEIGHT;j++)
    if(findex[j][i]!=0)
      printf("%d ",findex[j][i]);
  printf("\nNew layer=");
  gets(s);
  if(strlen(s))
    {
      for(j=1;j<MAX_HEIGHT;j++)
findex[j][i]=0;
      j=atoi(s);
      if(j<1 || j>NUM_CFUNCT)
j=1;
      cfindex[i]=j;
      pos=1;
      for(j=1;j<strlen(s);j++)
{
  if(isspace(s[j]) && pos < MAX_HEIGHT)
    {
      dc=atoi(&s[j]);
      if(dc<1 || dc>NUM_FUNCT)
dc=1;
      findex[pos++][i]=dc;
    }
}
    }
}
      else
{
  if(s[0]=='+')
    {
      i=atoi(&s[1]);
      if(i>0 && i<=len+1 && len<MAX_LAYERS)
{
  if(i<=len)
    for(k=len;k>=i;k--)
      {
cfindex[k+1]=cfindex[k];
for(j=1;j<MAX_HEIGHT;j++)
  findex[j][k+1]=findex[j][k];
      }
  printf("\nNew layer=");
  gets(s);
  if(strlen(s))
    {
      for(j=1;j<MAX_HEIGHT;j++)
findex[j][i]=0;
      j=atoi(s);
      if(j<1 || j>NUM_CFUNCT)
j=1;
      cfindex[i]=j;
      pos=1;
      for(j=1;j<strlen(s);j++)
{
  if(isspace(s[j]) && pos < MAX_HEIGHT)
    {
      dc=atoi(&s[j]);
      if(dc<1 || dc>NUM_FUNCT)
dc=1;
      findex[pos++][i]=dc;
    }
}
      len++;
    }
  else
    {
      s[0]='-';
      s[1]=48+(int)i/10;
      s[2]=48+i%10;
      s[3]='\0';
      len++;
    }
}
    }
  if(s[0]=='-')
    {
      i=atoi(&s[1]);
      if(i>0 && i<=len)
{
  for(k=i;k<len;k++)
    {
      cfindex[k]=cfindex[k+1];
      for(j=1;j<MAX_HEIGHT;j++)
findex[j][k]=findex[j][k+1];
    }
  len--;
  if(len==0)
    cfindex[1]=0;
}
    }
}
    }
  else
    noalg();
}
      if(!strncmp(select,"speed",5))
{
  if(cfindex[1])
    speed(findex,cfindex,len);
  else
    noalg();
}
      if(!strncmp(select,"test",4))
{
  if(cfindex[1])
    test(findex,cfindex,len);
  else
    noalg();
}
      if(!strncmp(select,"encrypt",7))
```

```
{
  if(cfindex[1])
    {
      memset(c,'\0',sizeof(c));
      strcat(c,key);
      pos=8;
      if(strlen(select)>7)
{
  strcat(c,&select[8]);
  pos=strlen(c)+1;
}
       else
{
  printf("\nInput:");
  do
    {
       c[pos]=getchar();
     }while(c[pos++]!='\n');
}
       if((pos-1)%MSG==0)
 c[pos++]=' ';
       j=0;
       for(i=pos-1;i<MSG*(((pos-1)%MSG==0) ? (pos-1)/MSG : (pos-1)/MSG+1);i++,j++)
 c[i]=' ';
       mlen=pos+j-1;
       encrypt(findex,cfindex,c,len,&mlen);
       printf("\nEncrypted:");
       for(i=8;i<mlen;i++)
 printf("%c",c[i]);
       printf("\n");
       for(i=8;i<mlen;i++)
 bin8(c[i]);
      }
    else
      noalg();
  }
        if(!strncmp(select,"run",3))
  {
    if(cfindex[1])
      {
        printf("Type \"exit\" to return to command mode.");
        memset(c,'\0',sizeof(c));
        strcat(c,key);
        pos=8;
        printf("\nInput:");
        do
  {
    do
       {
         c[pos]=getchar();
       }while(c[pos++]!='\n');
     if(!strncmp(&c[8],"exit",4))
       break;
     if((pos-1)%MSG==0)
       c[pos++]=' ';
     j=0;
     for(i=pos-1;i<MSG*(((pos-1)%MSG==0) ? (pos-1)/MSG : (pos-1)/MSG+1);i++,j++)
       c[i]=' ';
     mlen=pos+j-1;
     encrypt(findex,cfindex,c,len,&mlen);
     printf("\nEncrypted:%s\n",&c[8]);
     for(i=8;i<mlen;i++)
       bin8(c[i]);
     decrypt(findex,cfindex,c,len,mlen);
     printf("\nDecrypted:");
     for(i=8;i<mlen;i++)
       printf("%c",c[i]);
     printf("\n");
     for(i=8;i<mlen;i++)
       bin8(c[i]);
     memset(c,'\0',sizeof(c));
     strcat(c,key);
     pos=8;
     printf("\nInput:");
    }while(1);
        }
      else
        noalg();
    }
          if(!strncmp(select,"save",4))
    {
      if(cfindex[1])
        {
          if((j=strlen(select))>4)
    {
      s=&select[5];
      for(i=0;i<strlen(s);i++)
        if(isspace(s[i]))
          s[i]='\0';
    }
          else
    {
      printf("Filename:");
      gets(s);
      for(i=0;i<strlen(s);i++)
        if(isspace(s[i]))
      s[i]='\0';
}
      if((plain=fopen(s,"w"))==NULL)
printf("Error opening file.");
      else
{
  fprintf(plain,"ABSENT cryptographic algorithm structure\n");
  for(i=1;i<=len;i++)
    {
      fprintf(plain,"%d ",cfindex[i]);
      for(j=1;j<MAX_HEIGHT;j++)
if(findex[j][i] != 0 )
  fprintf(plain,"%d ",findex[j][i]);
      fprintf(plain,"0\n");
     }
   fclose(plain);
   printf("Wrote file %s.",s);
 }
     }
   else
     noalg();
 }
       if(!strncmp(select,"load",4))
 {
   if((j=strlen(select))>4)
     {
       s=&select[5];
       for(i=0;i<strlen(s);i++)
 if(isspace(s[i]))
   s[i]='\0';
     }
   else
     {
       printf("Filename:");
       gets(s);
       for(i=0;i<strlen(s);i++)
  if(isspace(s[i]))
    s[i]='\0';
      }
    load(findex,cfindex,&len,s);
  }
        if(!strncmp(select,"script",6))
  {
    if((j=strlen(select))>6)
      {
        s=&select[7];
        for(i=0;i<strlen(s);i++)
  if(isspace(s[i]))
    s[i]='\0';
      }
    else
      {
         printf("Filename:");
         gets(s);
         for(i=0;i<strlen(s);i++)
   if(isspace(s[i]))
     s[i]='\0';
       }
     if((plain=fopen(s,"r"))==NULL)
       printf("Error opening file.");
     else
       {
         scriptflag=1;
         while(fgets(s,128,plain)!=NULL)
   {
     printf("%s",s);
     if(!strncmp(s,"outfile=",8))
       {
         if(outfileflag)           /* close any previous output, and open a new one */
   fclose(outfile);
         /* open write file */
         for(i=8;i<strlen(s);i++)
   if(!isspace(s[i]))
      break;
          ss=&s[i];
          ss[strlen(s)-9]='\0';
          if((outfile=fopen(ss,"w"))==NULL)
    printf("Error opening output file.");
          else
    outfileflag=1;
        }
      if(!strncmp(s,"matlab=",7))
        {
          for(i=7;i<strlen(s);i++)
    if(!isspace(s[i]))
      break;
          ss=&s[i];
          matfile[0]='\0';
          if(strlen(ss)>0)
    strcat(matfile,ss);
           for(i=1;i<strlen(matfile);i++)
    if(isspace(matfile[i]))
      break;
           matfile[i]='\0';
         }
       if(!strncmp(s,"repeat=",7))
      {
        /* set neighbour params */
        repeatcount=atoi(&s[7]);
        ndc=1;
        original_len=len;
        filepos=ftell(plain);
      }
    if(!strncmp(s,"next",4))
      {
        if(repeatcount)
  {
    if(strlen(matfile)>0)
      {
        ss=ssp;
        memset(ss,'\0',sizeof(ss));
        ss=lltostr((long long)ndc,ss);
        if(ndc>100) /*remove three digits */
    matfile[strlen(matfile)-3]='\0';
        if(ndc>10 && ndc<=100) /*remove two digits */
    matfile[strlen(matfile)-2]='\0';
        if(ndc>1 && ndc<=10) /*remove one digit */
    matfile[strlen(matfile)-1]='\0';
        ndc++;
        strcat(matfile,ss);
      }
    repeatcount--;
    fseek(plain,filepos,SEEK_SET);
    fflush(outfile);
    continue;
  }
      }
    if(!strncmp(s,"random",6))
      {
        cag(findex,cfindex,&len);
        display(findex,cfindex,len);
        if(outfileflag)
  fdisplay(findex,cfindex,len);
      }
    if(!strncmp(s,"load ",5))
      {
        for(i=5;i<strlen(s);i++)
  if(isspace(s[i]))
    s[i]='\0';
        i=load(findex,cfindex,&len,&s[5]);
        if(outfileflag && i==0)
  {
    fprintf(outfile,"\"%s\"\n",&s[5]);
    fdisplay(findex,cfindex,len);
  }
      }
    if(!strncmp(s,"seed=",5))
      {
        for(i=5;i<strlen(s);i++)
  if(!isspace(s[i]))
    break;
        i=(int)(s[i]);
        cagseed(findex,cfindex,&len,i);
        display(findex,cfindex,len);
        if(outfileflag)
  fdisplay(findex,cfindex,len);
      }
    if(!strncmp(s,"edit",4))
      {
        if(cfindex[1] || s[5]=='+')
  {
    i=atoi(&s[4]);
    if(i>0 && i<=len && s[5]!='+')
      {
        ss=&s[7];
        if(strlen(ss))
  {
    for(j=1;j<MAX_HEIGHT;j++)
      findex[j][i]=0;
    j=atoi(ss);
    if(j<1 || j>NUM_CFUNCT)
      j=1;
    cfindex[i]=j;
    pos=1;
    for(j=1;j<strlen(ss);j++)
      {
        if(isspace(ss[j]) && pos < MAX_HEIGHT)
  {
    dc=atoi(&ss[j]);
    if(dc<1 || dc>NUM_FUNCT)
      dc=1;
    findex[pos++][i]=dc;
  }
      }
  }
      }
    else
      {
        if(s[5]=='+')
  {
    i=atoi(&s[6]);
    if(i>0 && i<=len+1 && len<MAX_LAYERS)
    {
      if(i<=len)
for(k=len;k>=i;k--)
  {
    cfindex[k+1]=cfindex[k];
    for(j=1;j<MAX_HEIGHT;j++)
      findex[j][k+1]=findex[j][k];
  }
      ss=&s[7];
      if(strlen(ss))
 {
  for(j=1;j<MAX_HEIGHT;j++)
    findex[j][i]=0;
  j=atoi(ss);
  if(j<1 || j>NUM_CFUNCT)
    j=1;
  cfindex[i]=j;
  pos=1;
  for(j=1;j<strlen(ss);j++)
    {
      if(isspace(s[j]) && pos < MAX_HEIGHT)
 {
   dc=atoi(&ss[j]);
   if(dc<1 || dc>NUM_FUNCT)
     dc=1;
   findex[pos++][i]=dc;
 }
     }
   len++;
  }
       else
  {          /* remove extra layer if */
    s[5]='-';  /* input is invalid */
    s[6]=48+(int)i/10;
    s[7]=48+i%10;
    s[8]='\0';
    len++;
  }
      }
  }
        if(s[5]=='-')        /* remove a layer */
   {
     i=atoi(&s[6]);
     if(i>0 && i<=len)
       {
         for(k=i;k<len;k++)
   {
     cfindex[k]=cfindex[k+1];
     for(j=1;j<MAX_HEIGHT;j++)
       findex[j][k]=findex[j][k+1];
   }
         len--;
         if(len==0)
   cfindex[1]=0;
       }
   }
       }
   }
       }
     if(!strncmp(s,"speed",5))
       {
         speed(findex,cfindex,len);
       }
     if(!strncmp(s,"key=",4))
        {
          ss=&s[4];
          if(strlen(ss))
    for(i=0;i<8;i++)
      if(i<strlen(ss))
        key[i]=ss[i];
      else
        key[i]=' ';
          if(outfileflag)
    fprintf(outfile,"key=%s\n",key);
        }
      if(!strncmp(s,"keybin=",7))
        {
          pos=0;
          ss=&s[7];
          if(strlen(ss))
     {
       for(i=0;i<8;i++)
         key[i]=0;
       for(i=0;i<8;i++)
         for(j=7;j>=0;j--)
           if((pos<strlen(ss)) && (ss[pos]=='1') && (!isspace(ss[pos])))
     {
       key[i]+=(0x01 << j);
       pos++;
     }
           else
     {
       while(isspace(ss[pos]))
         pos++;
       pos++;
     }
```

```
}
      if(outfileflag)
{
  fprintf(outfile,"key=");
  for(i=0;i<8;i++)
    fbin8(key[i]);

  fprintf(outfile,"\n");
}
    }
  if(!strncmp(s,"loops=",6))
    {
      ni=atoi(&s[6]);
    }
  if(!strncmp(s,"input=",6))
    {
      if(!strncmp(&s[6]," lin ",5) || !strncmp(&s[6],"lin ",4))
{
  inp=1;
  if(strlen(s)>10)
    nb=atoi(&s[9]);
  if(nb>=0 && nb<64)
    inp=2;
}
      if(!strncmp(&s[6]," linct ",7) || !strncmp(&s[6],"linct ",6))
{
  inp=4;
  nb=0;
  nb=atoi(&s[11]);
}
      if(!strncmp(&s[6]," rand",5) ||  !strncmp(&s[6],"rand",4) )
{
  inp=3;
}
      if(!strncmp(&s[6]," struct",7) ||  !strncmp(&s[6],"struct",6) )
{
  inp=5;
}
      if(!strncmp(&s[6]," key",4) ||  !strncmp(&s[6],"key",3) )
{
  inp=2;
}
    }
  if(!strncmp(s,"block",5))
    {
      sel=3;
      if(inp!=2)        /* select plaintext as a default variable */
inp=1;          /* unless key is selected*/
      test(findex,cfindex,len);
    }
  if(!strncmp(s,"freq",4))
    {
      sel=1;
      incf=1;
      if(outfileflag)
{
  switch(inp)
    {
    case 1:
      fprintf(outfile,"Input:linear Increase:%d Loops:%d ",incf,ni);
      break;
    case 2:
      fprintf(outfile,"Input:linear byte %d Increase:%d Loops:%d ",nb,incf,ni);
      break;
    case 3:
      fprintf(outfile,"Input:random Loops:%d ",ni);
      break;
    case 4:
      fprintf(outfile,"Input:constant&linear byte %dIncrease:%d Loops:%d ",nb,incf,ni);
      break;
    case 5:
      fprintf(outfile,"Input:structured ");
      break;
    }
}
      test(findex,cfindex,len);
    }
  if(!strncmp(s,"serial",6))
    {
      sel=2;
      incf=1;
      if(outfileflag)
{
  switch(inp)
    {
    case 1:
      fprintf(outfile,"Input:linear Increase:%d Loops:%d ",incf,ni);
      break;
    case 2:
      fprintf(outfile,"Input:linear byte %d Increase:%d Loops:%d ",nb,incf,ni);
      break;
    case 3:
      fprintf(outfile,"Input:random Loops:%d ",ni);
      break;
    case 4:
      fprintf(outfile,"Input:constant&linear byte %dIncrease:%d Loops:%d ",nb,incf,ni);
```

```
        break;
      case 5:
        fprintf(outfile,"Input:structured ");
        break;
      }
}
        test(findex,cfindex,len);
      }
    if(!strncmp(s,"autocor",7))
      {
        sel=4;
        test(findex,cfindex,len);

      }
    if(!strncmp(s,"diffusion",9))
      {
        sel=5;
        test(findex,cfindex,len);
      }
    if(!strncmp(s,"margdif",7))
      {
        sel=7;
        if(inp!=2)          /* select plaintext as a default variable */
inp=1;            /* unless key is selected*/
        test(findex,cfindex,len);
      }
    if(!strncmp(s,"depth",5))
      {
        sel=6;
        if(inp!=2)          /* select plaintext as a default variable */
inp=1;            /* unless key is selected*/
        test(findex,cfindex,len);
      }
    if(!strncmp(s,"zero",4))
      {
        sel=8;
        if(inp!=2)          /* select plaintext as a default variable */
inp=1;            /* unless key is selected*/
        test(findex,cfindex,len);
      }
}
        fclose(plain);
        if(outfileflag)
{
  fclose(outfile);
  outfileflag=0;
}
        scriptflag=0;
      }
}
      }while(strncmp(select,"quit",4));
}
```

D.12 DES implementation - README

```
                libdes, Version 4.01 13-Jan-97

                Copyright (c) 1997, Eric Young
                        All rights reserved.

    This program is free software; you can redistribute it and/or modify
    it under the terms specified in COPYRIGHT.

--
The primary ftp site for this library is
ftp://ftp.psy.uq.oz.au/pub/Crypto/DES/libdes-x.xx.tar.gz
libdes is now also shipped with SSLeay.  Primary ftp site of
ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL/SSLeay-x.x.x.tar.gz

The most recent version can be found at
ftp://ftp.psy.uq.oz.au/pub/Crypto/DES/libdes.tar.gz

The best way to build this library is to build it as part of SSLeay.

This kit builds a DES encryption library and a DES encryption program.
It supports ecb, cbc, ofb, cfb, triple ecb, triple cbc, triple ofb,
triple cfb, desx, and MIT's pcbc encryption modes and also has a fast
implementation of crypt(3).
It contains support routines to read keys from a terminal,
generate a random key, generate a key from an arbitrary length string,
read/write encrypted data from/to a file descriptor.

The implementation was written so as to conform with the manual entry
for the des_crypt(3) library routines from MIT's project Athena.

destest should be run after compilation to test the des routines.
rpw should be run after compilation to test the read password routines.
The des program is a replacement for the sun des command.  I believe it
conforms to the sun version.

The Imakefile is setup for use in the kerberos distribution.
```

```
These routines are best compiled with gcc or any other good
optimising compiler.
Just turn you optimiser up to the highest settings and run destest
after the build to make sure everything works.

I believe these routines are close to the fastest and most portable DES
routines that use small lookup tables (4.5k) that are publicly available.
The fcrypt routine is faster than ufc's fcrypt (when compiling with
gcc2 -O2) on the sparc 2 (1410 vs 1270) but is not so good on other machines
(on a sun3/260 168 vs 336).  It is a function of CPU on chip cache size.
[ 10-Jan-97 and a function of an incorrect speed testing program in
  ufc which gave much better test figures that reality ].

It is worth noting that on sparc and Alpha CPUs, performance of the DES
library can vary by upto %10 due to the positioning of files after application
linkage.

Eric Young (eay@mincom.oz.au)
```

D.13 rmd128.c header

```
/********************************************************************\
 *
 *      FILE:     rmd128.c
 *
 *      CONTENTS: A sample C-implementation of the RIPEMD-128
 *                hash-function. This function is a plug-in substitute
 *                for RIPEMD. A 160-bit hash result is obtained using
 *                RIPEMD-160.
 *      TARGET:   any computer with an ANSI C compiler
 *
 *      AUTHOR:   Antoon Bosselaers, ESAT-COSIC
 *      DATE:     1 March 1996
 *      VERSION:  1.0
 *
 *      Copyright (c) Katholieke Universiteit Leuven
 *      1996, All Rights Reserved
 *
\********************************************************************/
```