# The Design of a Secure Data Communication System

By

***Moutasem Shafa'amry*** *B.Eng., M.Sc.*

A Dissertation Presented in Fulfilment
of the Requirements for the Ph.D. Degree.

Dublin City University

**Supervisor**

Dr. Michael Scott

School of Computer Applications

February 1994

# Declaration

*I herby certify that this material, which I now submit for assessment on the programme of study leading to the award of Ph.D degree in Computer Science is entirely my own work and has not been taken from the work of others save and to extent that such work has been cited and acknowledged within the text of my work.*

Signed: ..... Date: 25/11/1994.

*Moutasem Shafa'amry*

# Acknowledgements

*I would like to express my heartfelt gratitude to Dr. Michael Scott whose help, supervision and guidance were invaluable during my period of study.*

*Sincere thanks are expressed to Andrew Mc Carren and Gary Keogh for all their kind help and assistance.*

*I would also like to thank my fellow postgraduate students at the School of Computer Applications for their kindness, encouragement and patience in answering all my questions, and for their assistance in proof-reading the text which has improved my English.*

*Especial thanks to my sincere friend Abdul-Gani Olabi whose tireless encouragement helped me to complete my study.*

*I would also like to thank the School of Computer Applications for its financial support. Many thanks are also expressed to the management board of the Scientific Studies and Research Centre for their help and encouragement.*

# The Design of a Secure Data Communication System

*Moutasem Shafa'amry* B.Eng., M.Sc.

## Abstract

The recent results of using a new type of chosen-plaintext attack, which is called *differential cryptanalysis*, makes most published conventional secret-key block cipher systems vulnerable. The need for a new conventional cipher which resists all known attacks was the main inspiration of this work.

The design of a secret-key block cipher algorithm called *DCU-Cipher*, that resists all known cryptanalysis methods is proposed in this dissertation. The proposed method is workable for either 64-bit plaintext/64-bit ciphertext blocks, or 128-bit plaintext/128-bit ciphertext blocks. The secret key in both styles is 128-bit long. This method has only four rounds and the main transformation function in this cipher algorithm is based on four mixed operations. The proposed method is suitable for both hardware and software implementation. It is also suitable for cryptographic hash function implementations.

Two techniques for file and/or data communication encryption are also proposed here. These modes are modified versions of the Cipher-Block Chaining mode, by which the threat of the known-plaintext differential cyptanalytical attack is averted.

An intensive investigation of the best known Identity-based key exchange schemes is also presented. The idea behind using such protocols, is providing an authenticated secret-key by using the users identification tockens. These kind of protocols appeared recently and are not standardized as yet. None of these protocols have been compared with previous proposals. Therefore one can not realize the efficiency and the advantages of a new proposed protocol without comparing it with other existing schemes of the same type. The aim of this investigation is to clarify the advantages and the disadvantages of each of the best known schemes and compare these schemes from the complixity and the speed viewpoint.

# *Table of Contents*

# *Chapter 1*

# Introduction

Although the need to keep certain messages secret has been appreciated for thousands of years, it is only recently that information security has become commercially important and thereby widely recognized as a necessity. Until the end of the second world war, military and diplomatic communications were the only major application areas for cryptographic techniques. The vast development in electronic data processing and telecommunications, leading to computer networks of ever-growing size, results in an increasing vulnerability of these systems to various attacks. The potential damage that can caused by such an attack is often tremendous, which explains the recent commercial interest in protecting information systems. No prophetic skills are required to foresee a dramatic growth in the need for cryptographic techniques in the near future.

Cryptography is today understood to be the science of secure communications or, more generally, of information security. However, it was not until 1949, when Shannon published his paper titled "Communication theory of secrecy systems", that cryptology (including both cryptography and cryptanalysis) deserved the attribute of a science. To protect information from unauthorized disclosure is only one of the goals of cryptography. Other goals are to ensure the integrity and authenticity of messages, and the identification of persons or computer systems.

This dissertation is concerned mainly with the problem of protecting information using a single secret-key cipher system. This research was motivated by the new type of cryptographic attack which has been proposed by Biham and Shamir and called *differential cryptanalysis* [BS91] [BS92a] BS92b], to which most of the published conventional block cipher systems have been subjected including the standard one, *DES (Data Encryption Standard).* DES has been adopted by *NBS (National Bureau of Standards)* and recommended by more than one standard-making organization, such as *ANSI (The American National Standards Institute), ISO (The International Organization for Standardization)* and *ABA (The American Bankers Association)* [SB92], and it was the only conventional cryptographic algorithm endorsed by the U.S. government until the very recent advent of the *Clipper* system [NEWS1]. Federal agencies are required to use DES for protection of unclassified data, but the private sector has adopted DES as well because government endorsement implies an approved degree of security. Attacking this widely used cipher algorithm puts all these systems in jeopardy.

Several other secret-key cryptosystems were proposed during the last few years as replacements of DES. Most of these published cipher algorithms have been successfully attacked by the differential cryptanalytic method.

Thus, a new secret-key block cipher algorithm is presented in this work which is resistant all known types of attack including differential cryptanalysis.
Differential cryptanalytic attack is considered as a chosen-plaintext type of attack, but it can be converted into a known-plaintext attack. Having enough plaintext/ciphertext pairs, the differential cryptanalytical method is able to attack long messages which are encrypted using a block cipher and chained by the standard mode of operation, the *CBC (Cipher Block Chaining).*

Therefore, two new modes of operation for a block cipher are proposed here immune to the threat of a known-plaintext differential cryptanalytical attack.

This dissertation is organized as follows:

In *Chapter 2*, the definition of main cryptographic notations are introduced as well as the basic components that are involved in building a conventional cipher system. Most of the published conventional cipher algorithms are investigated in depth in this chapter highlighting the need for a new conventional cipher system that overcomes all the weakness of the previous methods. The problem of exchanging a secret key between users is also addressed here, and it is shown how public-key cipher systems partially solve this problem. An intensive investigation of the best known Identity-based key exchange schemes which base their security on mathematically hard problems is presented in this chapter. The idea behind using such protocols is to provide an authenticated secret key by using the users identification tokens (numbers). These kind of protocols appeared recently and are not standardized as yet. Many protocols have been proposed during the last few years. None of these protocols has been compared with other existing one of the same type. Therefore one can hardly realize the efficiency and the advantages of a new identity-based key exchange protocol without comparing it with other existing schemes of the same type. The aim of the investigation of the best-known of these protocols is to clarify the relationships that link a user identification, his/her public information with his/her secret key in each scheme and compare these schemes from the complexity and speed viewpoint.

*Chapter 3*, discusses the different types of cryptographic attack .

The design of our new secret-key block cipher is explained in *Chapter 4*. The characteristics of this block cipher algorithm makes it a good candidate to be used in building a strong, collision-free hash function. Two new modes of operation are also proposed in this chapter.

In *Chapter 5*, the implementation of the new cipher system is illustrated. The results of some statistical tests which are implemented on our new block cipher algorithm are discussed.

The final chapter, *Chapter 6*, contains the concluding remarks.

Finally, it is worth mentioning here the most recent developments in this area of cryptography. On April 1993 the *White house* announced a new encryption technology, called *the Clipper Chip*, for securing the telephone communications. This state-of-the art microcircuit has been developed by government engineers. As R. Kammer, the acting director of *NITS (National Institute of Standards and Technology)* stated in [NEWS1] "The chip represents a new approach to encryption technology. It can be used in new, relatively inexpensive encryption devices that can be attached to an ordinary telephone. It scrambles telephone communications using an encryption algorithm that is more powerful than many commercial use today. The Clipper algorithm with 80 bit long cryptographic key is approximately *16 million* times stronger than DES".

Each Clipper chip contains, the encryption algorithm, classified control software, a device identification number, a family key used by law enforcement, and a device unique key that unlocks the session key used to encrypt a particular communication.

The new system contains also the following:

- A secure facility for generating a device unique keys and programming the devices with the classified algorithms, identifiers, and keys.

- Two escrow agents that each hold a component of every device unique key. When combined, those two components form the device unique key.

- A law enforcement access field (LEAF), which enables an authorized law enforcement official to recover the session key. The LEAF is created by a device at the start of an encrypted communication and contains the session key encrypted under the device unique key together with the device identifier, all encrypted under the family key.

- LEAF decoders that allow an authorized law enforcement official to extract

the device identifier and encrypted session key from an intercepted LEAF. The identifier is then sent to the escrow agents, who return the components of the corresponding device unique key. Once obtained, the components are used to reconstruct the device unique key, which is then used to decrypt the session key.

The Clipper encryption algorithm which is called *SKIPJACK*, is classified *secret not releasable to foreign nationals*. Therefore, there is no structural details available about this new cryptographic algorithm. The only known information about the *SKIPJACK* algorithm is that, it is a 64-bit algorithm that transforms a 64-bit input block into a 64-bit output block under control of 80-bit secret key. It involves performing 32 iterations of a complex, non-linear function.

*Chapter 2*

# Cryptographic Algorithms and Key Exchange Protocols

## 2.1 Cryptographic Algorithms

The basic problem in cryptography is devising procedures to transform sequences of messages (*plaintexts*) into sequences of apparently random data (*Ciphertexts*) that can withstand intense cryptanalysis. The procedures used to accomplish such transformations involved either *code systems* (systems that require a code book or dictionary to translate words), or *cipher systems*. Cipher systems require two basic elements: a *cryptography algorithm*, a procedure, or set of rules or steps that are constant in nature, and a set of variable *cryptographic keys*, a secret sequence of numbers or characters selected by the user.

The transformation of plaintext into ciphertext is known as *encipherment* or encryption. Each transformation must have a unique inverse operation, also identified by a cryptographic key. The inverse transformation from ciphertext to plaintext is called *decipherment* or decryption.

The procedure that involves both enciphering and deciphering operations is called the *cipher* procedure.

Shannon [Den82] described theoretically the possibility of designing

*unbreakable* ciphers by selecting the key randomly, and using that key only once. However, the length of the key must be equal or greater than the length of the plaintext to be enciphered. That means a large number of long keys must be transferred between the communicators and stored, before communication can be established. This makes the idea impractical.

The alternative solution is to design a pragmatic *strong* cryptographic algorithm, which in theory can always be broken, but in the practical sense it cannot. There are two ways to design a *strong* cryptographic algorithm. First, one can study the possible methods of solution available to the cryptanalyst (see *chapter 4*) and then define a set of design rules that thwart all of these methods. An algorithm is then constructed which can resist these general methods of solutions. Second, one can construct an algorithm in such a way that breaking it requires the solution of some known problem, but one that difficult to solve. The cryptographic algorithms which are designed based on the first method are called *conventional* (or sometimes *symmetric*), and the cryptographic algorithms that follow the second method in their design are known as *public-key* (or *asymmetric*) cryptographic algorithms. With a conventional cryptographic algorithm, the same key is used for enciphering and deciphering, while in the public-key cryptographic algorithm, the deciphering and enciphering keys are different in such a way that at least one key is computationally infeasible to determine from the other.

Therefore, the design of a strong cryptographic algorithm must satisfy the following conditions:

1. The mathematical equations describing the algorithm's operation are so complex that, for all practical purposes, it is not possible to solve them using analytical methods.
2. The cost or the time required to recover the message or the key is too great when using methods that are mathematically less complicated, because either too many computational steps are required, or too much data storage is required.

There are two main types of ciphers: *stream Cipher* and *block cipher*. In the stream cipher, a bit-stream generator produces a stream of binary digits (key-stream), which

is then combined either with plaintext (via an operation *OP*) to produce ciphertext, or with a ciphertext (via the inverse of *OP*) to recover plaintext. This type of cipher is beyond the scope this work.

In the block cipher the plaintext is partitioned into fixed length blocks. A block cipher transforms a block of input bits of *fixed* length into a block of output bits of *fixed* length under a *fixed* length of a user-selected key.

### 2.1.1 Conventional Block Cipher Algorithms

The two basic components of conventional cipher techniques are *transposition* (permutation or diffusion), and *substitution* (confusion). In substitution, letters, (or bits) are replaced by other letters (or bits), whilst in transposition, letters (or bits) are arranged in a different order.

Many ciphers which have used one of these techniques alone, such as *Vigenére* cipher, *Nihilist*, the *Jefferson Cylinders* and others [DP84] were very weak. As was pointed out by Shannon, cipher operations which are weak in themselves can be combined together to form something much stronger, this is the concept of the *product* cipher which has been widely followed in the design of modern conventional block cipher systems.

In the early 1970s IBM[1] designed a substitution/permutation network cryptographic algorithm called *Lucifer* [Den82]. In *Lucifer* the input of the substitution tables is the bit permuted output of the substitution tables of the previous round. The input of the substitution tables of the first round is the plaintext itself. A key bit is used to choose the actual substitution table at each entry out of two possible four-bit to four-bit invertible substitution tables. The *Lucifer* block size was 128-bits, with no data expansion in the encipherment process, and the key size was also 128-bits long.

Later on, in mid-1970s another algorithm was proposed by IBM, which has been adopted by NBS (*National Bureau of standards)*, called the *Data Encryption Standard DES*. It is an improved version of *Lucifer* and the building blocks of this

[1] *IBM: is a trademark for International Business Machines Co.*

algorithm are permutations, substitutions and binary addition (X-OR). Permutations in the *DES* are of three kinds, straight, expanded and permuted choices. *DES* enciphers 64-bit blocks of data with a 56-bit key. The input Block is first transposed under an initial permutation *IP*. After it has passed through 16 iterations of a function *F*, it is transposed under the inverse permutation $IP^{-1}$ to give the final result. Between the initial and the final transpositions, the algorithm performs 16 iterations of a function *F* that combines substitution and transposition. Substitutions in the DES are known as *S-boxes* and are specified by eight different tables. Each of these *S-box* has 6-bit input and 4-bit output. As shown in the diagrammatic representation of the DES in figure 2-1, the plaintext block splits into two equal parts (32-bit each) after passing though the initial permutation *IP*. The symbols *L* and *R* refer to the left and right part respectively. Expressing that mathematically:

Let the subscript $_j$ refer to the various rounds, and let $R_j$ and $L_j$ refer to the right and left sub-block after the *jth* round. The sub-key at the *jth* round, which is generated from a 64-bit user selected key by a key schedule, is $k_j$.
During the enciphering the following relationships are true:

$$L_j = R_{j-1}$$
$$R_j = L_{j-1} \oplus F(R_{j-1},k_j).$$

DES was designed for hardware implementation, and implementing DES in software is inefficient.

Diffie and Hellman argue that with 56-bit keys, DES may be broken under a *known-plaintext* attack or by *exhaustive search* (see *chapter 4*). In 1977 they showed that a hypothetical special-purpose machine consisting of million LSI chips could try all $2^{56}$ keys in one day [SB92]. The cost of that machine would be about $20 million. Amortized over 5 years, the cost per day would be about $10,000. They predicted the cost of building this machine will drop substantially by 1990 ( that prediction was made in the early 1980s). Hellman has also shown that it is possible to speed up the

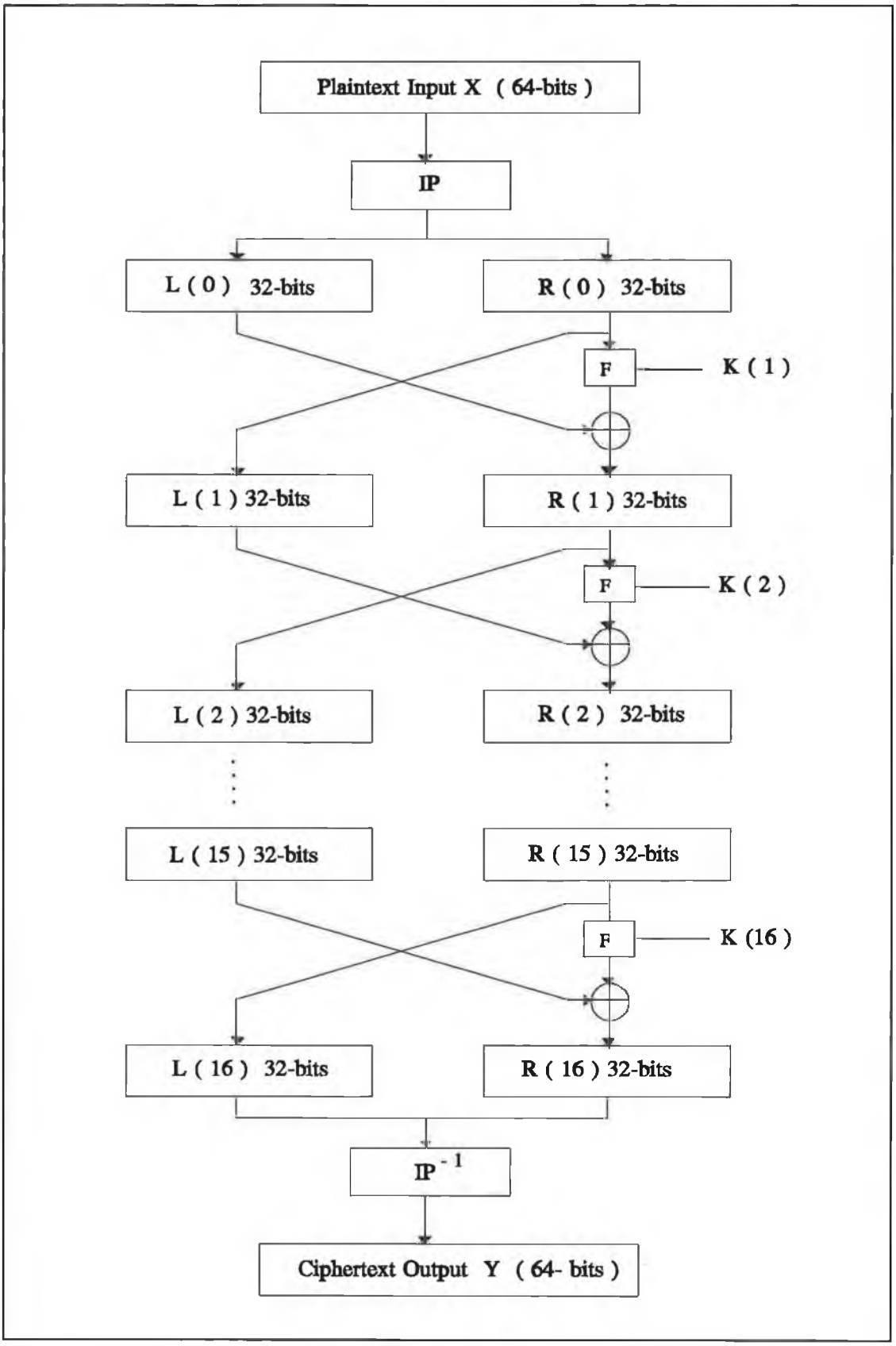



***Figure 2.1** The DES Structure*

searching process by trading time for memory in a *chosen-plaintext* attack. The cost per solution would be $10 on a $5M machine.

Hellman and others argue that the key size should be doubled, but Tuchman claims that the same level of security can be obtained with 56-bit keys, using a multiple encryption scheme invented by Matyas and Meyer [Den82] [DP84].

The criticism of the DES algorithm has also concerned the choice of S-boxes. Hellman, Diffie, Merkle, Scroeppel and others have investigated the S-box structure and have shown that the security of DES-like algorithm can be reduced by careful choice of S-boxes. By replacing the DES S-boxes by others of their own design, they have shown that it is possible to weaken the security of the encipherment while concealing the weak S-boxes structure to some extent. Desmet, Quisquater and Davio [DQD85], evaluated the non-substitution effect of *F* function and the key clustering in DES, and they proved that the F function is not one-to-one for a fixed key.

Despite all these criticisms, DES has been widely used as a secure block cipher algorithm for commercial systems after it has been adopted by NBS and recommended by more than one standard-making organization such as ANSI (*The American National Standard Institute*), ISO (*The International Organization for Standardization*) and ABA(*The American Bankers Association*).

In 1985 Chaum and Evereste [BS91] showed that a *meet in the middle* attack (see chapter 4) can reduce the key search for DES reduced to a small number of rounds by the following factors:

| Number of rounds | Reduction factor |
|---|---|
| 4 | $2^{19}$ |
| 5 | $2^{9}$ |
| 6 | $2^{2}$ |
| 7 | - |

They also showed that a slight modified version of DES reduced to seven rounds can be solved with a reduction factor of 2. However, they proved that a *meet in the middle* attack is not applicable to DES reduced to eight or more rounds.

In 1987 Davies [BS91] described a *known-plaintext* cryptanalytic attack on DES. Given sufficient data, it could yield 16 linear relationships among key bits, thus reducing the size of the subsequent key search to $2^{40}$. The full rounds DES withstood the intense cryptanalysis until 1992, when Biham and Shamir introduced a new attack called *differential cryptanalysis*, by which the full 16-rounds DES was attacked [BS92c].

During the last decade several cryptographic algorithms were suggested as replacement of the original DES. Some researchers have proposed to strengthen DES by making all the sub-keys independent (or at least to derive them in more complicated way from a longer actual key K). *The Generalized DES scheme (GDES)* is an attempt to speed up DES which was suggested by Schaumuller and Bichl [BS91]. The *GDES* blocks are divided into $q$ parts of 32 bits each. The $F$ function is calculated once per round on the right-most part, and the result is X-ORed into all the other parts, which are then cyclically rotated to the right. After the last round the order of the parts is exchanged to make the encryption and the decryption differ only in the order of the sub-keys. The scheme is shown in Figure 2-2 , where $n$ is the number of the rounds in the *GDES* cryptosystem. This cryptographic algorithm was broken by Biham and Shamir [BS91].

In 1987 Shimizu and Miyaguchi [SM88] proposed a conventional block algorithm called *FEAL (Fast Data Encryption Algorithm)*. The intention was a fast software implementation and an avoidance of discussions about random tables. This algorithm acts on 64 bits of plaintext to produce a 64-bit ciphertext controlled by 64-bit key. The two building blocks of this cipher are the exclusive-or and a one byte data transformation $S$ defined by:

$$S(x,y,z) = Rot2((x + y + z) \bmod 256)$$

where $x,y$ are 8-bit numbers, $z$ is a constant of value 0 or 1, and *Rot2* cyclically rotates the bits of its input 2 places. The first and last permutations in *DES* are replaced here by the binary addition of the input plaintext/the final round's output with four 16-bits subkeys, respectively. The first version of *FEAL*, called *FEAL-4*

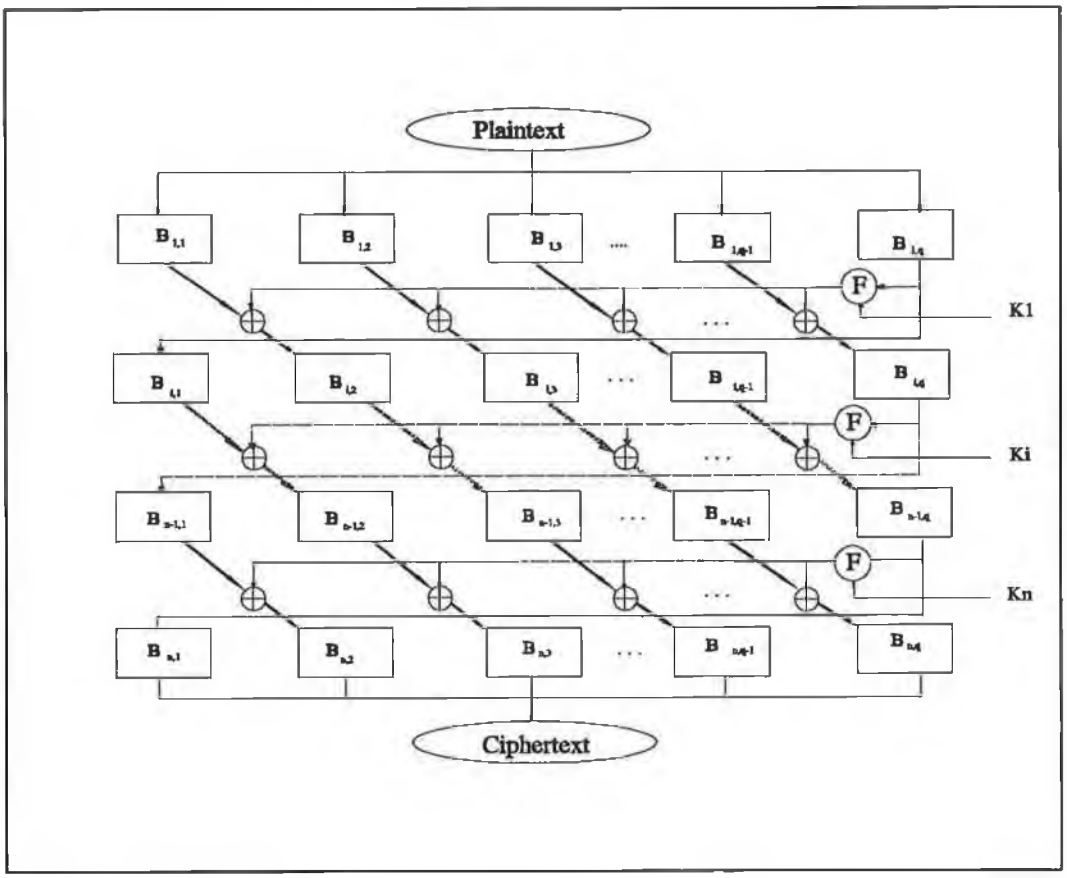



***Figure 2.2*** *The Generalized DES Scheme*

with four rounds, was broken by Den Boer [Boe89] using a *chosen-plaintext* attack with 100-10,000 encryptions. The inventors of *FEAL* reacted by introducing new version, called *FEAL-8*, with eight rounds. Both versions were described as cryptographically better than *DES*. *FEAL-8* was also attacked by Biham and Shamir. The creators modified their method again by increasing the number of the rounds and introduced two new versions, called *FEAL-N* [MKO90] with any even number of rounds, e.g. 16 or 32, and *FEAL-NX* , similar to *FEAL-N* with the extension of the key size to 128 bits. However, Biham and Shamir were able to break the new versions of FEAL using the *differential cryptanalysis* technique.

In 1989, a conventional cryptographic algorithm, called *LOKI* [BPS91a], was proposed. *LOKI* is a 64-bit key/64-bit block cryptosystem similar to *DES* which uses one twelve-bit to eight-bit S-box (based on irreducible polynomials) replicated four

times in each round. The expansion and the permutation are replaced by new choices and the initial and the final transformations are replaced by X-ORs with the key. The bit permutations in the key scheduling are replaced by rotations and the sub-keys become 32-bit long. The X-OR of the input of the *F* function with the key is done before the expansion and therefore neighbouring S-boxes receive common bits. This algorithm was attacked by Biham and Shamir using the *differential cryptanalysis* method. The creators responded by modifying their method to oppose such kind of attack [BPS91b].

Shimizu and Yamakami proposed, in July 1990, a fast 32-bit microprocessor oriented data encryption algorithm [SY90]. The encryption speed of the proposed algorithm is about three to four times the speed of *FEAL-8*. This algorithm is 128-bit plaintext/128-bit ciphertext with 128-bit key. The main functions are used in the structure of the encryption algorithm and its key schedule are exclusive-or, addition and fixed/and variable rotations. There are six steps in this cipher algorithm, the last three of them are the first three steps in the reverse order. The plaintext splits into four 32-bit sub-blocks. In the first step, the first and the second sub-blocks effect the other two sub-blocks by X-ORing with them respectively. In the second step, the last two sub-blocks are transposed using four variable rotations and two addition operations (each sub-block rotates right and left simultaneously, the output of the left rotation is added to the output of the right rotation of the other sub-block). These two transposed sub-blocks then influence the other sub-blocks using X-OR. In the third step, the transposition is carried out on the third and the fourth sub-blocks in similar way as in the second step, but using fixed rotations, then the first and the second sub-blocks are X-Ored with the two resulted sub-blocks respectively. There is no published paper on attacking this method yet.

A new block cipher, called *REDOC-II*, was published in 1990 [CW91]. *REDOC-II* is a high speed confusion/diffusion cryptosystem suggested by Cryptech. REDOC-II has ten-rounds with 70-bit blocks (arranged as ten bytes of seven bits). Each round contains six phases: (1) First variable substitution, (2) Second variable

substitution, (3) First variable key X-OR, (4) Variable enclave, (5) Second variable key X-OR and (6) Variable permutation. This method has also been successfully attacked by Biham and Shamir [BS92b] .

Merkle introduced, in 1990, another conventional block cipher called *Khafre* [Mer91]. *Khafre* is a software oriented cryptosystem with 64-bit blocks whose number of rounds (which should be a multiple of eight) is not yet determined, but the designer expects that almost all the applications will use 16, 24 or 32 rounds. Each block is divided into two halves. In each round the lowest byte of the right half is used as an index to an S-box with 32-bit output. The left half is X-Ored with the output of the S-box, the right half is rotated and two halves are exchanges. The rotation is such that every byte is used once every eight rounds as an input to S box. Before the first round and after every eighth round the data X-Ored with 64-bit sub-keys. These sub-keys are only the way the key is involved in the cryptosystem. In 1991, this algorithm was effectively broken by Biham and Shamir.

Another new conventional block cipher algorithm proposed also in 1990 by Lai and Massey, as a candidate for a new encryption standard [LM91], and called *PES (Proposed Encryption Algorithm). PES* is 8-round algorithm which operates on 64-bit plaintext to generate a ciphertext of 64-bits long, under control of 128-bit key. This method has two main differences in comparison to all the above mentioned algorithms. First, the designers use in fabricating their algorithm three operations from different algebraic groups, namely, bit-by-bit X-OR, addition modulo $2^{16}$ and multiplication modulo $2^{16}+1$ with zero sub-blocks corresponding to $2^{16}$. Second, all the round's input sub-blocks are involved in constructing the *F* function's input within the round, while in other methods, only part of the round's input (half in most of them) is implicated in the *F* function. The method starts by splitting the 64-bit plaintext into four 16-bit input sub-block. A multiplicative operation is then implemented on each of the first two input sub-blocks by a different 16-bit sub-key, while each of the other two input sub-blocks is effected by another different 16-bit sub-keys using additive operation. The first resulting sub-block is X-Ored with the third one, while the second and the

fourth resulting sub-blocks are X-Ored together, generating two of the four F input sub-blocks. The other two *F* inputs are 16-bit sub-keys. Each of the two *F* outputs is then X-Ored with a pair of the round input sub-blocks. Swapping the resulting sub-blocks provides the input sub-blocks for the next round. After the publication of the *differential cryptanalysis* method of attack, the PES designers applied this type of the cryptanalysis on their own method, then modified it to resist such kind of attack [LMM92]. The modifications involved rearranging the operations that are implemented on the round's input sub-blocks and changing the swapping technique of the sub-blocks at the end of each round. This modified algorithm is known as *IPES (Improved Proposed Encryption Standard)*, recently renamed *IDEA (International Data Encryption Algorithm).*

In 1992, Ohtsuka and Taniguchi proposed a conventional cryptographic algorithm called *CALC (A Cipherment Algorithm for C programming Language)* [OT92]. This method has eight rounds and acts on 96-bit plaintext to form a ciphertext block of 96-bit long controlled by 96-bit user selected key. The two building blocks of this cipher are the exclusive-or and a transformation function S defined by:

$$S(x,y) = Rol3(x + y + a) \bmod 2^{16}$$

where *x,y* are 16-bit numbers, *a= 258* a constant value, and *Rol3 (X)* is 3-bits rotation of the X bits. It considered faster than FEAL-8. No attack on this method has been published.

The difficulty of distributing keys has been the major limitation of the use of conventional cryptographic technology, where there was no trusted way to transfer the secret key from one party to other. The first scheme that solved this problem has been proposed by Diffie and Hellman in 1976 [DH76] by using the public-key algorithm's idea for key exchange (See 2.2).

### 2.1.2 Public-Key Cipher Algorithms

The concept of the two-keys cryptosystem was introduced by Diffie and Hellman in 1976 [DH76] to overcome the difficulty of transferring the secret key that

faced the users of the conventional ciphers. They proposed a new method of encryption, called *public-key* encryption, wherein each user has both a *public* and *private* key. Both keys are related mathematically in such away that knowing the public key is insufficient to reveal the secret one in a feasible time. The two users can communicate knowing only each other's public key. Diffie and Hellman suggested applying computational complexity in cryptology where they noted that NP-complete (*Non-deterministic Polynomial*) problems might make excellent candidates, because they cannot be solved in polynomial time by any known techniques. However the security of the Diffie-Hellman scheme is related to the difficulty of computing a discrete logarithm in a finite field *GF(p)*, where *p* is a large prime number which is not known to be NP-complete.

Merkle and Hellman [MH78] developed a public-key encryption algorithm based on an NP (*Non-deterministic Polynomial*) problem called subset-sum or *knapsack* problem. This problem has been explained by Hellman as follows:
Giving a set of numbers $a_1,a_2,.....,a_n$ and the sum *C*, determine which of these numbers add up to *C*. In this public key cryptosystem, the sender converts his messages into a string of binary numbers, then he consults the public key directory to get the receiver's public key which is a vector (set) of ordered numbers

$$A = (a_1,a_2,.....,a_n).$$

The sender then breaks the string of binary numbers that represents his message into a block of *n* bits, and for each block *X* he forms the *dot product* $C = A.X$. The result *C* is the encrypted message which the sender transmits over the insecure channel.
At the receiving side, the receiver has the corresponding secret key vector *S* and the two random numbers *W*, and *m* from which his public key was derived by:

$$A = S.W \bmod m.$$

To decrypt the message *C*, the receiver first calculates:

$$H = C.W^{-1} \bmod m .$$

Then he applies his secret vector *S* to solve this knapsack problem for *H* and recover *X*. In this method *W,m* and *S* must be kept secret and *A* is published in a public directory. In 1980 Shamir found that if the value of the modulus *m* is known it may

be possible to determine the secret vector $A$ [Sha80]. In 1982, Shamir introduced another approach [Sha82] to deduce $W$ and $m$ by using the elements of the public vector only.

In 1978 another public key algorithm was introduced by Rivest, Shamir and Adleman, called RSA [RSA78]. The RSA public key cryptosystem is based on the fact that although finding a large prime is computationally easy, factoring the product of two such numbers is computationally infeasible. In this method, the user chooses big primes $p$ and $q$ and computes $n = p.q$ and $m = (p-1)(q-1)$. He then chooses $e$ to be integer in $[1,m-1]$ with greatest common divisor GCD$(e,m) = 1$, and computes $d$ such that $e.d = 1 \pmod m$. Now $n$ and $e$ are public; $d,p,q$ are the secret key.

After a user has computed $p$, $q$, $e$, and $d$ the encryption transformation $E$ and the decryption transformation $D$ are defined by:

$$C = E(M) = M^{e} \bmod n$$
$$M = D(C) = C^{d} \bmod n$$

where $M$ is the plaintext block and $C$ is the cipher text block.

In 1984 T.ElGamal proposed a new public key algorithm based on the difficulty of computing discrete logarithms over finite fields [ElG85]. In this system, each user has two keys, the private key $x$ and the public key which consists of three elements $(y, \alpha, p)$, where $p$: is a large prime integer, $\alpha$: is a primitive element to $p$, and $y$: an integer calculated by:

$$y=\alpha^{x} \bmod p$$

To encrypt a message $M$ using this method, the sender first chooses a value $k$, $0 < k \leq p-1$, and then computes :

$$Key = Y_B^{\,k} \bmod p$$

where $Y_B$ is the receiver's public key, which is $Y_B = \alpha^{x_B}$.

Second, the sender forms the ciphertext which consists of the pair $c_1$, $c_2$:

$$C_1 = \alpha^{\,k} \bmod p$$
$$C_2 = Key.M \bmod p.$$

These two messages are sent as a ciphertext corresponding to the plaintext $M$. $C_1$ provides information about the chosen value $k$ which helps the receiver to find *key* and recover the plaintext $M$ from $C_2$. This system is not secure if the same $k$ is used in more than one block. To recover the plaintext message, the receiver obtains the *key* value by rising $C_1$ to the power of his private key $x_B$, since $key = \alpha^{k.x_B}$. The plaintext message $M$ is then revealed by dividing $C_1$ by *key mod p*.
One of the disadvantages of this method is that the ciphertext is double the size of the plaintext, and the public key file is triple the size of the *RSA* public key file.

Using public-key cipher algorithms for encryption might give the impression that any user such as *Charlie* can send to *Bob* a message impersonating *Alice* and fooling *Bob*. This is correct if the public key directory is open for anyone to add his/her public key or pick-up an other's public key without any control or supervision. In practice, the case is completely different. A trusted certification authority assigns a unique name to each user and issue a certificate containing the name and the user's public key. A Certifying Authority *(CA)* signs all certificates. If *Alice* and *Bob* want to communicate, each of them has to verify the signature of other person's certificate. If they use the same *CA*, this is easy. If they use different *CA*s, then a tree structure of different *CA*s will be involved in the verification. On the top of the structure there is one master *CA*. Each *CA* stores the certificate obtained from the superior *CA*, as well as all the certificates issued by it. *Alice* and *Bob* have to traverse the certification tree, looking for the common trusted point where the *CA* can certify *Alice* to *Bob* and *Bob* to *Alice*.
Certificates have a specific validity period. When a certificate expires, it should be removed from any public directories maintained by the *CA*s. The issuing *CA*, however, should maintain a copy of the certificate. It will be required to resolve any

dispute that might arise. This method of authentication has been recommended by *ISO* as an authentication framework and known as the *X.509* protocols [Fah93],[Sch94]. There are three types of protocols under *X.509*, one-way, two-way, or three-way authentication protocols. One way protocol is a single communication from *Alice* to *Bob*. Two-way protocol is identical to one-way protocol, but it also adds a reply from *Bob*. Both protocols use time-stamps. A three-way protocol adds another message from *Alice* to *Bob* and obviates the need for time-stamps.

The one-way protocol can be demonstrated as following:

1) Alice generates a random number $R_A$.

2) Alice constructs a message, $M_A = (T_A, R_A, I_B, Data)$, in which $T_A$ is Alice's time-stamp. $I_B$ is Bob's identity, and *Data* is an arbitrary piece of information. The *Data* may be encrypted with Bob's public key, $E_B$, for security.

3) Alice sends $D_A(M_A)$ to Bob.

4) Bob obtains Alice's public key $E_A$. He makes sure that this key has not expired.

5) Bob uses $E_A$ to decrypt $D_A(M_A)$. This verifies both Alice's signature and the integrity of the signed information.

6) Bob checks the $I_B$ in $M_A$ for accuracy.

7) Bob checks the $T_A$ in $M_A$ and confirms that the message is current.

The two-way protocol consists of the one-way protocol and then the same steps from Bob to Alice, except that the message $M_B$ from Bob to Alice contains Alice's random number $R_A$ as an extra information. The three-way protocol accomplishes the same thing as two-way protocol, but without time-stamps ($T_A = T_B = 0$) and the following extra steps:

- Alice checks the received version of $R_A$ against the $R_A$ she sent to Bob.
- Alice sends $D_A(R_B)$ to Bob.
- Bob uses $E_A$ to decrypt $D_A(R_B)$. This verifies both Alice's signature and the integrity of the signed information.
- Bob checks the received version of $R_B$ against the $R_B$ he sent to Alice.

The main problem in public-key systems in general is the need for

management, security and maintenance of a large public-key file which contains all users' public keys (sometimes called the *public-key directory*). Such a file contains sensitive data that must be protected well, otherwise it will be an easy target to attack. In case of partial or entire damage being caused to this file, the entire system would collapse. Maintaining and securing such a file is not an easy task.

The range of applicability of public key systems is limited in practice by relatively low bandwidth associated with public-key cipher, compared to their conventional counterparts. It has not been proven that time and space complexity must necessarily be greater for public key systems than for conventional systems. However, the public key systems that have withstood crypt-analytical attacks are all characterized by a relatively low efficiency. Some are based on modular exponential, a relatively slow operation, others are characterized by high data expansion. This inefficiency seems to preclude the use of public key systems as replacements for conventional systems utilizing fast encryption techniques such as permutations and substitutions. That is, the use of the public key systems for bulk data encryption is not feasible. In fact, the two major application areas for public key cryptosystems are distribution of secret keys and digital signature.

## 2.2 Key Exchange Protocols

The first scheme that solved the key distribution problem was proposed by Diffie and Hellman in 1976 [DH76]. Diffie-Hellman scheme can be described as follows:

Let $p$ be some large prime number and let $g$ a primitive element of $GF(p)$, where $1< g <p-1$. If two users such as Alice and Bob wish to establish a common key for their secure communication, Alice selects a random number $x \in [1, p-1]$ and computes

$$P_A = g^x \ (mod\ p). \qquad ...(1)$$

Similarly Bob chooses a random number $y \in [1, p-1]$ and computes

$$P_B = g^y \ (mod\ p).$$

Alice and Bob exchange their $P_A$, $P_B$ values (public keys ) over the insecure channel, but they keep $x$ and $y$ as their secret. Finally Alice Computes $P_B^x \ (mod\ p)$ and Bob computes $P_A^y \ (mod\ p)$ as their common key, since:

$$K = P_B^x \ (mod\ p) = P_A^y \ (mod\ p) = g^{xy} \ (mod\ p).$$

Yacobi and Shmuely [YS90] propose a Diffie-Hellman related key exchange system. Their system has two advantages over the original Diffie-Hellman one. The first is providing a different common key for each session based on the random numbers that are selected by the parties, and the second one is using the RSA-like modulus (called sometimes *Composite Diffie-Hellman CDH*), which makes the scheme more secure. Shmuely and later McCurly [McC88] proved that the difficulty of breaking the Diffie-Hellman system with a composite modulo $n$ (RSA-like) can be made equivalent to the factoring problem and it is much harder to break than the original one, since an attacker will face two hard mathematical problems, factoring a large composite number $n$, and computing a discrete logarithm in the field of the factors of $n$. In this scheme, each user has a secret key $s$ and a public key $P = g^s \ (mod\ n)$ generated by a centre. If Alice and Bob wish to communicate secretly, they select a random numbers $r_A$, $r_B$, and compute:

$$x_A = r_A + s_A, \qquad x_B = r_B + s_B$$

respectively.

They afterward exchange their $x$ elements and the session key is computed in each side as:

$$k_A = (g^{Xb}.P_B^{-1})^{rA}$$

$$k_B = (g^{xA}.P_A^{-1})^{rB}$$

$$K = k_A = k_B = g^{rA.rB} \ (mod\ n).$$

Diffie-Hellmans' idea has been widely used in methods of generating session

keys for different applications such as group oriented cryptography in Hwang protocol [Hw91], or in a digital mobile communication system that was proposed by Tatebayashi-Matsuzaki [TM90], and in many other protocols [AMV89] [FR90].

These types of public key systems solved the key distribution problem among trusted partners. These systems still have the main public-key cryptosystem's problem, which is the need for management, security and maintenance of a large *public-key directory* .

The best solution to overcome these problems is to find an alternative key distribution method that provides the following properties:

1. A user's public key must be related to his/her identity to avoid personation problem, (authenticated public key).
2. Drop the need for public-keys directory, allowing users to contact each other directly (eliminates the management and security problems of the public-keys directory).
3. Availability of a trusted authority (trusted centre) that provides some secret information to each user and where no one else can generate such information.

Protocols with such characteristics are called *identity-based* key-exchange protocols.

### 2.2.1. Identity-Based Key Exchange Protocols

An identity-based key exchange protocol has in general two phases: *Card issue phase* and a *Communication phase*. In the first phase the trusted centre typically distributes a *smart card* to each user, which is a tamper-proof integrated circuit (IC) card which includes the system and user's public information as well as the user's secret key(s). In the second phase, users communicate securely with each other using their smart cards (The Card issue phase, in some protocols, is divided into two phases called *set-up* phase and *pre-authentication* phase [Gü90],[BK90]).

Shamir [Sha85] proposed in 1984 the first interesting approach for identification and digital signature. In his approach, the user only needs to know the

identification information of his communication partner and the public key of the authority centre.

During the last few years, several new identity-based key exchange schemes have been proposed, started by the Japanese researcher E. Okamoto [Ok86], who introduced an idea for an interactive Id-based key exchange protocol, and discussed its usage for centralized and decentralized networks. He later used the same idea to provide a secure mail system [TO90]. The following is the sketch of his protocol:

*Okamoto's ID-based key exchange protocol for decentralized networks:* In common with all identity-based key exchange protocols, it has two phases. In the first phase, the Authority Centre (AC) generates the basic elements of the RSA public key cryptosystem, which are the two prime numbers $p$, $q$ each of them is about 256 bits long, a primitive integer $g$ in $GF(p)$ and $GF(q)$, and numbers $e,d$ such as:

$$e.d = 1 \ (mod \ (p-1)(q-1))$$

If Alice wishes to join the system she gives the authority centre her identification $ID_A$, the AC then calculates her secret integer $s_A$ :

$$s_A = ID_A^{-d} \ (mod \ n) \ , \text{ where } n = p.q.$$

and stores the integers $(n,g,e,s_A)$ in Alice's card. Bob does the same for joining the system.

The second phase begins when users such as Alice and Bob wish to establish a communication. Each of them chooses a random number $r_A$, $r_B$ respectively. Alice computes her public key:

$$P_A = s_A.g^{rA} \ (mod \ n) \qquad ...(2)$$

as well as Bob : $$P_B = s_B.g^{rB} \ (mod \ n)$$

They exchange their public keys. The session key is calculated by both ends as following:

$$k_A = (P_B^{\,e}.ID_B)^{rA} \pmod n$$

$$k_B = (P_A^{\,e}.ID_A)^{rB} \pmod n$$

$$\Rightarrow K = k_A = k_B = g^{e.rA.rB} \pmod n \quad ...(3)$$

After that, they can use any symmetric encryption algorithm, such as *DES* [NBS77] or *FEAL* [SM88] to encrypt or decrypt messages using the resulted session key.

In *centralized* networks, all the communications goes through a network centre, so the authority centre in the Okamoto protocol for such a network supplies the network centre with the values of $(n,e,r)$, where $r$ is any fixed integer less than $n$, and issues users' smart cards containing similar information as in the previous scheme but replacing $e$ by $y$, where :

$$y = g^{e.r} \pmod n$$

When Alice wishes to generate a session key between herself and the network centre she generates a random number $r_A$, and computes :

$$P_A = s_A.g^{rA} \pmod n$$

She then sends $P_A$ to the network centre. The session key between them can be generated by Alice as:

$$K = k_A = y^{rA} \bmod (n) = g^{e.r.rA} \pmod n$$

which can be computed by the network centre as:

$$k_C = (P_A^{\,e}.ID_A)^{r} \pmod n$$

As it appears in this protocol, there is no need to keep a file for public keys, instead, the public information $P_A$ and $P_B$ are exchanged directly between the parties, and this public information is related to its user's identity, which makes sure that Alice is talking to Bob and not to anybody else. If another user such as Charlie tried to personate Bob, different keys would result in each side.
Rewriting the formula (2) as :

$$P^e.ID = g^{e.r} \pmod n \qquad ...(4)$$

This explains the relationship between the user's public key, and his/her identity in Okamoto Id-based key exchange scheme. Only one data-exchange is required in this protocol of the size $D \leq 2.|n|$, where $|n|$ donates the number of bits, ($n$= *512* bits from each direction), and the maximum number of modular multiplications required in each side is:

$$M < 2|n| + |e| + 2$$

$|n|+1$ of these modular multiplications can be achieved off-line (e.g. a user might select a random number and compute his public key $P_i$ in advance).
This method appears to be as secure as Diffie-Hellman key distribution system and the RSA cryptosystem, but it has not yet been proved.

In 1989, two similar interactive Id-based key exchange protocols were proposed by Günther [Gü90] and Bauspieß-Knobloch [BK90]. Both are based essentially on the ElGamal digital signature scheme, and both used a kind of zero-knowledge proof to implement the authentication procedure [Be89][CED87] which ensures Bob that Alice is authentic and vice versa. At the end of the authentication procedure, a user ends up with a key as a power of a base value different than his partner's one. Thus both users use the commitments of the respective verifiers in these protocols, which are authenticated if the protocols end successfully, as inputs to Diffie-Hellman exchanges. They thus end up with two keys on each side, which they could then suitably combine to construct the session key.

Using one iteration in Beth zero-knowledge protocol and having only one user secret value, the Günther and Bauspieß-Knobloch key distribution schemes can be described as follows:

First, the authority centre generates a large prime $p$ and a primitive element $g \in GF[p]$. It also selects a random number $x \in [1,p-1]$ as its own secret key, and computes its public key $y = g^x \pmod p$. When Alice wishes to join the system, she visits the authority centre providing her identification information $I_A$. The centre computes her identity string $ID_A = f(I_A)$ where $f$ is a one-way-hashing function. The centre selects a random number $z_A \in [1,p-1]$, and computes Alice public key $P_A = g^{zA} \pmod p$, and her secret key $s_A$ that satisfies:

$$ID_A = x.P_A + z_A.s_A \bmod (p-1).$$

The centre issues a smart card to Alice contains $(ID_A, P_A, s_A)$. and keeps $x$ and $z_A$ secret. The second phase of these protocols begins when two users such as Alice and Bob wish to communicate secretly. They apply the following steps :

| **Alice** | | **Bob** |
|---|---|---|
| 1- | $I_A$ , $P_A$ ⟶ | |
| | ⟵ $I_B$, $P_B$ | |
| 2- computes: $ID_B = f(I_B)$<br>$x_A = P_A^{-rA} \pmod p$ | | computes: $ID_A = f(I_A)$<br>$x_B = P_B^{-rB} \pmod p$ |
| | $x_A$ ⟶ | |
| | ⟵ $x_B$ | |
| 3-chooses random:<br>$e_B \in [1,p-1]$ | | chooses random:<br>$e_A \in [1,p-1]$ |
| | $e_B$ ⟶ | |
| | ⟵ $e_A$ | |
| 4- computes<br>$W_A = r_A + e_A.s_A \bmod (p-1)$ | | computes:<br>$W_B = r_B + e_B.s_B \bmod (p-1)$ |
| | $W_A$ ⟶ | |

$\leftarrow$ $W_B$ $\rightarrow$

5- verifies:
$g^{eB.ID_B} \stackrel{?}{=} y^{PB.eB}.P_B^{WB}.x_B \pmod p$

if not verified: *HALT*
else
6- chooses a random $z_A \in [p\text{-}1]$,
computes: $E_A = P_B^{zA} \pmod p$

verifies:
$g^{eA.ID_A} \stackrel{?}{=} y^{PA.eA}.P_A^{WA}.x_A \pmod p$

if not verified: *HALT*
else
6- chooses a random $z_B \in [p\text{-}1]$
computes: $E_B = P_A^{zB} \pmod p$

$\rightarrow$ $E_A$ $\rightarrow$
$\leftarrow$ $E_B$ $\leftarrow$

6'- *(Extra step in Günther protocol only)*
computes:
$P_B^{sB} = g^{ID_B}.y^{-PB} \pmod p$

computes:
$P_A^{sA} = g^{ID_A}.y^{-PA} \pmod p$

7- constructs the session key $K = k_1.k_2$
where :
$k_1 = x_B^{zB}$, $k_2 = E_B^{rA}$ (Bauspieß-Knobloch) $k_1 = x_A^{zA}$, $k_2 = E_A^{rB}$

$k_1 = (P_B^{sB})^{zA}$, $k_2 = E_B^{sA}$ (Günther scheme) $k_1 = (P_A^{sA})^{zB}$, $k_2 = E_A^{sB}$

The resulted session key form Günther scheme is :

$$K = g^{x_A \cdot s_A \cdot z_B + x_B \cdot s_B \cdot z_A} \pmod p$$

and from Bauspieß-Knobloch is:

$$K = g^{x_A \cdot r_A \cdot z_B + x_B \cdot r_B \cdot z_A} \pmod p$$

The use of zero-Knowledge proof protocols for authentication, such as Beth or Chaum-Evertse-de Graaf [CED87] presents some drawbacks to these Id-based key exchange schemes, because these authentication protocols require many data-exchanges. Therefore, more communication time and memory space are required in these systems.

Both Günther and Bauspieß-Knobloch protocols require at least six data exchanges (using one iteration during the authentication procedure and having only one secret key for each user). The maximum size of each of these data-exchanges is

approximately the size of $p$ (512 bits), that gives the total number of bits that transfer in both directions to generate a session key using one of these protocols:

$$D \leq 2\,(2\,|p-1|\ +4\,|p|\,) \approx 12\,|p|\ \text{bits.}$$

The number of modular multiplications required in Bauspiß-knobloch scheme is :

$$M \leq 7\,|p|\ + 6$$

and one modular addition in each side ( for only one iteration within the zero-knowledge protocol). If $p$ = 512 bits long, then M≤ 7 x 512 + 6 = 3590 modular multiplications in each side. Comparing this scheme with Okamoto's scheme in which the composite modular $n$ has the same bit-length as $p$ , the transmission efficiency here is approximately *six* times less than Okamoto's one, and its processing speed is approximately 3.5 times less than Okamoto's. Günther protocol has $2\,|p|$ modular multiplications more than Bauspieß-Knobloch one.

The security of both protocols is believed to be related to security of ElGamal digital signature system and Diffie-Hellman scheme. The security level depends only on the length of the words exchanged and not on the number of exchanges.

T.Okamoto and K. Ohta [OO91] proposed other key distribution systems in which they make use of the randomized information that is exchanged between the prover and the verifier in zero-knowledge protocols such as Fiat-Shamir [FS87] and its variants [GQ89][OhO89] or Beth [Be89]. They suggested that 12 Id-based key exchange protocols could be constructed from the above four types of zero-knowledge protocols, since each of them could be implemented in a *sequence*, *parallel* or *non-interactive* form [OhO89].

The security of Id-based protocols that use the Fiat-Shamir scheme in their authentication phase is associated to the security of both the Fiat-Shamir scheme,(which is based on the fact that extraction of modular square roots of random values is as difficult as the factorization of the modulus) and the Diffie-Hellman key

exchange scheme.

The total number of bits that transfer between two users during the implementation of Okamoto-Ohta key exchange scheme (based on the parallel version of the extended Fiat-Shamir zero-knowledge protocol) is:

$$D \leq 8 \mid n \mid \text{ bits}$$

and the number of modular multiplications required is: $3 \mid n \mid +2 \mid e \mid +3$ in each side. The parallel version of Okamoto-Ohta key-exchange scheme is slower than E.Okamoto's one and requires more data to be transferred between the users. On the other hand, it is still faster than both the Günther and Bauspieß-Knobloch methods.

In 1991 Girault [Gi92] proposed another non-interactive Id-based key exchange scheme in which the modulus is also a composite large integer $n$. The first phase of this scheme is approximately similar to Okamoto's one where the authority centre generates all the RSA elements. The difference here is that Girault introduced the *self certified* principle where the secret key is selected by the user and the public information is generated by co-operation between the user himself and the centre, to avoid cheating by the centre. So the user selects a random value $s$ as his secret, computes $u = g^{-s}$ *(mod n)* and gives $u$ and his/her *ID* to the centre. The authority centre computes a user public key as:

$$P = (g^{-s}\text{-ID})^d \pmod{n} \qquad \ldots (6)$$

Because the centre does not know the user secret key, he can not cheat, and neither can the user.

Generating the session key between two users such as Alice (with $ID_A$, $s_A$,$P_A$) and Bob ( $ID_B$, $s_B$,$P_B$) is carried out as:

$$k = (P_A^e + ID_A)^{s_B} = (P_B^e + ID_B)^{s_B} = g^{s_A \cdot s_B} \pmod{n} \qquad \ldots (7)$$

Girault protocol is a non-interactive one, which means there is still a need for a public directory containing all users' public keys, and also the same session key will

be generated each time. There is no data-exchange during the construction of the session key in this protocol, except access to the public directory to look up the partner's public key. The main use of such non-interactive key exchange protocols is for one-way transmission applications such as electronic mail.

The number of modular multiplications needed during the construction of the session key in this protocol is $M = |e| + |s|$ and one modular addition is also required in each side. The difficulty of breaking this protocol is related to Diffie-Hellman.

Maurer and Yacobi [MY91] proposed an idea for a new non-interactive key distribution system, and later [MY92, *rump session*] they discussed the limitation of this method and proposed some possible solutions. The idea of their scheme was to use Diffie-Hellman scheme in such a way that the public key is equal to the user identity, mathematically :

$$P = ID = g^s \ (mod\ n)$$

where $n$: a big composite number.

$s$: a user secret key issued by the authority centre,

The problem here is that not every *ID* has a discrete logarithm ( e.g. the centre could not be able to find the secret key value $s$ for each arbitrary *ID* value given by a user), and in other hand, calculating a discrete logarithm is a very difficult problem.

In [MY92] they proposed some solutions for their protocol's problems, such solutions were:

1)-selecting the composite modulo $n$ as a product of some primes, e.g. $n = p_1.p_2. \ \ldots \ .p_r$, and $p_i$ is strong prime. These primes are small enough so that computing discrete logarithm (DL) is feasible and finding the prime factors of $n$ is hard.

2)- Or selecting the composite modulo $n$ as a multiplication of two primes, e.g. $n = p.q$, where $p-1$ ,$q-1$ has only moderate size prime factors.

A practical implementation for Maurer-Yacobi's idea has been discussed and implemented on a 25 MHz 386 Personal Computer by Scott and Shafa'amry [SS92b].

The composite modulo in this implementation is chosen as a product of two primes. The size of each of these primes is 80 decimal digits. The prime numbers constructed in such a way that it is easy to compute a discrete logarithm using Pollard's method [Pol78], and at the same time it is hard to compute prime factors of *n*.

The characteristics of all the above studied protocols are illustrated in Table 1, and the efficiency of the interactive schemes is compared in Table 2.

| Scheme Name | mod | Secret key | Relationship of ID,P,s(or $x^1$) | Type Non-interact/ interact | Session Key K |
|---|---|---|---|---|---|
| E.Okamoto | n | $ID^{-d}$ | $P^e.ID = g^{-e.x}$ | interactive | $g^{e.xA.xB}$ |
| Maurer-Yacobi | n | $\log_g ID$ | $P = ID = g^s$ | Non-interact. | $g^{sA.sB}$ |
| Girault | n | random s | $P^e+ID = g^{-s}$ | Non-interact. | $g^{-sA.sB}$ |
| Günther [2] | p | (ID-x.P)/z | $y^P.P^s = g^{ID}$ | interactive | $g^{xA.ZB.sA+xB.ZA.sB}$ |
| Bauspieß-knobloch | p | (ID-x.P)/z | $y^P.P^s = g^{ID}$ | interactive | $g^{xA.ZB.rA+xB.ZA.rB}$ |
| T.Okamoto-Ohta | n | $f(I,j)^{-1/2}$ | $s = f(I,j)^{-1/2}$ | interactive | $\Sigma_i\, g^{xAi.xBi}$ |

[1] x : a user random number

[2] $y = g^x \pmod p$: The centre public key.

**Table 1**

*A summary of the features of all the above studied Id-based key exchange algorithms.*

| Scheme Name | mod | SM | D (Bits) | M |
|---|---|---|---|---|
| E.Okamoto | n | \|n\| | 2\|n\| | 2\|n\|+\|e\|+1 |
| Günther | p | \|p\| | 12\|p\| | 9\|p\|+6 |
| Bauspieß-Knobloch | p | \|p\| | 12\|p\| | 7\|p\|+6 |
| T.Okamoto-Ohta[1] | n | \|n\| | 8\|n\| | 3\|n\|+2\|e\|+3 |

[1] T.Okamoto_Ohta: using parallel version of extended Fait-Shamir zero-knowledge protocol.

**Table 2** *Illustrates*

*The Secret Memory size **SM** , the Transmitted information size **D** (bits) and the modular multiplications **M** required for each interactive Id-based key exchange protocols*

## 2.3 File and Communication Security

Block ciphers operate on blocks of data of fixed size, but a message or a file is of arbitrary length. One of the basic methods when using a block cipher to encrypt a file is to partition the file into blocks of fixed size and encrypt each block individually, this method is known as *Electronic Code book (ECB).*The biggest danger of using this kind of technique arises when significant parts of the messages changes very little and appear in fixed locations. Analyzing these parts becomes a 'code book' exercise in which the number of code values is small. The weakness of the ECB method lies in the fact that it does not connect the message's blocks together. By enciphering each block separately it leaves them as separate pieces which the cryptanalyst can analyze and assemble for his own benefit.

There are three other modes of operation that links all the blocks together and cover most of the requirements for the use of encryption in computer and network systems. These methods are:

1. Cipher Block Chaining (CBC).
2. Cipher Feedback (CFB).
3. Output Feedback (OFB).

These methods can be used with any block cipher. Each of them has its own advantages and applications.

### 2.3.1 Cipher Block Chaining (CBC)

Cipher block Chaining uses the output of one enciphered step to modify the input of the next, so that each of the cipher block is dependent not just on the plaintext block from which it immediately came, but on all the previous plaintext blocks. The first block is modified by an external block called *initializing variable (IV)* as it shown if figure 2.3. The choice of the *IV* value is very important and it must be the same for the sender and the receiver. Dividing the message into blocks leaves, at the end, a part less than a the size of the block. There are several ways in dealing with the short end blocks, one of them is padding some extra bits until the block reaches the correct size. However the number of the padded bit must be indicated somewhere so that the receiver can remove them. Another method has been suggested in [DP84] in which the last complete ciphertext block in the chaining process is enciphered again and used by X-OR to treat the last, short block as shown in figure 2.3. CBC is the recommended method for messages of more than one block. This method avoids codebook analysis generally but not at the start of the chain. Communication systems generally use chain formats which begin with a serial number so that the first block differs for all chains using given key. CBC extends a single bit error in the ciphertext to affect two successive blocks at the plaintext block output.

### 2.3.2 Cipher Feedback (CFB)

This kind of technique is used for enciphering a stream of characters, where each character is represented by $K$ bits. The important differences between this method and the *CBC* are that the block encryption operation, e.g. DES, take place in the feedback line at the transmission side and in the feed-forward line at the receiving

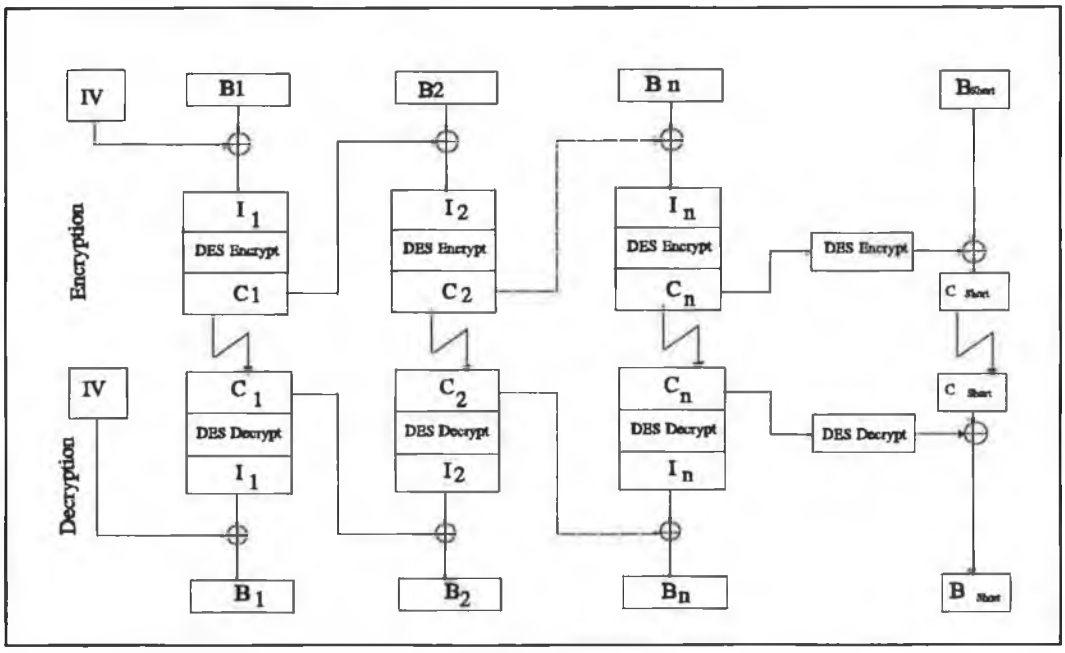



**Figure 2.3** Cipher Block Chaining (CBC) mode

side, and the block cipher algorithm is performing an encipherment at both ends.

The process of the cipher feedback method is the bit-by-bit addition of a stream of K-bit characters coming from the last significant K-bit positions of a block cipher output, e.g. DES, into the plaintext K-bit character stream. The input of the block cipher comes from a shift register which contains the most recent bits transmitted as a ciphertext as shown in figure 2-4. An *initializing variable IV* must be loaded to the shift register at the beginning of the transmission session. This value must be the same at both ends. Like the CBC, cipher feedback chains the characters togethers, making the ciphertext a function of all the proceeding plaintext. This method is recommended for enciphering stream of characters when the characters must be treated individually. Error extension is present also here in CFB. In 8-bit CFB, 9 bits of ciphertxt are garbled by a single-bit error. After that, the system recoveres and all subsequent ciphertext is decrypted corrrectly. One subtle problem with this kind of error propagation is that if someone knows the plaintext of a transmission, he can toggle bits in a given block and make that block decrypt to whatever he wants. The next block will decrypt to garbage but, depending on the application, the damage may

already be done.

CFB is self-recovering with respect to synchronization errors as well. The error enters the shift register, where it garbles 8 bytes of data unit it falls off the other end. If someone tries to used this type of mode for full Block-size feedback (K = 64, the size of the entire block), the task of the shift register will be no longer effective, since the shifting by 64-bits means replacing the content of the register by the content of the feedback block. Moreover the structure of the CFB with full block-shift will be approximately similar to the CBC structure. Therefore, any successful attack on CBC mode will be effective on the CFB.

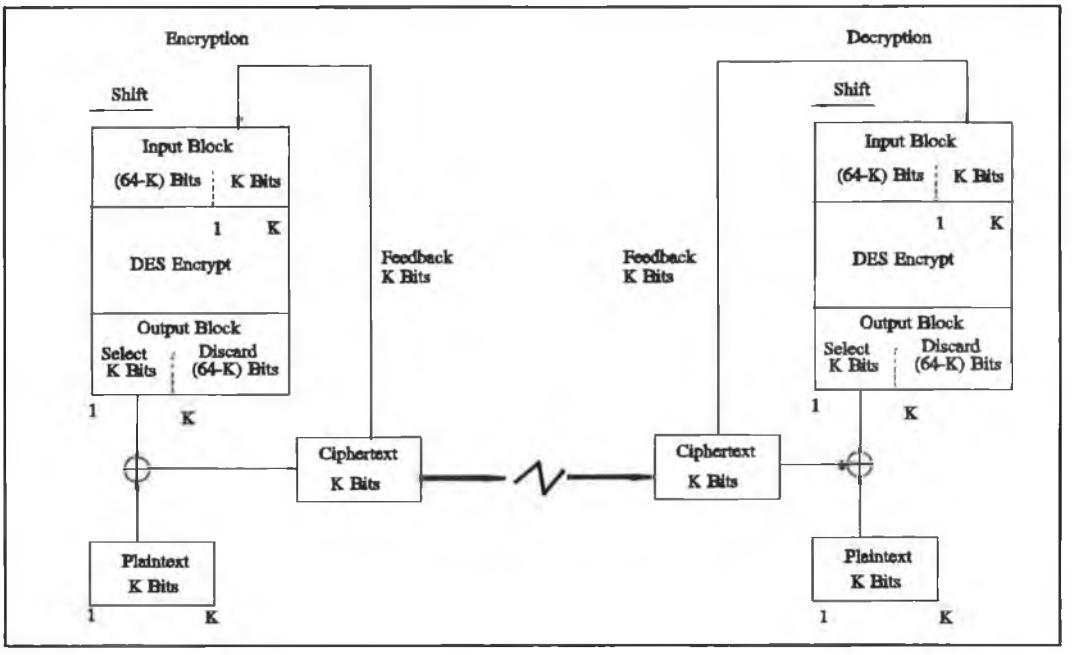



**Figure 2.4** K-bit Cipher Feedback (CFB) mode

### 2.3.3 Output Feedback (OFB)

The mode resembles CFB operation in all respects except the place from which the feedback is taken as shown in figure 2-5. It can be applied to stream of K-bit characters. It has the property that errors in ciphertext are simply transferred to corresponding bits of the plaintext output. The output feedback is needed when the

error extension is undesirable. In this method of operation, synchronization errors are not recovered.

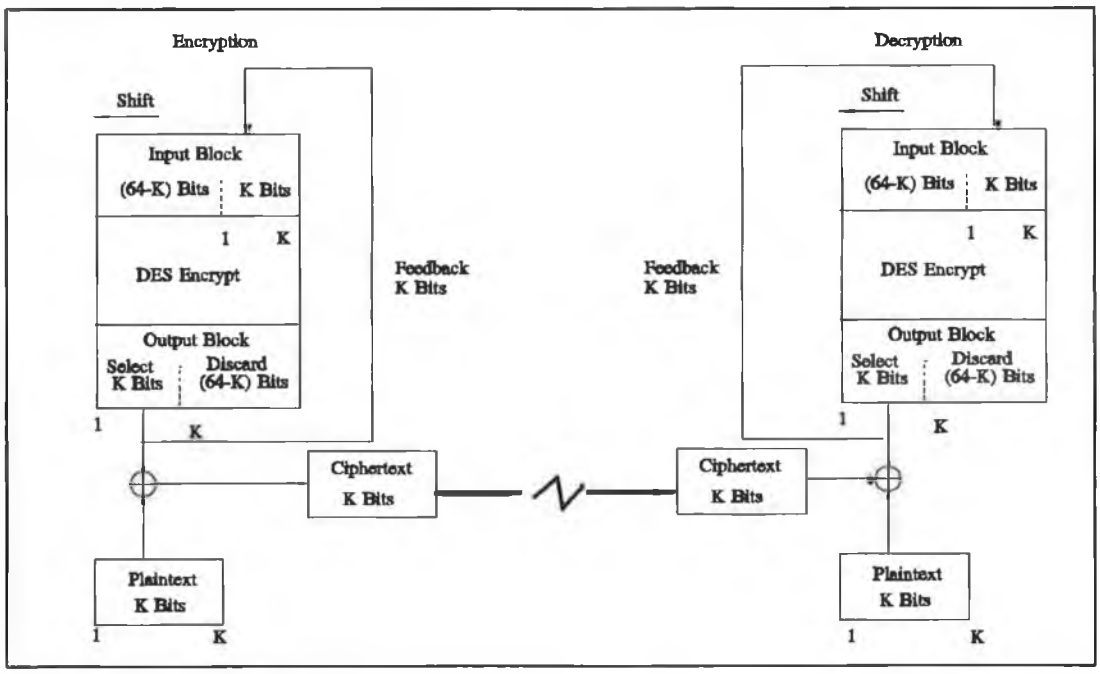



**Figure 2.5** K-bit Output Feedback (OFB)

## 2.4. Conclusion

Most of the published conventional cryptographic algorithms have been discussed in this chapter as well as the techniques of manipulating these block ciphers in securing messages and files of variant length. It appeared from this discussion that almost all of these cryptographic algorithms have been attacked by Biham and Shamir using their new cryptanalysis method which is called the *differential cryptanalysis*. Currently there are many commercial networks still basing their security on some of these conventional ciphers, mainly DES. Breaking such algorithms puts all these networks in jeopardy. The intensive need for a new conventional cipher which resists all known attacks, including the differential cryptanalysis, was the inspiration of this work.

The best known public-key algorithms have been also reviewed in this chapter. This review shows that there are many mathematically secure protocols for exchanging an

authenticated key which can be used within a strong conventional cryptographic algorithm. These methods are secure and will remain so for the foreseeable future.

We close this chapter by summarizing the required steps for achieving a secure communication session. If users such as Alice and Bob wish to establish secure communication, they have to implement the following steps:

1. Alice generates her public key which is related to her identification number, transmits it with her Id-token to Bob and vice versa.
2. Each of them authenticates the other's identification.(Not all Id-based key exchange protocols allow such verifications, e.g. Okamoto's method).
3. Alice and Bob generate together a *secret session key (K)* based on their identification tokens. This key will be used by both sides as a secret key for the selected block cipher.
4. If Alice wishes to send her secret message $M$ to Bob, she first encrypts the message using a strong block cipher algorithm with a mode of operation (e.g. DES with CBC mode) under control of the generated key. Then, she sends the encrypted message $C = E_K (M)$ to Bob over the network line. The type of block cipher algorithm and the operation's mode are agreed in advance between the communication's partners.
5. Bob decrypts the received message $C$ by implementing the same block cipher and mode of operation using the session key as a secret key for the block cipher algorithm $M = D_K (C)$.

For example, if Okamoto's Id-based key exchange protocol is selected to generate an identity-based secret session key, and *DES* with *CBC* mode is selected as a block cipher algorithm, Alice and Bob will communicate secretly as follows:

| **Alice** | | **Bob** |
|---|---|---|
| 1- Generates a random $r_A$ | | Generates a random $r_B$ |
| Computes $P_A = s_A.g^{rA}$ | | Computes $P_B = s_B.g^{rB}$ |
| where: $s_A = ID_A^{-d} (mode\ n)$ (Issued by the centre) | | where: $s_B = ID_B^{-d} (mode\ n)$ |
| | $P_A, ID_A$ ——> | |
| | <—— $P_B, ID_B$ | |
| 2- Calculates the secret session's key: | | Calculates the secret session's key: |
| $K_A = (P_B^e.ID_B)^{rA} (mode\ n)$ | | $K_B = (P_A^e.ID_A)^{rB} (mode\ n)$ |

The secret session key :

$$K_A = K_B = g^{e.rA.rB} (mode\ n)$$

3. Encrypts the message *M* using *DES-CBC:*

$$C = E_{KA}(M)$$

—— $C$ ——>

4. Decrypt *C* by using *DES-CBC*

$M = D_{KB}(M)$.

Let's see what will happen if someone traies to fool Alice and impersonate Bob. If Alice and Bob agreed in advance to use an Id-based key exchange protocol with a verification step, step number 2, (such as Gunther or Bauspiess-Knobloch protocol), a cheater who might try to impersonate Bob will be detected by Alice at this stage of the protocol. Alice will then halt the procedure and cancel the communication session before sending any message. If Alice and Bob are using an Id-based exchange key protocol which has no separate authentication step (such as Okamoto's one), the cheater will end-up with a key different than the one which has been generated by Alice. Therefore he will receive the encrypted message from Alice, but will never be able to reveal it.

# *Chapter 3*

# Methods of Cryptographic Attack

The possibility exists that unauthorized individuals can intercept data by eavesdropping. In fact, there are several methods of eavesdropping such as *wiretapping*, interception of individual transmissions over communication lines by using hardware connections, or *electromagnetic eavesdropping*, interception of wireless transmissions such as radio and microwave transmission. Eavesdropping is completely passive, where the opponent only listens to or records information being transmitted. An attack involving only eavesdropping is called a *passive attack*. If, in addition, the opponent modifies the transmitted information or injects information into the communication path, the attack is called an *active attack*.

Methods of attacking a cryptographic algorithm fall into two categories: *crypt-analysis* and *exhaustive* or *"Brute force"*, methods. Exhaustive methods can be further divided into two sub-categories: *key exhaustion* and *message exhaustion.*

Crypt-analytic methods can be divided into two sub-categories: deterministic or *analytical* methods, and *statistical* methods.

Some other methods of attack are a combination of more than one of the above classes.

In practice, the attack is carried out as a mixture of more than one class of attacks for the purpose of speeding up the search for the unknown quantity (the key, or the message). Attacks are also classified based on the type of the information available

to the crypt-analyst. An attack is called a *ciphertext-only* attack if the crypt-analyst has only access to ciphertexts. If the crypt-analyst knows some plaintext-ciphertext pairs, his attack is called a *known-plaintext* attack, and if the crypt-analyst is able to select the plaintext to be ciphered, his attack is then called *chosen-plaintext* attack.

## 3.1 Exhaustive attack

This attack method assumes that the opponent knows the cryptographic algorithm and possesses a fragment of ciphertext and/or corresponding plaintext. In an exhaustive attack, an attempt is made to recover the plaintext or key by using a direct search method. Recovering the plaintext is called *message exhaustion* while revealing the secret key is called *key exhaustion*. In *key exhaustion*, if only the ciphertext is available, a crypt-analyst must determine the key solely from intercepted ciphertext, though the method of encryption, the plaintext language, the subject matter of the ciphertext, and certain probable words may be known. The ciphertext can be decrypted with the trial key and the resulting plaintext can be inspected to see if it makes any sense. In this way, it can be determined if the trial key is a candidate for the unknown key or not. This type of the key exhaustive attack is called *ciphertext-only* exhaustive attack.

If the crypt-analyst knows some plaintext-ciphertext pairs, his attack is called a *known-plaintext* exhaustive attack Suppose an enciphered message transmitted from a user's terminal to the computer is intercepted by crypt-analyst who knows that the message begins with standard header such as "LOGIN". Such a known plaintext is enciphered with a trial key and the result is compared for equality with the known corresponding ciphertext.

Another type of key exhaustive attack is called *chosen-plaintext* exhaustive attack, in which the crypt-analyst is able to acquire the ciphertext corresponding to a selected plaintext. The crypt-analyst selects the plaintexts in such a way that serve him to cut down the number the trials needed to reveal the correct secret key. This kind of attack is possible when a user implements the cryptographic algorithm in the *ECB (Electronic Code Book)* mode. It would be the most favourable case for the crypt-

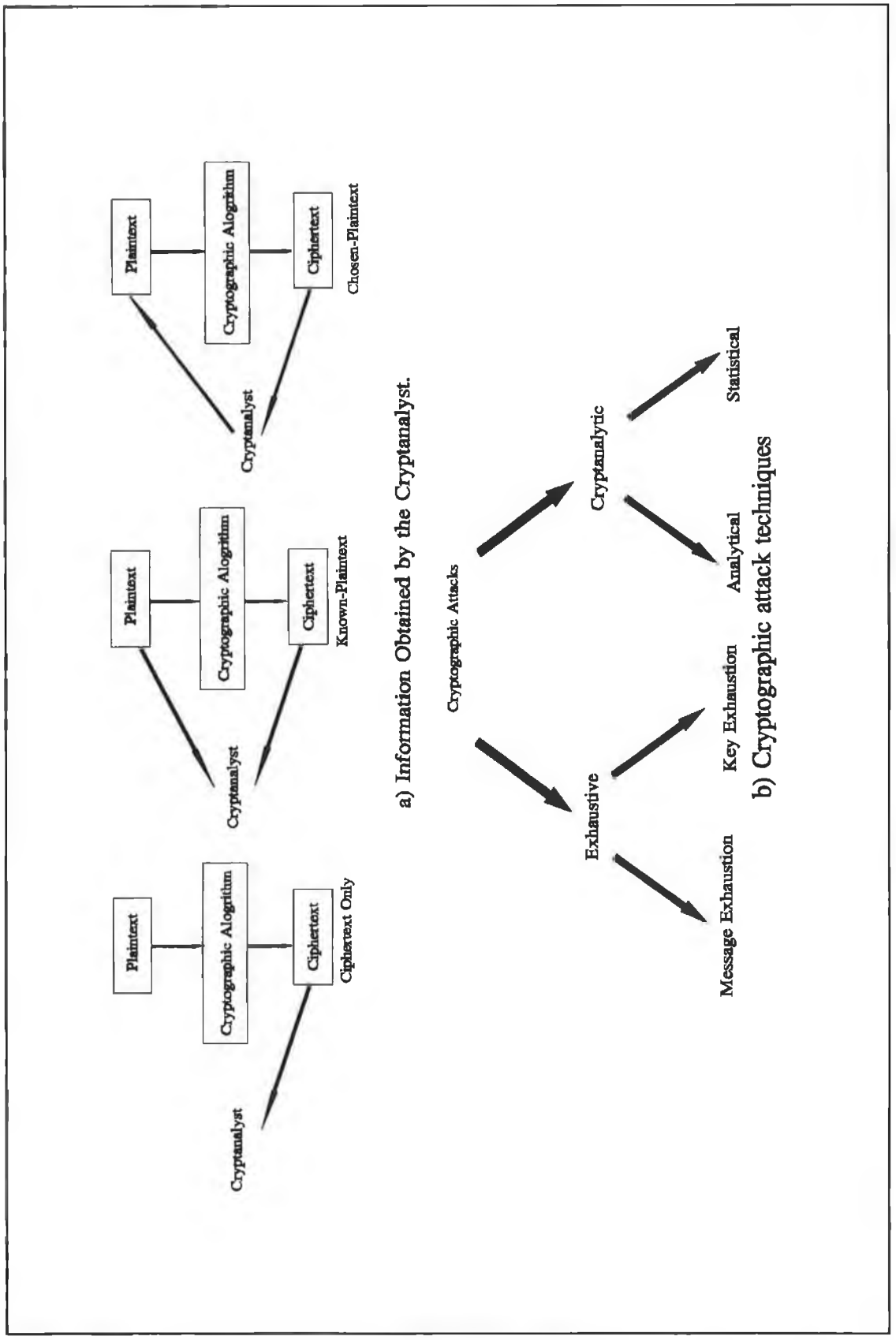



***Figure 3.1*** *The basic classification of Cryptographic Attacks*

analyst if he could manage to provide the user with a program that generates his selection of plaintexts either through a communication line or a storage media. The user enciphers these chosen plaintexts in the *ECB* mode and returns the generated ciphertexts to the attacker.

This type of attack can be prevented by reducing the user cooperation with the crypt-analyst and using the encryption method in either *CBC (Cipher Block Chaining)* or *CFB (Cipher Feedback)* modes.

Exhaustive attacks can be thwarted in general by making the number of the required trials very large. However, the work factor of an exhaustive attack, which is directly proportional to the number of trials is so large that the attack is not feasible. This is not the case for other attacks.

## 3.2 Crypt-analytical Methods

Crypt-analytic methods can be divided into two sub-categories: deterministic or *analytical* methods, and *statistical* methods. In a deterministic approach, the crypt-analyst first attempts to express a desired unknown quantity (such as the key or message) in terms of some other known quantity or quantities (such as given ciphertext, or given plaintext and corresponding ciphertext) whose relationship to the unknown quantity depends on the nature of the algorithm. Then the crypt-analyst solves for the unknown quantity.

Let $Y$ denote the ciphertext produced by enciphering plaintext $X$ with cryptographic key $K$, and let $f_k$ represent the function that relates $X$ and $Y$:

$$Y = f_k(X)$$

In a deterministic attack against the key, the opponent tries to find a function $F$, where

$$K = F(X,Y)$$

such that $F$ can be represented by an easy computer procedure.

In a poorly designed algorithm, it may be possible to solve for the key by decoupling $F$ into a set of equations:

$$k_1 = F_1(Y,X)$$

$$k_2 = F_2(Y,X,k_1)$$

$$\vdots \qquad \vdots$$

$$k_n = F_n(Y,X,k_1,.....,k_{n-1})$$

and then to solve for the key bits $k_1,k_2,..,k_n$ one at a time. For instance, Davies [DP84] analyzed the DES cryptographic algorithm and reported that having sufficient data, a known-plaintext crypt-analytic attack yielded 16 linear relationships among the key bits, that reduced the size of the subsequent key search to $2^{40}$.

While analytical methods will generally succeed in breaking an algorithm that uses *linear* functions, this method of attack can be effectively thwarted if the algorithm makes use of *non-linear* functions of sufficient complexity.

In the *statistical* approach, the crypt-analyst attempts to exploit statistical relationships between plaintext, ciphertext, and key. To thwart statistical attacks, the algorithm's output (ciphertext) should be *pseudo-random.* In other words, for a large set of plaintext and key inputs, one must not be able, on the basis of statistical analysis, to reject the hypothesis that the output bit stream is random.

## 3.3 Meet-in-the-middle attack

*Meet-in-the-middle* attack is a *known-plaintext* attack in which a kind of combination between the *ciphertext-only exhaustive* and the *known-plaintext exhaustive* search techniques is used. Such an attack on a block cipher composed of $n$ consecutive rounds can be described as follows: Suppose a crypt-analyst has a plaintext $P$ and corresponding ciphertext $C$. For each guessed key $K$ the crypt-analyst enciphers $P$ with the first $s$ rounds of the cipher algorithm yielding $d_1$, and deciphers $C$ with the last $n$-$s$ rounds yielding $d_2$. If $d_1 = d_2$, the crypt-analyst concludes that $K$ is the true key. Considerably less guesses for the key are required compared to chosen-plaintext exhaustive key search when there are $i$ and $j$ such that both the $j$-th bit of $d_1$ and the $j$-th bit of $d_2$ are independent of the $i$-th key bit. Independence here means that for all $P$, $C$, and $K$, the $j$-th bit of $d_1$ and the $j$-th bit of $d_2$ are unchanged when the $i$-th bit of the key $K$ is complemented. Chaum and Evereste [ch3 ] applied this type of the

cryptographic attack on DES reduced to small number of rounds (4,5,6 and 7) and showed that the reduction factors of the key search are ($2^{19}$, $2^9$, $2^2$ and 2) respectively.

Meet-in-the-middle attack is considered one of the exhaustion attacks, therefore avoiding such kind of attack is possible by making the number of the required trials very large.

## 3.4 Differential Crypt-analysis

Differential crypt-analysis is a new type of chosen-plaintext statistical attack, introduced by Biham and Shamir in 1990, in which the crypt-analyst is concerned with the difference between a pair of plaintexts/ ciphertexts rather than the plaintexts and the ciphertexts themselves [BS91]. The differential crypt-analysis attack exploits the fact that the round function $F$ in an iterated cipher is usually cryptographically weak and tend to overuse the *X-OR* function define what is meant by *"difference"*. Thus, if a ciphertext pair is known and the difference of the pair of inputs to the last round can somehow be obtained, then it is possible to determine (some substantial part of) the key of the last round. In differential crypt-analysis, this is achieved by *choosing* plaintext pair ($X$, $X^*$) with a specified difference $A$ such that the difference $\Delta Y(r-1)$ of the pair of the inputs to the last round will take on a particular value $B$ with high probability.

The basic procedure of a differential crypt-analysis attack on an $r$-round iterated cipher is summarized in [LMM92] as follows:

1) Find an $(r-1)$-round differential $(A,B)$ such that:
   $p(\Delta Y(r-1) = B | \Delta X = A)$ has maximum, or nearly maximum, probability.
2) Choose a plaintext $X$ uniformly at random and compute $X^*$ so that the difference $\Delta X$ between $X$ and $X^*$ is A. Submit $X$ and $X^*$ for encryption under the actual key $Z$. From the resultant ciphertexts $Y(r)$ and $Y^*(r)$, find every possible value (if any) of the sub-keys $Z(r)$ of the last round corresponding to the anticipated difference $\Delta Y(r-1)=B$. Add one to the

count of the number of the appearances of each such value of the sub-key *Z(r)*.

3) Repeat 2) until one or more values of the sub-keys *Z(r)* are counted significantly more often than others. Take this most often counted sub-key, or this small set of such sub-keys, as the crypt-analyst's decision for actual sub-key *Z(r)*.

In the original differential crypt-analysis attack, all the sub-keys are fixed and only the plaintext can randomly be chosen.

Biham and Shamir were able to break the reduced variant of DES with eight rounds in few minutes on a personal computer, and break any reduced variant of DES with up to 15 rounds using less than $2^{56}$ operations and chosen-plaintext [BS91]. Later on, in August 1992, they modified their method and announced that they are able to compute the secret key of the full DES-16 rounds by analyzing about $2^{36}$ ciphertexts in a $2^{37}$ time. The modified differential crypt-analysis is able to analyze ciphertexts that are derived from up to $2^{33}$ different keys. Biham and Shamir also managed to break almost all the FEAL family using their new type of attack. They reported in [BS92a] that, by running the attack on a personal computer they found the secret key of the FEAL-8 in less than two minutes using 1000 pairs of chosen-plaintext with more than 95% success rate. The differential crypt-analytic attacks can be transformed into known-plaintext attacks, and can be applied even in the *cipher Block Chaining CBC* mode of operation, provided there is sufficiently many known plaintext/ciphertext pairs, about $2^{38}$ in case of FEAL-8 [BS92a].

In [LMM92], the iterated block ciphers have been explained in terms of a *Markov Cipher*[1]. The differential crypt-analysis for *PES* cipher is then considered using the transition matrix to calculate the 7-rounds high probability differentials. It has been shown that the most probable 7-round differential has a probability about $2^{-58}$

[1] An Iterated cipher with round function $Y = f(X,Z)$ is a *Markov cipher* if there is a group operation $\otimes$ for defining differences such that, for all choices of A and B,
$P(\Delta Y = B | \Delta X = A, X = V)$ is independent of V when the sub-key Z is uniformly random.

and a differential crypt-analysis attack of *PES* based on their proposed differential is shown to require all $2^{64}$ possible encryptions.

The differential crypt-analysis attack can be thwarted by making the function *F* in the cryptographic algorithm more complicated, which prevents building an easy differential relationship.

Differential crypt-analysis is considered the most dangerous method of attack, by which most of the published conventional cryptographic algorithms have been successfully broken, and it also have the property of its conversion into a known-plaintext crypt-analytical method by which a text encrypted by a block cipher with CBC mode can be attacked .

## 3.5 Conclusion

We conclude that, a well-designed cryptographic algorithm is one that will withstand all known crypt-analytical and exhaustive methods of attack including the differential crypt-analysis. But it should also be realized that if an algorithm has no crypt-analytical solution, then it can always be implemented in such a way that the minimum work factor of all brute force attacks is larger than any desired value. These points have been taken in the consideration during the design of the new cryptographic algorithm *DCU-Cipher* which is explained in the next chapter.

*Chapter 4*

# The Design of a Secure Communication System

## 4.1 Introduction

The discussion in the previous chapters showed that the recent results obtained using the new type of chosen plaintext attack, which called *differential cryptanalysis*, makes most of today's published conventional secret key block cipher systems vulnerable. That motivates us to design a new secret key block cipher system which resists all known methods of cryptanalysis including differential cryptanalysis. The proposed method has only four rounds. It is workable for either 64-bit plaintext/64-bit ciphertext or 128-bit plaintext/128-bit ciphertext, and the key in both styles is 128-bits long. Different algebraic group operations are selected and used in this cipher to make the algorithm suitable for both hardware and software implementation. The new method is called *DCU-Cipher (Dublin City University Cipher).*

The threat of the differential cryptanalysis attack goes much further, since Biham and Shamir observed that given enough matching known plaintext and ciphertext, differential cryptanalysis can be applied to attack a secret file which is encrypted using the Cipher Block Chaining (CBC) mode. This mode is often recommended and widely used for encrypting long messages, protecting them from a chosen plaintext attack.

Two new modes of operation for file and data communication encryption are also proposed in this work that thwart differential cryptanalysis. The first mode is called *Plaintext-Ciphertext Complex Block Chaining (PCCBC)* and the second is called *CBC-PX*.

The design principles and structure for both the new secret block cipher, *DCU-Cipher*, and the new operation modes for file encryption, are discussed in this chapter. The implementation of these methods, their security and some statistical tests are presented in the next chapter.

## 4.2 The Design of a Cipher System

No secure cryptographic system could be designed without looking back to Claude.E. Shannon's theory and his considerations which were published in 1949 and discussed in many text books such as [BP82], [DP84], [Koh86] and others. Shannon considered two very different notations of security for cryptographic systems. He first considered the question of *theoretical security*, by which he meant, "How secure is a system against cryptanalysis when the enemy has unlimited time and man-power available for the analysis of intercepted cryptograms?". Shannon's theory of security cast much light into cryptography, but leads to the pessimistic conclusion that the amount of secret key needed to build a theoretically secure cipher will be impractically large for most applications. Thus Shannon also treated the question of *practical security*, by which he meant: "Is the system secure against a cryptanalyst who has a certain limited amount of time and computational power available for the analysis of intercepted cryptograms?". Shannon also introduced the *perfect secrecy* notation and specified two general principles, which he called *diffusion* and *confusion* to guide in the design of practical ciphers.

The new block cipher algorithm is designed in accordance with Shannon's diffusion and confusion principles providing perfect secrecy and frustrating all known types of cryptanalysis attacks.

### 4.2.1 The Design Requirements:

The new block cipher algorithm must provide the following properties:

**1)- *Perfect secrecy*:**

The system is said to have perfect secrecy if, for every message $M_i$ and for every cryptogram (ciphertext) $C_j$,

$$P_j(M_i) = P(M_i)$$

where $P(M_i)$ is the *a priori* probability of $M_i$ being transmitted and $P_j(M_i)$ is the probability that $M_i$ was transmitted given that $C_j$ was received (*a posteriori* probability) [BP82]. In this case, the cryptanalyst who intercepts $C_j$ has obtains no further information to enable him to decide which message was transmitted.

For any ciphertext $C_j$, let $P(C_j)$ denotes the probability of obtaining $C_j$ from any message, and $P_i(C_j)$ the probability of obtaining $C_j$ if the message $M_i$ is transmitted. Let $P_u$ be the probability of choosing the transformation $F_u$, or equivalently, the key $K_u$, then $P_i(C_j)= \Sigma P_u$, where the summation is over all those $u$ for which $C_j = F_u(M_i)$. Bayes' Theorem [BP82] says that for any $M_i$ and $C_j$:

$$P_j(M_i).P(C_j) = P_i(C_j) \; . \; P(M_i)$$

Therefore a necessary and sufficient condition for perfect secrecy is that

$$P_i(C_j) = P(C_j)$$

for all $M_i$ and $C_j$. That means, for any messages $M_i$, $M_j$ and any ciphertext $C_k$, the total probability of the keys which transforms $M_i$ into $C_k$ is the same as that of all the keys which transform $M_j$ into $C_k$, $P_i(C_k) = P(C_k) = P_j(C_k)$. Thus, when each key is equally likely, the number of keys which transforms $M_i$ into $C_k$ is the same as the number of the keys which transform $M_j$ into $C_k$. Since $M_i$, $M_j$, and $C_k$ were arbitrary, this means that in a system with all keys equally probable, perfect secrecy implies that there is

number of messages, $w$ say, such that there are exactly $w$ keys which map any given message $M$ on to any given ciphertext. This leads to the following very important design condition:

> *The number of different keys in a perfect secrecy system must be at least as great as the number of possible messages, and there is exactly one key transforming each message to each cryptogram and all the keys are equally likely.*

Clearly perfect secrecy is highly desirable objective, since the cryptanalyst obtains no information whatsoever form his intercepted ciphertext.

**2)- *Confusion*:**

Confusion (or *substitution*) means that the ciphertext depends on the plaintext and the key in a complicated and involved way. The idea of confusion is to make the relation between a ciphertext and the corresponding key a complex one. This aims to make it difficult for statistics to pinpoint the key as having come from any particular part of the key space. In particular, it tries to ensure that all of the key is needed to obtain even very short ciphertexts. This implies that every message character enciphered will depend on virtually the entire key.

***3) Diffusion:***

Diffusion (or *permutation*) is re-arranging the order of the plaintext's binary bits. The idea behind the diffusion is to spread out the influence of a single plaintext digit over many ciphertext digits so as to hide the statistical structure of the plaintext. An extension to that is to spread the influence of a single key digit over many digits of ciphertext so as to frustrate a piecemeal attack on the key.

***4) Uniquely reversible Function with Involution Property***

Let $F$ denote a transformation function which maps a message $M$ into a ciphertext $C$. In other words, $C = F(M)$. If there is exist a function $Q$ that maps $C$ to $M$, $M = Q(C)$, then we call $F$ a reversible function and $Q$ is the inverse of $F$. If $F$ has a unique inverse $Q$ then we say that $F$ is *uniquely reversible* function and write:

$$F = Q^{-1}$$

This is a very significant property in a cipher system, where the encryption function which transforms a message $M_i$ into a ciphertext $C_i$ has a unique inverse which enables us to recover the correct plaintext $M_i$ from $C_i$. It would be nice if the same design for enciphering the plaintext could serve (with minor modifications) for deciphering the ciphertext. If a cipher system has the same structure for encryption and decryption procedures, we say that the system has the *involution* property. Therefore, a good block cipher is one which is designed to use the same structure (with minor modifications) for both encryption and decryption.

***6) Easy to implement in a hardware and software***

The design of a cipher system must make it difficult to attack, but at the same time, the operations and the computational functions which are involved in building the system must be selected to facilitate the hardware and software implementation of the algorithm. Therefore, implementing the cipher in either software or hardware must be easy without reducing its security or processing speed.

A new block cipher system have been designed that fulfils all the above mentioned requirements and called *DCU-Cipher "Dublin City University Cipher"*. This cipher system is applicable for the implementation in one of two modes, Called *DCU64* and *DCU128*. In the first mode, the plaintext and the ciphertext are both 64-bit long. The plaintext and the ciphertext in the second mode are blocks of 128-bit long, while the secret key in both modes is 128-bits long. The design is based on a mixed group operations which have been chosen to make the new cipher suitable for both software and hardware implementation.

In this cipher we used the principle of mixed operations form different groups which has been proposed by Lai and Massy [LM91]. Three of these operations are similar to those are used in their system (taking in the consideration that we are using them in two different modes), and the fourth one has been chosen as a $K$-bits right rotation, to increase the complexity in the transformation function making it more 2

difficult to attack.

Recall the definitions of the three different group operations on pairs of N-bit sub-blocks from [LM91], namely:

**1)** Bit-by-bit exclusive-OR of two N-bit sub-blocks, denoted by $\oplus$.

**2)** Addition of integers modulo $2^N$, denoted by $\boxplus$

**3)** Multiplication of integers modulo $2^N + 1$, with the N-bit block with all zeros represented by $2^N$. This operation is donated by $\odot$.

where: $N \in \{8,16\}$, and,

**4)** *K*-bits right variable Rotation, denoted by $\textcircled{R}$, where: $K \in \{0,1,..,7\}$.

### 4.2.2 The General Structure of DCU-Cipher

The general structure of the DCU-cryptosystem is illustrated in figure 4.1. The DCU-cipher consists of only four rounds. Each round begins by dividing the input block into eight equal size sub-blocks, $X_1$, ..., $X_8$ (8-bits long each in *DCU64* mode /16-bits long each in *DCU128* mode). Each of these input sub-blocks is then effected by one of the sub-keys $Z^r_i$ (where $r = 1,.. 5$ is the current round number and $i = 1,.., 10$ is the sub-key number within this round), that are generated from a 128-bit secret key block (see section 4.2.4). The sub-keys effect the first pair and the last pair of the input sub-blocks by using modular multiplication operations, while the second and the third pairs of the input sub-blocks are mixed with the key sub-blocks using modular addition operations, generating eight sub-blocks $X`_1$, ..., $X`_8$ as following:

$$\dot{X}_1 = X_1 \odot Z^1_1 , \quad \dot{X}_2 = X_2 \odot Z^1_2 , \quad \dot{X}_3 = X_3 \boxplus Z^1_3 , \quad \dot{X}_4 = X_4 \boxplus Z^1_4$$

$$\dot{X}_5 = X_5 \boxplus Z^1_5 , \quad \dot{X}_6 = X_6 \boxplus Z^1_6 , \quad \dot{X}_7 = X_7 \odot Z^1_7 , \quad \dot{X}_8 = X_8 \odot Z^1_8$$

The first four of these sub-blocks ($X`_1$, ..., $X`_4$) are Ex-Ored with the other four sub-

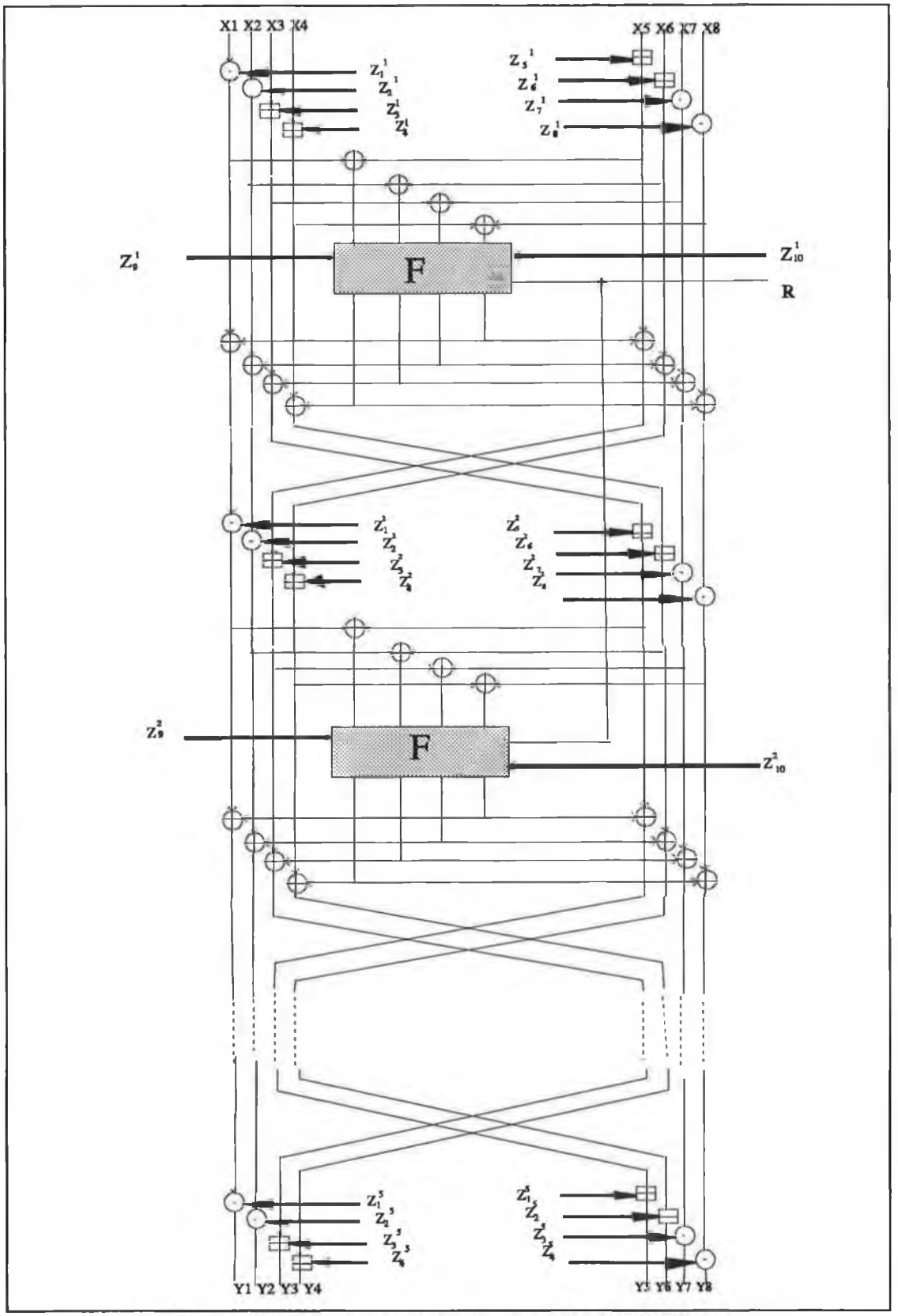



**Figure 4.1**. *The general structure of the DCU-Cipher*

blocks ($X`_5$, ..., $X`_8$) respectively, generating four inputs to the main transformation function $\boldsymbol{F}$, ($U_1$,..., $U_4$), which has seven inputs and four outputs (details about the structure of $\boldsymbol{F}$ are given in the next section). Each one of the four output sub-blocks of the function $\boldsymbol{F}$, ($W_1$, ..., $W_4$), is then Ex-Ored with a pair of the input sub-blocks, generating eight sub-blocks. Swapping some of these sub-blocks as shown in figure 4.1, generates the inputs of the next round as follows:

Let $W_i$ denote the output sub-blocks of $F$ function, where $i$= 1,..., 4., $X_j$ denote the input sub-blocks of the current round, $X`_j$ indicate the resulting sub-blocks of the effected input sub-blocks $X_j$ by the sub-keys $Z^r_j$, and $X``_j$ are the input sub-blocks of the next round, where $j$ = 1,..., 8.

$$X``_1 = \dot{X}_1 \oplus W_4 , \quad X``_2 = \dot{X}_2 \oplus W_3 , \quad X``_3 = \dot{X}_5 \oplus W_4 ,$$

$$X``_4 = \dot{X}_6 \oplus W_3 , \quad X``_5 = \dot{X}_3 \oplus W_2 , \quad X``_6 = \dot{X}_4 \oplus W_1 ,$$

$$X``_7 = \dot{X}_7 \oplus W_2 , \quad X``_8 = \dot{X}_8 \oplus W_1 .$$

This procedure is repeated four times constituting the DCU-cipher algorithm. At the end of the final round, a reverse swapping of the output sub-blocks is implemented (in other words, there is a cancelation of the output sub-blocks switching in the final round).

The ciphertext sub-blocks are then generated by effecting the final round outputs by key sub-blocks in the same way as happened at the beginning of each round.

### 4.2.3 The Transformation Function *F*

The structure of the transformation function $\boldsymbol{F}$ is illustrated in figure 4.2. This function has seven inputs, six of them are N-bit long ($U_1$,..,$U_4$ *and* $Z_9^r$,$Z_{10}^r$, where $r$ indicates the round number), and one 3-bits long input denoted by $\boldsymbol{R}$. The value of the last input $\boldsymbol{R}$ which is fixed ($R$=4) determines the number of bits that $V_4$ is rotated

right. The number of bits that $V_1$, $V_2$ and $V_3$ are rotated right is based on the value of the first three bits of $W_2$, $W_3$ and $W_4$, respectively. The transformation function generates four N-bit long outputs.

The rotation in this algorithm has been selected as a variable one to increase the complexity of the algorithm without effecting its speed, because in the case of using a fixed rotation value, the randomisation capability would be the same. Using variable bit rotation improves the structural strength without reducing the encryption speed. This rotation provides eight different choices for bit rotation value ranging from *zero* to *seven*. There are four iterations in the DCU encryption/decryption algorithm, and each of them has an $F$ function with four variable rotations (one of them, $R$, is fixed for all the rounds). Therefore, the proposed method provides $8^{12}$ variations of bit rotation which makes a structural attack very difficult.

We can formulate the relationship between the $F$ function input and its output as follows:

$$V_1 = U_1 \odot Z'_9 = (\dot{X}_1 \oplus \dot{X}_5) \odot Z'_9 , \qquad V_2 = U_2 \boxplus V_1 = (\dot{X}_2 \oplus \dot{X}_6) \boxplus V_1$$

$$V_3 = U_3 \odot V_2 = (\dot{X}_3 \oplus \dot{X}_7) \odot V_2 , \qquad V_4 = U_4 \boxplus V_3 = (\dot{X}_4 \oplus \dot{X}_8) \boxplus V_3$$

and The function output sub-blocks are given by the following:

$$W_4 = Rol_R(V_4) \odot Z'_{10} , \qquad W_3 = Rol_{W_4}(V_3) \boxplus W_4$$

$$W_2 = Rol_{W_3}(V_2) \odot W_3 , \qquad W_1 = Rol_{W_2}(V_1) \boxplus W_2 .$$

Where $Rol_J(X)$: rotates $X$ sub-block's bits right by the value of the first three bits of $J$.

Keep in mind that all the sub-blocks $X_i$` result from mixing the input sub-blocks with key sub-blocks, which are generated from the user selected secret key. That allows us to say that designing the main transformation function **F** in this form makes each of $F$ 's output sub-block related to all input sub-blocks (plaintext) and the secret key sub-blocks (user selected secret key) in a very complicated and involved way.

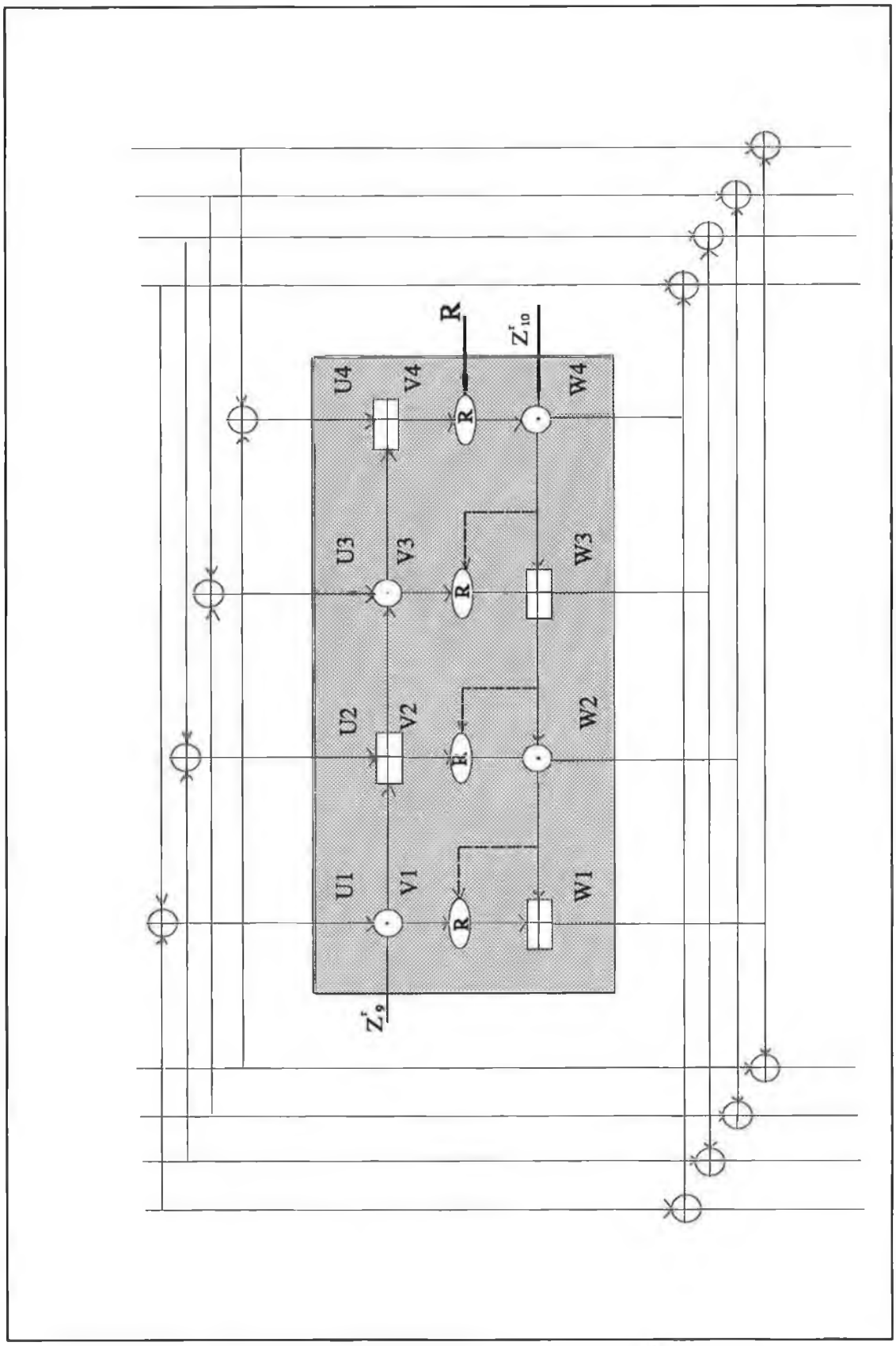



***Figure 4.2.*** *The structure of the transformation function* ***F****.*

### 4.2.4 The Key Schedule

The *DCU-Cipher* algorithm requires 48 sub-keys during the encryption procedure. Ten sub-keys are needed for each round which are distributed as following:

- Eight sub-keys effect each round's input ($Z'_1$,..., $Z'_8$).
- Two sub-keys involve in the transformation function **F** ($Z'_9$ *and* $Z'_{10}$).

this amounts to 40 sub-keys (*10* **x** *4*), and the final permutation requires another eight sub-keys which makes the total 48 sub-keys.

These sub-keys are generated from a 128-bit user selected secret key by using the key schedule shown in figure 4.3. Each round in this key schedule generates eight sub-keys each of them is 8-bits long (or four 16-bit sub-keys for *DCU128* mode, considering each of the 16-bit sub-keys is a concatenation of two subsequent 8-bit sub-keys).

When using the cipher algorithm to encrypt 128-bit blocks, 12 rounds in the key schedule is required. Only six rounds are needed to generate all the 8-bit sub-keys for *DCU64* mode. The structure of the transformation function $F_K$ is illustrated in figure 4.4. $F_K$ has two 64-bits inputs *X,Y*, each of them partitioned into eight bytes. The 8-bit sub-keys are generated as following:

$$K_1 = Rol_{X_2 \boxplus Y_2}(X_1 \boxplus Y_1), \quad K_2 = Rol_{X_3 \boxplus Y_3}(X_2 \boxplus Y_2), \quad K_3 = Rol_{X_4 \boxplus Y_4}(X_3 \boxplus Y_3),$$

$$K_4 = Rol_{X_5 \boxplus Y_5}(X_4 \boxplus Y_4), \quad K_5 = Rol_{X_6 \boxplus Y_6}(X_5 \boxplus Y_5), \quad K_6 = Rol_{X_7 \boxplus Y_7}(X_6 \boxplus Y_6),$$

$$K_7 = Rol_{X_8 \boxplus Y_8}(X_7 \boxplus Y_7), \quad K_8 = Rol_{X_1 \boxplus Y_1}(X_8 \boxplus Y_8).$$

where $Rol_i(X)$ is the rotation right of $X$ by the value of the first three bits of $i$. These sub-blocks (all the 64-bits) are used also as $Y$ input for the next round and as **X** input for the second next round as it shown in figure 4.3.

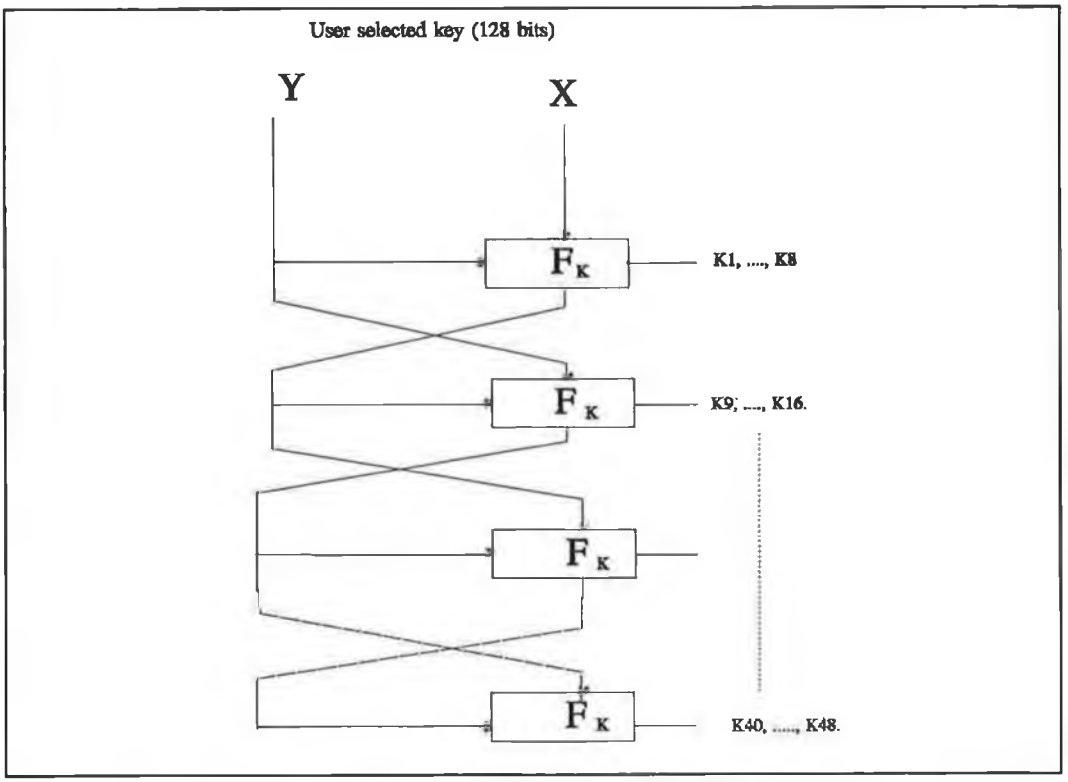



***Figure 4.3*** *The key schedule for DCU cipher*

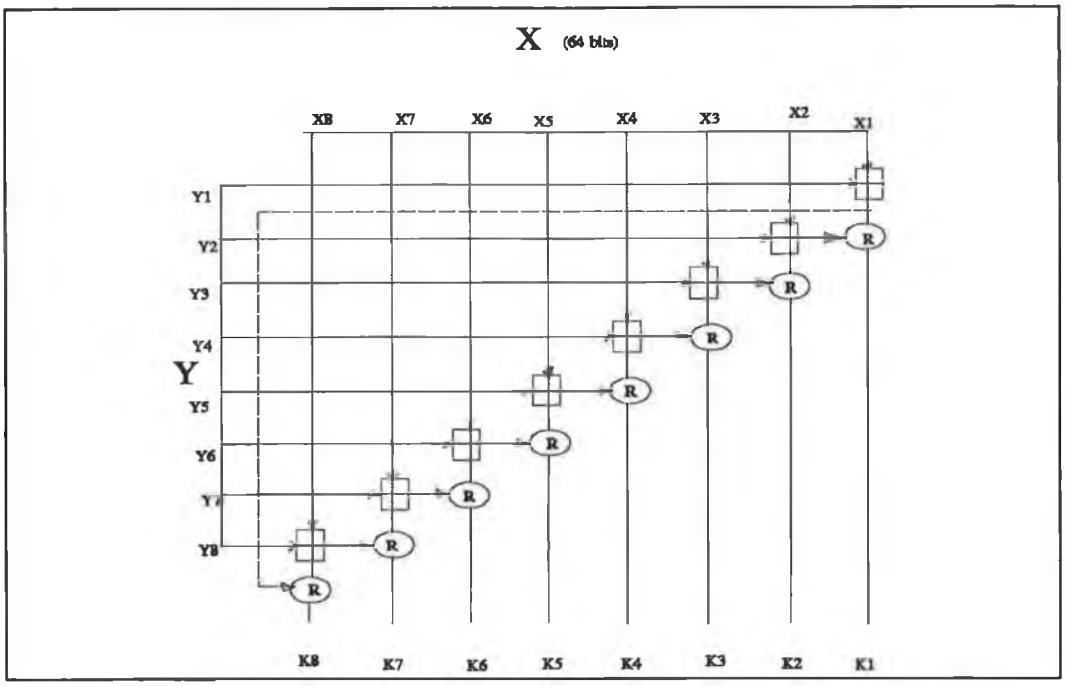



***Figure 4.4*** *The structure of the $F_K$ function*

### 4.2.5 The Decryption Algorithm:

The computational graph of the decryption process is essentially the same as that of the encryption process, figure 4.1, the only change being that the decryption key sub-blocks. The decryption sub-keys $DK_i^r$ (where $r$ indicates the current round number and $i$ the number of the sub-key within this round) are generated from the encryption sub-keys as following:

for $r = 1,5$:

$$DK_1^r = Z_1^{(6-r)^{-1}}, \quad DK_2^r = Z_2^{(6-r)^{-1}}, \quad DK_3^r = -Z_3^{(6-r)}, \quad DK_4^r = -Z_4^{(6-r)},$$

$$DK_5^r = -Z_5^{(6-r)}, \quad DK_6^r = -Z_6^{(6-r)}, \quad DK_7^r = Z_7^{(6-r)^{-1}}, \quad DK_8^r = Z_8^{(6-r)^{-1}}.$$

for $r = 2,...,4$:

$$DK_1^r = Z_1^{(6-r)^{-1}}, \quad DK_2^r = Z_2^{(6-r)^{-1}}, \quad DK_3^r = -Z_5^{(6-r)}, \quad DK_4^r = -Z_6^{(6-r)},$$

$$DK_5^r = -Z_3^{(6-r)}, \quad DK_6^r = -Z_4^{(6-r)}, \quad DK_7^r = Z_7^{(6-r)^{-1}}, \quad DK_8^r = Z_8^{(6-r)^{-1}}.$$

for $r = 1,..,4$:

$$DK_9^r = Z_9^{(5-r)}, \quad DK_{10}^r = Z_{10}^{(5-r)}.$$

where $Z^{-1}$ donates the multiplicative inverse (*modulo* $2^N+1$) of $Z$ and $-Z$ denotes the additive inverse of $Z$ (*modulo* $2^N$).

### 4.2.6 The Group Operations Characteristics

The DCU-Cipher is based on the design concept of "mixing operations form different algebraic groups having the same number of elements". These group operations, $\oplus$, $\odot$, $\boxplus$, have been chosen to provide the perfect secrecy property and the combination of these different group of operations provide the confusion required for a secure cipher (See next section).

The interaction of different operations is explained in [LM91] in terms of isotopism of quasi-groups and in terms of polynomial expressions. Recall the definitions of quasi-group and isotopic:

• *Quasi-group:*

Let $S$ be a set and let * denote an operation from pairs $(a,b)$ of elements of $S$ to an element $a*b$ of $S$. Then $(S,*)$ is said to be *quasi-group* if, for any $a,b \in S$, the equations:

$$a * x = b \quad and \quad y * a = b$$

both have exactly one solution in $S$.

The operation * in a *quasi-group* $(S,*)$ is associative, mathematically :

$$a * (b * c) = (a * b) * c$$

for all $a,b$ and $c$ in the set $S$.

• *Isotopic:*

Quasi-groups $(S_1,*_1)$, $(S_2,*_2)$ are said to isotopic (or equivalent) if there are *one-to-one* mapping $\theta$, $\phi$, $\Psi$, from $S_1$ to $S_2$, such that, for all $x,y \in S_1$,

$$\theta(x) *_2 \phi(y) = \Psi(x *_1 y).$$

Such a triple is called isotopism of $(S_1,*_1)$ upon $(S_2,*_2)$.

Let $n$ be one of the following integers 1,2,4,8 or 16 so that the integer $2^n+1$ is a prime, and let $Z_1$ denote the ring of integers modulo $2^n$ and $Z_2$ denote the ring of integers modulo $2^n+1$ and let $x, y \in Z_1$ and $X,Y \in Z_2$. Let $(Z^*_2,\odot)$ denote the multiplicative group of the non-zero elements of the field $Z_2$, let $(Z_1,+)$ denote the additive group of the ring $Z_1$, and let $(F^n_2, \oplus)$ denote the group of n-tuples of $F_2$ under the bitwise exclusive-or operation. Then the following properties have been proved by Lai and Massey [LM91]:

For $n \in \{1,2,4,8,16\}$:

**1)** Quasi-groups $(F^n_2,\oplus)$ and $(Z_1, +)$ are not isotopic for $n \geq 2$, because $(Z_1,+)$ is a cyclic group while $(F^n_2, \oplus)$ is not.

**2)** Quasi-groups $(Z^*_2 , \odot)$ and $(F^n_2,\oplus)$ are not isotopic for $n \geq 2$, that results

from being $(Z^*_2, \odot)$ and $(Z_1, +)$ isomorphic groups because both groups are cyclic. Thus, $(Z^*_2, \odot)$ is isotopic to $(F^n_2, \oplus)$ if and only if $(Z_1, +)$ is isotopic to $(F^n_2, \oplus)$.

**3)** $(\theta, \phi, \Psi)$ is isotopism of $(Z^*_2, \odot)$ upon $(Z_1, +)$ if and only if there exist constants $c_1, c_2 \in Z_1$ and a primitive element $a$ of the field $Z^*_2$ such that for all $x$ in $Z_1$:

$$\theta(x) - c_1 = \phi(x) - c_2 = \psi(x) - (c_1 + c_2) = \log_a(x)$$

That means, any isotopism between these group is essentially the logarithm. Moreover, if $(\theta, \phi, \Psi)$ is isotopism, none of these maps will be the "*mixing mapping*" $m$ from $Z^*_2$ to $Z_1$ defined by $m(i) = i$, for $i \neq 2^n$ and $m(2^n) = 0$ when $n \geq 2$.

The cryptographic significance of inhibiting isotopisms between the selected operations is that, if there were an isotopism between two operations, then one could replace one operation with other by applying bijective mapping on the inputs and on the output. The isotopism from $(Z^*_2, \odot)$ onto $(Z_1, +)$ is essentially the discrete logarithm, which is considered to be a complex function.

**4)** Under a mixing mapping $m$, multiplication modulo $2^n + 1$, which is a *bilinear* function over field $Z_2$, induces the function $G: Z_1 \times Z_1 \dashrightarrow Z_1$, over the ring $Z_1$. Similarly, under the inverse mixing mapping $m^{-1}$, addition modulo $2^n$, which is an *affine* function in each argument over the ring $Z_1$, induces a polynomial function $F(X,Y)$ over the field $Z_2$. For example, when n=1, $x, y \in Z_1$, $X, Y \in Z_2$, where $m(X) = x$ and $m(Y) = y$, we have:

$$x+y \bmod 2 \longleftrightarrow F(X,Y) = 2XY \bmod 3.$$
$$XY \bmod 3 \longleftrightarrow G(x,y) = x+y+1 \bmod 2.$$

This means, to get the same result, which is an outcome of implementing an operation on elements from a specified ring say $Z_1$, using the element's images in the other ring ($Z_2$), a function with different characteristics is required.

• For any fixed $X \neq 2^n$ (i.e.,$x \neq 0$), the function $F(X,Y)$, corresponding to addition $x+y \ mod \ 2^n$ in $Z_1$, is a polynomial in $Y$ over $Z_2$ with degree $2^n-1$. Similarly, for any fixed $Y \neq 2^n$, $F(X,Y)$ is polynomial in $X$ over $Z_2$ with degree $2^n - 1$.

• For any fixed $x \neq 0, 1$, the function $G(x,y)$, corresponding to multiplication XY mod $2^n+1$ in $Z_2$ can not be written as a polynomial in $x$ over $Z_1$. Similarly for any fixed $y \neq 0,1$, $G(x,y)$ is not a polynomial in $x$ over $Z_1$.

Therefore, under mixing mapping $m$ and its inverse $m^{-1}$, its possible to consider the operations $\odot$ and $\boxplus$ as acting on the same set (either in the ring $Z_1$ or in $Z_2$). By this consideration, we must analyze some highly non-linear function, sense that the multiplication modulo $2^n+1$, which is bilinear over $Z_2$, corresponds to a non-polynomial function over $Z_1$, and addition modulo $2^n$, which is an affine function in each argument over $Z_1$, corresponds to a two-variable polynomial of degree $2^n-1$ in each variable over $Z_2$. Thus, based on the above consideration, we can construct a non-linear transformation function $\boldsymbol{F}$ using the $\oplus$, $\odot$, $\boxplus$ operations. Using the variable rotation operation (as shown in Figure 4.2) increases the function's complexity.

### 4.2.6 Achieving the Design Requirement in DCU-Cipher

After looking at the design requirements for a secret block cipher, and representing the concept and the characteristics of mixing operations form different algebraic groups, which has been used in structuring the DCU-cipher, the question to be asked now is: "Does the DCU-Cipher achieve all the design requirements? and if so, How?".

***Confusion:***

Confusion is achieved by mixing the three different group operations, $\oplus$, $\odot$, $\boxplus$, and

using the variable rotation as well, to increase complexity of the algorithm.
The three operations have also the following attributes:

• No pair of the three operations satisfies a distributive law, mathematically:

$$x \boxplus (y \odot z) \neq (x \boxplus y) \odot (x \boxplus z).$$

$$x \oplus (y \odot z) \neq (x \oplus y) \odot (x \oplus z).$$

$$x \boxplus (y \oplus z) \neq (x \boxplus y) \oplus (x \boxplus z).$$

• No pair of the three operations satisfies an associative law. In mathematical notations:

$$x \boxplus (y \oplus z) \neq (x \boxplus y) \oplus z.$$

$$x \boxplus (y \odot z) \neq (x \boxplus y) \odot z.$$

$$x \odot (y \oplus z) \neq (x \odot y) \oplus z.$$

To gain the advantages of the non-distributive and non-associative properties of these groups as well as all the above mentioned attributes of mixed group operations, the three different group of operations are arranged in the DCU-Cipher structure in such way that none of an operation's output of one type is used as the input to an operation of the same type, as shown in figure 4.1 and 4.2. Moreover, the combination of the three operations by the mixing mapping *m*, inhibits isotopisms as we have seen in the previous discussion. Thus, using any bijections on the operands it is impossible to realize any one of the three operations by another operation. Under the mixed mapping, multiplication modulo $2^N$ +1, which is a bilinear function over $Z_2$, corresponds a *non-polynomial* function over $Z_1$. Under the inverse mixing mapping, addition modulo $2^N$, which is an affine function in each argument over $Z_1$, corresponds to a two variable *polynomial* of degree $2^N$-1 in each variable over $Z_2$, where N is

either 8 or 16 (regarding the DCU-cipher mode), $Z_1$ is the ring of integers modulo $2^N$ and $Z_2$ is the ring of integers modulo $2^N+1$.

Therefore, the ciphertext in this algorithm depends on the plaintext and the key in a very complex manner providing the required confusion.

***Diffusion***

For the DCU-Cipher, a diffusion, by which we mean that each ciphertext bit should be effected by each plaintext bit and each key bit as well, the avalanche test and the strict avalanche test have been carried out on DCU-cipher showing that, by changing one bit in the plaintext each bit of the ciphertext block has a probability of being changed is around 50%. The same effect is obtained when changing one bit of the key. Each of the ciphertext bits has a probability close to 50% of being changed (see next chapter). The results of these tests prove that the diffusion property is achieved in DCU-Cipher.

***Perfect secrecy***

The DCU-Cipher require a user selected key of 128 bits long. The selection of this key should be random and therefore all keys are equally likely to be selected. Therefore, there are $2^{128}$ different choices of the keys. The size of this key is equal to the size of the plaintext (or ciphertext) block in the DCU128 mode, while in DCU64 mode, the key size is double the size of the plaintext (or ciphertext) block. Therefore, the design condition which has been derived from the definition of perfect secrecy is achieved by the DCU-cipher structure. Moreover, perfect secrecy is achieved at the first round of DCU-cipher where there are exactly $2^{512}$ different choices of the key sub-keys ($Z_1$, ..., $Z_{10}$) for transforming the sub-blocks ($X_1$,..., $X_8$) to the sub-blocks of the next round's inputs ($X``_1$,..., $X``_8$).

***Uniquely Reversible Function with Involution Property***

The key schedule design of the DCU-cipher provides a unique inverse for each encryption function in the DCU-cipher, therefore, there is no ciphertext that could be recovered by using two different keys. The general structure of the DCU-Cipher

provides the involution property, since the same structure is used for encryption and decryption procedures. Moreover, the round structure of this algorithm provides the involution property.

***The Simplicity in the Software and Hardware Implementations***

The different group operation functions which are involved in the DCU-cipher structure, namely, bit-by-bit exclusive-or $\oplus$, modular multiplication $\odot$, modular addition $\boxplus$, and the variable rotation, are implemented on pair of sub-blocks of N-bits long, where N = 16 or 8. Therefore implementing these operations in either software or hardware is very easy, since we are dealing with 8-bits (byte) or 16-bits (word or integer value) as operands for simple arithmetic operations.

We conclude that the DCU-Cipher satisfies all the design requirements. Some randomness tests have been implemented on this algorithm and gave good results. These tests are discussed in the following chapter.

## 4.3 The Design of Encryption Modes of Operation

For encrypting long messages using our DCU-cipher system, There is a possibility of applying the well known modes discussed in chapter 2, namely: Cipher FeedBack (CFB), Output Feedback (OFB) and Cipher Block Chaining (CBC). The first two types are used for enciphering a stream of characters when characters must be treated individually in data communication protocols, and the block encryption takes place in the feedback.

As mentioned in chapter 3, error extension is present in CFB, and OFB has the property that errors in the ciphertext are simply transferred to corresponding bits of the plaintext. A known-active attack is possible on OFB mode, since an attacker knows the ciphertext/plaintext pairs can change the plaintext to anything else without immediate detection.

CBC is the most widely used chaining technique for encrypting files and data blocks that are transferred within a data communication network, in which each

ciphertext block is related to the plaintext block and the previous ciphertext block. Mathematically the encryption and decryption are given by:

$$C_i = E_k (P_i \oplus C_{i-1})$$

and

$$P_i = D_k (C_i) \oplus C_{i-1}.$$

where $E_k$ is encryption and $D_k$ is the decryption functions using the key $k$.
For the first plaintext block, there is no previous ciphertext to be ex-ored with, therefore an Initial Vector *IV*, which is a random block, is ex-ored with the first block of plaintext.

Using the exclusive-or to combine the plaintext with the previous block of ciphertext, which consists of essentially random data, thwarts a chosen plaintext attack, but it still does not prevent a known plaintext attack as mentioned by Biham and Shamir [BS92a], where they pointed out that given enough known plaintext and ciphertext pairs, differential cryptanalysis attack can still be employed.

A weakness in this classic CBC arises if a large number of plaintexts are encrypted using the same key. If using a 64-bit block size, as it common, and if many more than $2^{32}$ ciphertext blocks are generated then, as a consequence of *Birthday paradox*, pairs of identical ciphertext will occur. Knowing the plaintext associated with one block in pair, trivially reveals the plaintext associated with the other [SS93a].
If a ciphertext block is damaged in CBC chaining mode, only its plaintext and the plaintext of the next block will be effected on decryption. This is sometimes called the *self-healing* property of CBC.
Thus, differential cryptanalysis shows that the best known technique for file encryption, CBC, is perhaps not strong enough in its current structure to stand against a known plaintext attack.

To complete the work of designing a secure communication system, new encryption modes are presented here which maintain the advantages of previous methods and appear to deny the cryptanalyst any kind of known plaintext attack on the underlying block cipher or a chosen plaintext attack. These encryption modes are modifications of techniques described by Meyer and Matyas [MM82]. The first new

encryption mode of operation is called *Plaintext-Ciphertext Complex Block Chaining (PCCBC)*, while the other mode is called *Cipher Block Chaining with cross-plaintext feedforward CBC-PX.*

### 4.3.1 Meyer-Matyas Encryption Mode

Meyer and Matyas proposed a non-standard method for encrypting long messages [MM82] which considered a modification of CBC mode. This method has been also used by J.Kohl in designing version 4 of the *Kerberos* system for network authentication [Koh90]. In this mode, the ciphertext block $C_i$ is related not only to the plaintext $P_i$ and previous ciphertext block $C_{i-1}$ (as in CBC mode), but also to the previous plaintext block $P_{i-1}$, as illustrated in figure 4.5. Mathematically:

$$C_i = E_k\,[P_i \oplus f(C_{i-1}, P_{i-1})]$$

$$P_i = D_k\,[C_i] \oplus f(C_{i-1}, P_{i-1})$$

where $f(x,y)$ is the function which is represented by a triangle in figure 4.5. The first plaintext block is ex-ored by an *Initial Vector (IV)* which is a random block the same size as the plaintext block. This scheme is called sometimes *PCBC* (*Plaintext-Ciphertext Block Chaining*). This scheme does not self-heal. If a ciphertext is corrupted, the error propagates and all the subsequent decrypted plaintexts will be in error. Meyer and Matyas suggested that the function $f$ could be an exclusive-or operation in the actual implementation [MM82]. An alternative feedback function $f$, could be used to strength this mode of operation, for example, the function of multiplication modulo $2^n+1$. This multiplication operation and the exclusive-or operation are neither associative nor distributive as was shown earlier. The encryption and decryption are then achieved by:

$$C_i = E_k\,[P_i \oplus (C_{i-1} \odot P_{i-1})]$$

$$P_i = D_k\,[C_i] \oplus (C_{i-1} \odot P_{i-1})$$

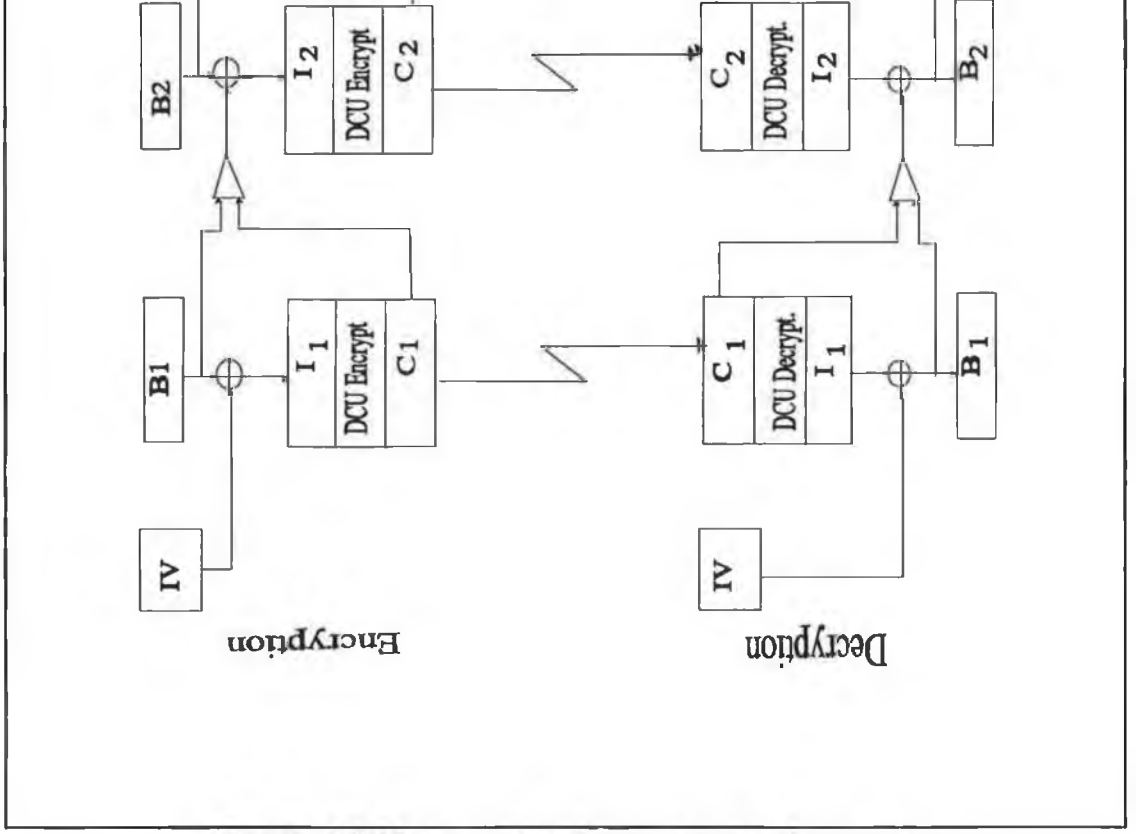
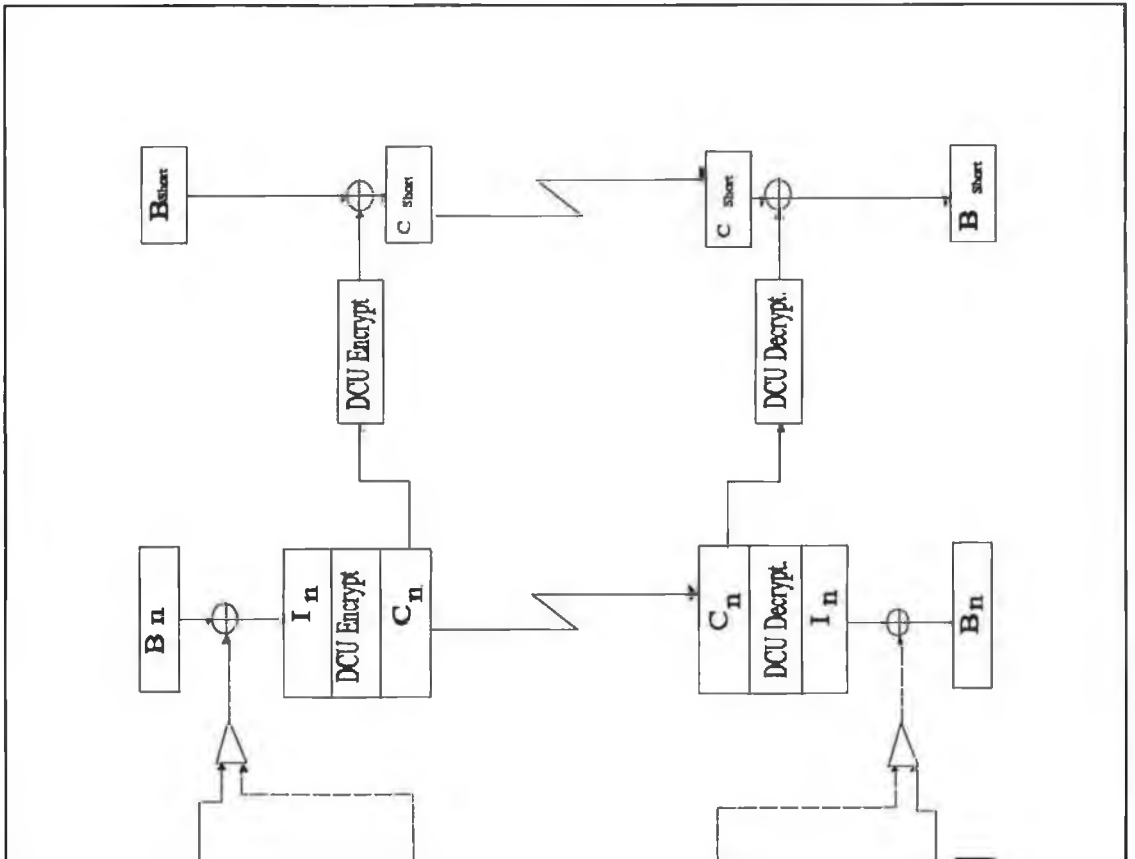



***Figure 4.5** Meyer-Matyas (PCBC) encryption mode*


B Short
C Short
C Short
B Short
DCU Encrypt
DCU Decrypt
B n
I n
DCU Encrypt
C n
C n
DCU Decrypt
I n
B n

Obviously a known-plaintext attack is still possible in this structure, if two neighbouring blocks of plaintext are known.

### 4.3.2 New Proposed Encryption Modes

Two new modes for encrypting long messages in data communication networks are discussed here. Both are modification of Meyer-Matyas method which is itself a modification of the standard method CBC. This modifications are summarised in the following points:

- Rewiring the joint point of the feedback link.
- Choosing a complex feedback function.
- Using an initial random block at the beginning of the chain.

**The First Proposed Operation Mode (PCCBC):**

This encryption mode is called *Plaintext-Ciphertext Complex Block Chaining (PCCBC)* or *CBC-P* as it is described in [SS93a]. The structure of this new mode is illustrated in figure 4.6. Each ciphertext block in this mode depends not only on the current plaintext block, current ciphertext block, and previous ciphertext, but also on all previous plaintext and ciphertext blocks. In mathematical notations:

Let $f(x,y)$ is the feedback function which denoted by a triangle in figure 4.6. The encryption and decryption are given by:

$$C_i = E_k[P_i \oplus f(C_{i-1},(P_{i-1} \oplus f(C_{i-2}, (P_{i-2} \oplus f(.....,f(P_1,IV)... ))))]$$

$$P_i = D_k[C_i] \oplus f(C_{i-1},(P_{i-1} \oplus f(C_{i-2}, (P_{i-2} \oplus f(.....,f(P_1,IV)... ))))]$$

As it appear form these formulas, each ciphertext block is a function of all previous ciphertext blocks, plaintext blocks and the initial random vector. Note that the function $f(x,y)$ is not specified yet. This function should be selected as non-associative with the exclusive-or operation $\oplus$. One option for $f(x,y)$ is the modular multiplication function. Encryption and decryption are then given by:

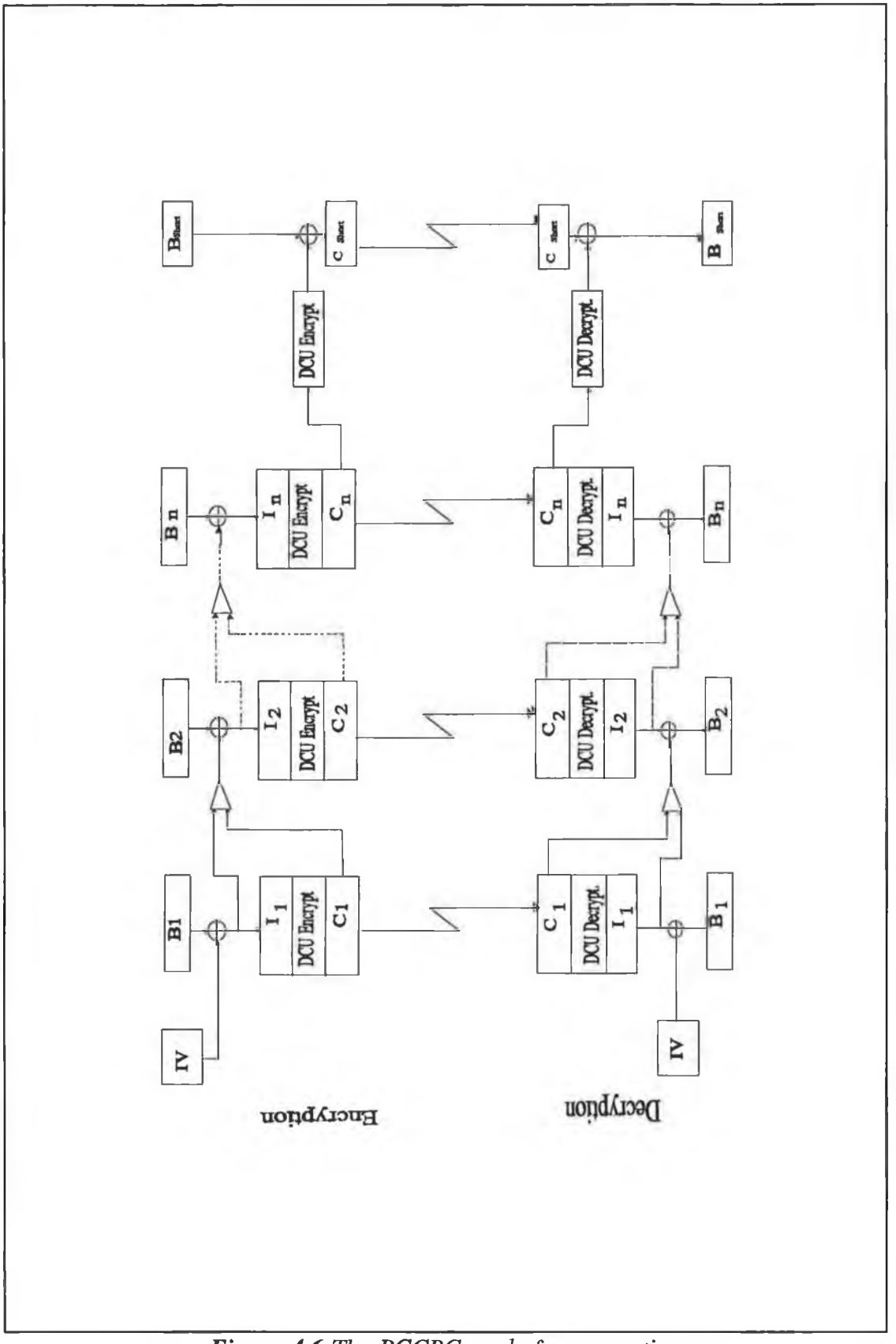



***Figure 4.6*** *The PCCBC mode for encryption*

For: $i = 1,..., n$:

$$C_i = E_k[P_i \oplus (C_{i-1} \odot (P_{i-1} \oplus (C_{i-2} \odot (P_{i-2} \oplus (.....(P_2 \oplus (C_1 \odot I_1))... )))))]$$

$$P_i = D_k[C_i] \oplus (C_{i-1} \odot (P_{i-1} \oplus (C_{i-2} \odot (P_{i-2} \oplus (.....(P_2 \oplus (C_1 \odot I_1))... )))))]$$

where $I_1 = P_1 \oplus IV$.

Another choice for the feedback function might be to use a many-to-one function. One interesting possibility is to use a mini-encryption algorithm, such as FEAL-4, as a feedback function as shown in figure 4.7. The file ciphertext block is forming the input "plaintext" block for FEAL-4 and aggregate plaintext input forming the FEAL-4 keys. All FEAL procedures are used here in the encryption mode.
FEAL-4 is a four-rounds encryption algorithm which transforms 64-bit plaintext into 64-bit ciphertext under control of a 64-bit key. The operations inside the FEAL-4's structure are byte-oriented. The input of this algorithm is ex-ored with four sub-keys (4 x 16 bits), and the output of the final round is also ex-ored with other four sub-keys generating the FEAL ciphertext block. These sub-keys are generated by a key schedule. Those ex-or operations, at the beginning and the end, are excluded in our implementation of FEAL-4 in the feedback function. Therefore, only four sub-keys are required for FEAL algorithm in such implementation. Each of these sub-keys is 16-bit long. By doing this, the need for FEAL-4 key schedule is no longer necessary, since the key can be simply divided into 4 sub-keys each of them is 16-bit long. In case of using the DCU128 mode of DCU-Cipher, the operations in FEAL-4 can be implemented as word-oriented (each operand is 16 bits long), and FEAL-4 in this case will have 128-bit input/128-bit output controlled by 128-bit key.

Its possible to include the random variable *IV* as the first plaintext block in the file encryption, forming a part of the encrypted file itself as it described in [Sco92]. When the file is decrypted it can simply be discarded.
In case of using mini-cipher (e.g.FEAL-4) as feedback function, the method appears

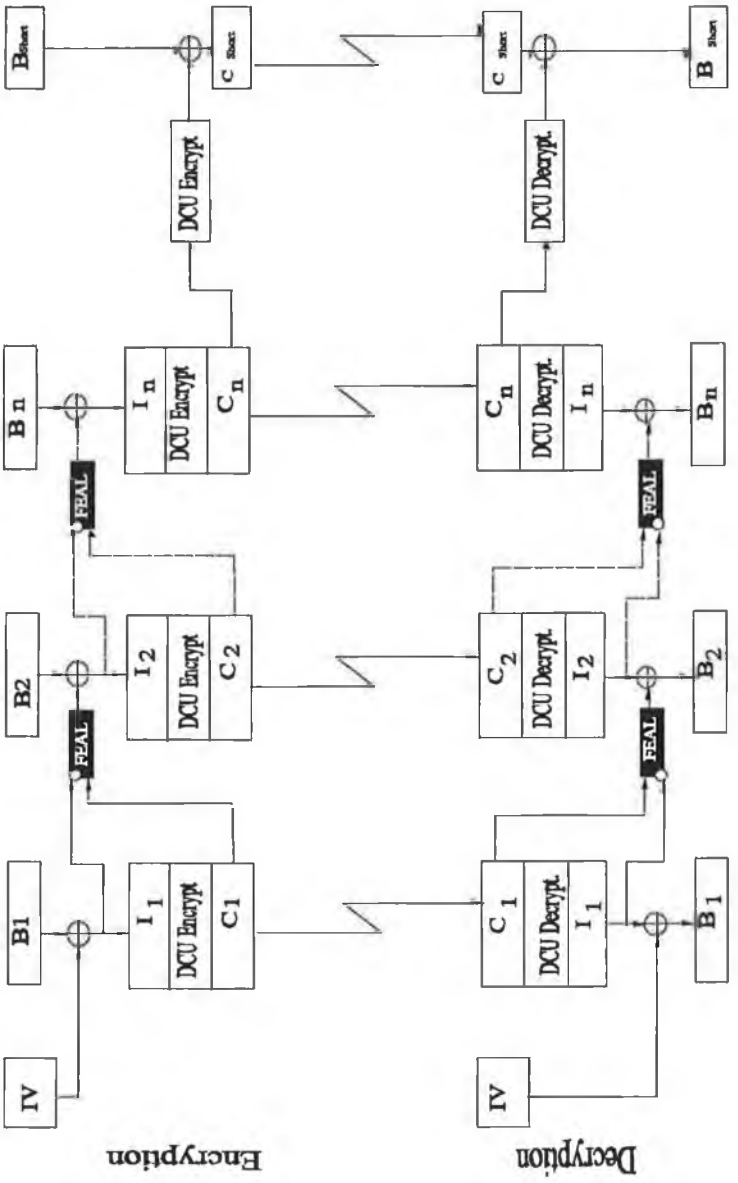



***Figure 4.7*** *The PCCBC mode with FEAL-4 Cipher as a feedback function*

to be very strong against all known attacks including the known plaintext differential cryptanalytic one, since the cryptanalyst is facing two encryption algorithms to attack. Even if the plaintext-ciphertext pairs are known, he still does not know the FEAL-4 keys (the IV value which is the key for the first FEAL-4, or any of the others keys $I_1$, ..., $I_n$, which are the results of ex-oring the plaintext with the previous FEAL's output block). The cryptanalyst does not also know the FEAL's outputs. Thus, a direct known plaintext attack is no longer possible unless all previous plaintexts are known including the initial random variable IV.

$I_1$,..., $I_n$ are the actual real inputs for DCU-encryption. Hiding this information from the cryptanalyst, changes the attack from a known plaintext attack into a ciphertext only attack which can not be launched on this structure of the encryption mode, since the DCU-cipher is a strong block cipher.

However, one might construct a different type of attack based on closed-form description of the process as a kind of known-plaintext attack as mentioned in [SS93].

If plaintext/ciphertext pairs are known, this attack will be on a back-to-back concatenation structure of a decipherment and encipherment as illustrated in figure 4.8, where the previous ciphertext block $C_{i-1}$ forms the known-plaintext, and the current ciphertext block $C_i$ the ciphertext output block:

$$C_i = E_k [P_i \oplus f (C_{i-1}, D_k [C_{i-1}])]$$

If the cipher system uses $n$ rounds, then this back-to-back structure might be considered to be, at least, more difficult to break than the same block cipher system with $2n$ rounds. Since both the encryption and the decryption modes of $n$ rounds are involved in this structure and the feedback function which could be a mini-cipher such as FEAL-4, which makes the structure more complex, this type of attack appears to be fruitless unless the cipher algorithms are weak. Note that this scheme is retains the self-healing property.

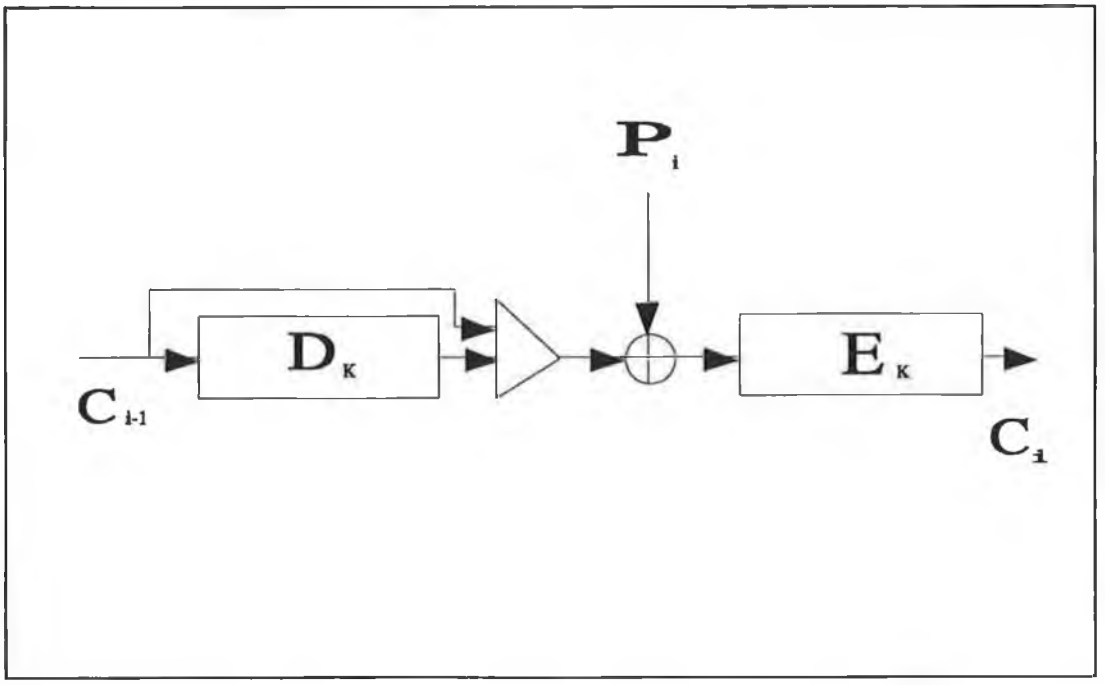



***Figure 4.8*** *Known-Plaintext attack on CBC-P mode*

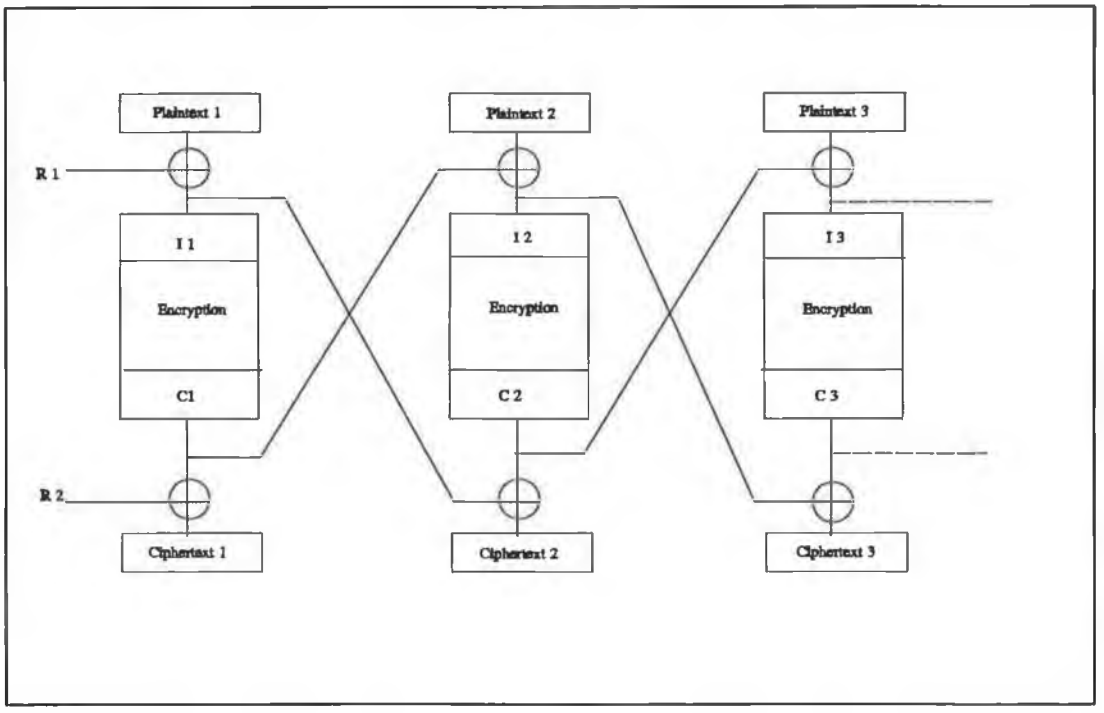



***Figure 4.9*** *The structure of CBC-X mode of operation*

**The Second Proposed Operation Mode (CBC-PX):**

To thwart the above mentioned type of a known-plaintext attack (if, by any chance, an attacker succeeds in finding a message encrypted by the above mode of operation), another type of encryption mode is considered here. This mode of operation is a modification of the CBC mode as illustrated in figure 4.9. The main modifications are:

- Re-position the joining point of the feedback branch. This makes each ciphertext block related to its plaintext block and to all previous ciphertext and plaintext blocks as well (similar to the modification which applied in the previous proposed mode, PCCBC).
- Adding a feed-forward line to the structure.
- Using two initial random variables, $R_1$ and $R_2$, at the beginning of the procedure to avoid a known / chosen plaintext attack.

This method is called CBC-X mode which provides a type of cross linking between the inputs and the outputs of the cipher algorithm in the chain. This mode of operation is error-propagating, since tampering with a ciphertext block will have quite unpredictable effects on both the current and all subsequent decrypted plaintexts.

To prevent a possible differential cryptanalytic attack, which does not require known plaintext as such but rather the exclusive-or differences between plaintexts and ciphertexts, the method can be used in combination with the previous proposed idea, yielding to the structure which illustrated in figure 4.10. We call this type of file encryption mode *PCCBC-X*, (or *CBC-PX* as it called in [SS93a]).

Again, the feedback function which is represented by triangle in the figure must be selected as non-associative function with the exclusive-or. Thus, the modular multiplication operation or a mini-cipher algorithm, such as FEAL-4, are very suitable candidates for this feedback function. In this case no closed-form description is possible and the known plaintext active attack including the differential one appears to be no longer feasible.

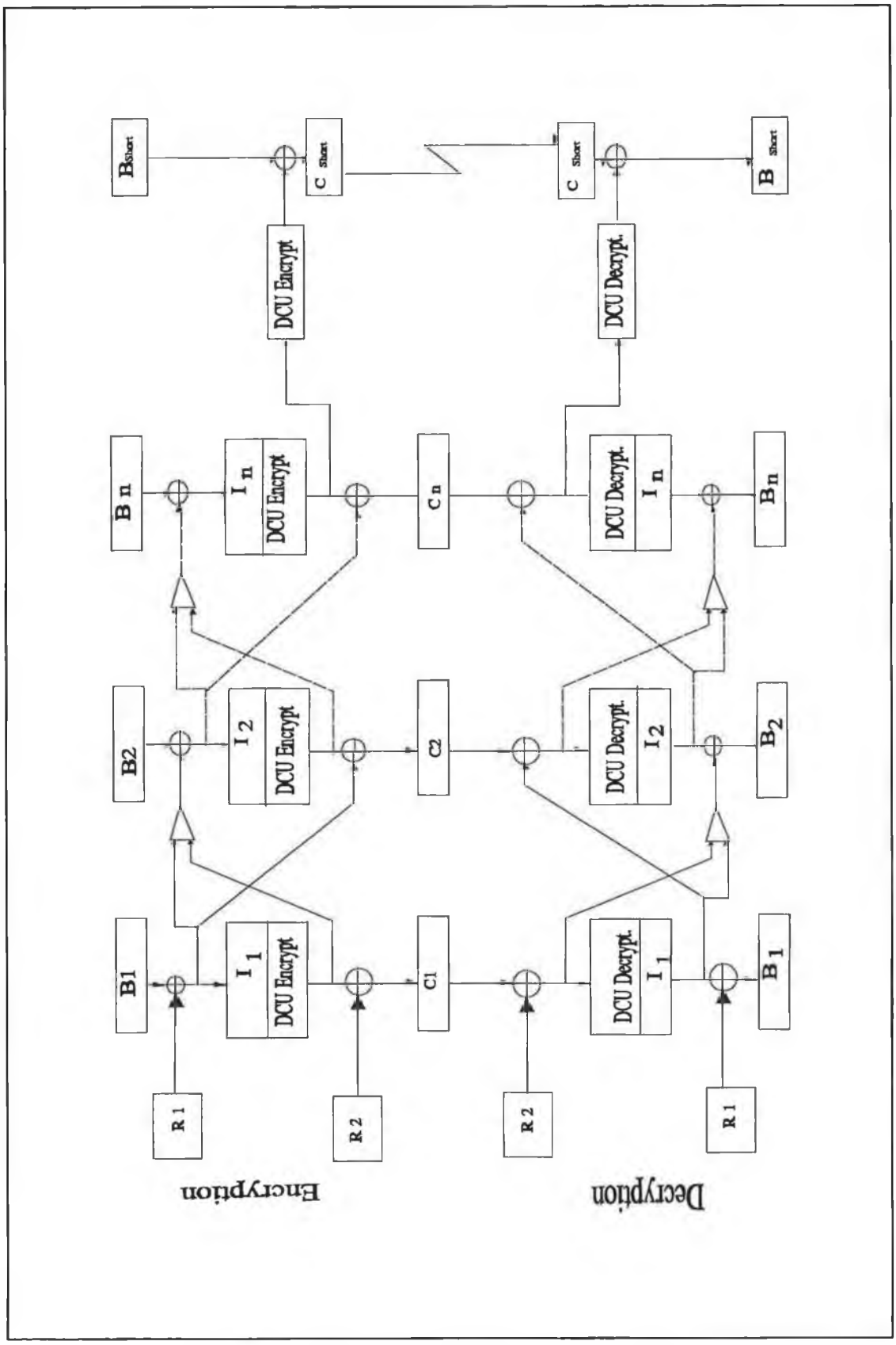



***Figure 4.10*** *The CBC-PX mode of operation*

## 4.4 Using DCU-Cipher for Message Authentication (Hashing function)

A hash function is an easily implementable mapping from the set of all binary sequences of some specified minimum length or greater to the set of binary sequences of some fixed length. In cryptographic applications, hash functions are used within digital signature schemes and within schemes which provide data integrity and authentication to detect any modification of a message.

There are large number of hash functions that have been developed and suggested for cryptographic purposes. Some of these use block ciphers like DES to produce a hash value the same size of the block cipher output. The *CBC-MAC (Cipher Block Chaining- Message Digest Code)* is the most obvious way of using block cipher to construct a message digest which based on the standard mode of chaining, *CBC*. The derived digest is simply the last ciphertext block of the chain. *CBC-MAC* was considered as a standard digest method for commercial systems such as banks [MPW92]. The problem in this method is that it gives a digest of at most $n$ bit, where $n$ is the block size of the cipher system ($n = 64$ in most of the standard/ or proposed standard block cipher methods) which is very small and easy to attack. Many attempts have been made to overcome the above problem by using a block cipher in different way, for example the *Bidirectional Message Authentication code (BMAC)* is a modification of CBC-MAC which produces a message digest of $2n$ bits. This message digest is simply a concatenation of the digest of a message $M = m_1, m_2,...m_t$, generated by *CBC-MAC* and the *CBC-MAC* message digest of the same message taking the message blocks in the reverse order (e.g. $m_t m_{t-1},...m_1$).

Given the knowledge of the cipher key $k$, *CBC-MAC* and *BMAC* are not one-way hash functions.

A different and apparently more secure hashing scheme using block ciphers was presented separately by Davies, Meyer and is referred to it as the DM scheme [MPW92] . The message in this scheme is divided into a series of fixed length block; this time, however, the block length is $k$ (the key length for the cipher block cipher)

rather than $n$.

$$M = m_1, m_2, ..., m_t$$

where $m_i$ contains $k$ bits. The hash round function is given by:

$$H_i = E_{mi}(H_{i-1}) \oplus H_{i-1}$$

for $i = 1, ..., t$. Where $E_{mi}(H_{i-1})$ is the encryption of the block $H_{i-1}$ under control of $m_i$, as a key, and $H_0 = IV$, an initial value which might be a random number. The message digest is simply the last block of this sequence $H_t$. The DM hashing scheme is illustrated in Figure 4.11.

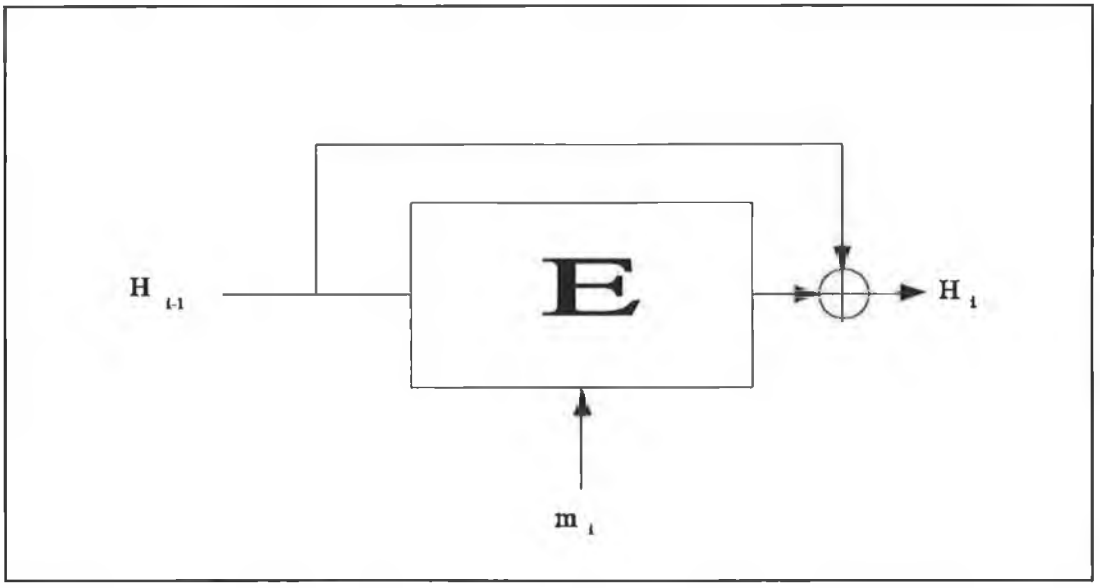



***Figure 4.11*** *The DM scheme for message digest*

Using a block cipher with 64-bit plaintext/ciphertext block and a key of 64 bits long, this method has been attacked by either brute-force, birthday or meet-in-the-middle *collision* attacks which have the complexity of $2^{32}$ [LM92].

Because of the widespread use of 64-bit block cipher and the unavailability of

the 128-bit block cipher. efforts have recently been made to modify DM scheme such as those are presented in [QG90] and [LM92]. The main goal of these modifications is to construct a *2n* hash function based on one of the *n*-bit block cipher (which has 64-bit key such as DES, FEAL, or 128-bits key block cipher such as PES or IPES) using the DM structure.

Another group of hash functions relies on modular squaring modulus a large prime such as Jueneman's methods which has been discussed in [MPW92]. There are also number of suggestions that don't match these categories, e.g. Snefru [Mer90b], N-Hash [MKO90], MD4 [RD91], MD5 [RD92] and FFT_Hash [BG91]. The newly proposed US federal Secure Hash Standard (SHS) [SHS92] is similar in the structure to MD4 and belongs to the last mentioned group too.

We can say, in general, the main deficiency of hash functions that are based on block cipher in their construction, is the short length of the generated digest, mainly 64 bits long. All the reported modifications of these methods showed that it must go through the message several times, two at least, to generate hash value with double the length of the cipher block, or the key size, of the employed block cipher.

Using the structure of DCU128 mode of DCU-cipher in hashing system fulfils the functional and security requirements of cryptographic hashing algorithm that are listed in [BD92]. It also overcomes all the above problems, since the size of the ciphertext/plaintext blocks as well as the key length is 128 bits.
Nevertheless, a size of 128 bits appears (nowadays) to be secure for most types of hash functions applications.

A good hash function is suggested here by using the DM hashing scheme, which appears to be the most secure hashing scheme based on block cipher, with MODE128 of the DCU-block cipher to generate a message digest of 128-bits long. Only one pass through the message is required in this system to produce 128-bit hash value, providing one-way collision-free hash function (i.e. Given a message $M$ and its hash value $H$, it is computationally infeasible to find another message $M_1$ with the same hash value $H$).

## 4.5 Conclusion

The design of a new block cipher algorithm is presented which has only four rounds and workable in two modes, DCU64 and DCU128. This cipher algorithm is based in its structure on the principle of mixing operations from different algebraic groups. This mixing of the group provides the perfect secrecy and the combination of the different group operations provide the confusion and the diffusion. Its structure is suitable for software and hardware implementations. The transformation function ***F*** in this algorithm has a very complicated structure, which prevents building an easy differential relationship. Therefore applying differential cryptanalysis attack on DCU-cipher appeared to be fruitless. The key size of the DCU-cipher, 128-bits, makes the minimum work factor of all brute force attacks larger than any desired value. Thus, this method appeared to be secure against all known attacks on block cipher systems. Because of the size of the secret key in the DCU-cipher, this algorithm is approximately $47 \times 10^{20}$ ($2^{128}$-$2^{56}$) times stronger than the current standard block cipher, *DES*, and about $28 \times 10^{13}$ ($2^{128}$-$2^{80}$) times stronger than the new proposed cipher, *SKIPJACK*, which has been announce recently by the United States authority.

The length of the key and the size of the plaintext/ciphertext blocks of DCU128 mode of this cipher algorithm (i.e 128- bits long) makes it a very significant candidate to be used in the construction of one-way collision-free hash function.

The design of new chaining methods for block cipher are discussed providing secure way of encrypting data of arbitrary length, that are transferred within data communication networks. These new encryption techniques thwart the known-plaintext attack as well as the differential cryptanalytic one, which have been successfully applied in attacking messages chained by the standard method *CBC*.

*Chapter 5*

# The Implementation and Tests

## 5.1 The implementation

The DCU-Cipher algorithm, for both DCU64 and DCU128 modes, has been implemented on a 25 MHz 386 IBM Personal Computer using the C programming language (*Turbo C*[1]). The operations which are involved in the structure of this cipher algorithm are either operations on 8-bit sub-blocks or on 16-bit sub-blocks. Therefore implementing such operations in software is very easy.

The most difficult part in the implementation is the multiplication modulo $(2^n+1)$. This operation has been implemented using the lemma which has been suggested by Lai and Massey in [LM91] as following:

Let *a, b* be two *n*-bit non-zero integers in the ring $2^n+1$, then:

[1] *Turbo C*: is a trade mark for a C compiler for PCs by Borland.

$$ab \bmod (2^n+1) =$$

$$\begin{cases} (ab \bmod 2^n) - (ab \; divi \; 2^n) & if \; (ab \bmod 2^n) \geq (ab \; divi \; 2^n) \\ (ab \bmod 2^n) - (ab \; divi \; 2^n) + 2^n + 1 & if \; (ab \bmod 2^n) < (ab \; divi \; 2^n) \end{cases} \quad (5.1)$$

Where *(ab div $2^n$)* denotes the quotient when ab is divided by $2^n$.

This simplifies the implementation. Note that *(ab mod $2^n$)* corresponds to the *n* least significant bits of *ab*, and *(ab div $2^n$)* is just the right-shift of *ab* by *n*-bits. Note also that *(ab mod $2^n$) = (ab div $2^n$)* implies that *ab mod ($2^n$+1) = 0*, and hence can not occur when $2^n+1$ is a prime.

The complete C-programming code of the DCU-Cipher system is listed in Appendix- A.

## 5.2 Tests

Theoretically, the best block cipher function is one which has the following features [WT86]:

- *Randomness*: The cryptographic function generates a truly random sequence of *n* bits, where *n* is the block's length.
- *Completeness*: Each ciphertext bit must depend on all of the plaintext bits and the key bits.

Beker and Piper stated in [BP82], referring to the first feature, that what is normally required for the output sequence in cryptography, is unpredictability rather than true randomness. For completeness, its also hard to find a simple Boolean expression for each ciphertext bit in term of the plaintext bits to proof that the function is complete. Alternatively, if there is at least one pair of *n*-bit plaintext vectors *X* and $X_i$ that differ only in bit *i* and *f(X)* and $f(x_i)$ differ at least in bit *j* for all

$$\{(i, j) : 1 \leq i,j \leq n)\}$$

then the function $f$ must be complete.

To measure a cryptographic function's randomness and its completeness, some statistical tests must be applied.

A statistical test $T$ for sequences of length $N$ is a function:

$$T:B^N \text{-->}\{accept, reject\} \text{ where } B = \{0,1\}$$

which divides the set $B^N$ of binary length $N$ sequences $S^N = S_1, ..., S_N$ into a small set

$$S_T = \{S^N: T(S^N) = reject\} \in B^N$$

of *"bad"* sequences and the remaining set of *"good"* sequences. The probability that the sequences that are rejected is:

$$p = |S_T|/2^N$$

and is called the *rejection rate*. In practice, $p$ should be small.

A statistical test $T$ for a reasonable sample length $N$ cannot feasibly be implemented by checking a list of set $S_T$. Instead, a statistical test $T$ is typically implemented by specifying an efficiently computable test function $f_T$ that maps the binary length $N$ sequences to the real numbers $R$:

$$f_T: B^N \text{ --> } R: S^N \text{ --> } f_T(S^N).$$

The probability distribution of the real-valued random variable $f_T(R^N)$ is determined where $R^N$ denotes a sequence of $N$ statistically independent and symmetrically distributed binary random variables. Usually $f_T$ is chosen such that $f_T(R^N)$ is distributed (approximately) according to a well-known probability distribution, most often the *normal distribution* or the *Chi-square* ($\chi^2$) distribution with $d$ degrees of freedom for some positive integer $d$. The normal distribution results when a large number of independent and identically distributed random variables are summed. The $\chi^2$ distribution with $d$ degrees of freedom results when a squares of $d$ independent and normally distributed random variables with zero mean and variance 1 are summed. *Chi-square* ($\chi^2$) test is perhaps the best known of all statistical tests for studying random data, and it is a basic method which is used in connection with many other tests. *Chi-square* test can be summarized as follows:

A fairly large number ($n$) of independent observations is made. We count the number of observations falling into each of $k$ categories and compute the quantity $\chi^2$

given as:

$$\chi^2 = \sum_{1 \le s \le k} \frac{(Y_s - np_s)^2}{np_s} \qquad (5.2)$$

Where:

$p_s$: is the probability that each observation falls into each category,

$Y_s$: is the number of observations that actually do fall into category $s$.

To decide wither the test is rejected or accepted, the value of $\chi^2$ test is compared with the standard value that is given by the $\chi^2$ statistical table.

The following tests have been implemented on the *DCU* cipher algorithm (for both DCU64 and DCU128) :

- Frequency test
- Serial test.
- Runs test.
- Universal test.
- Avalanche effect test.
- Strict avalanche criterion test.

The first four tests are statistical tests which provide a quantitative measure of randomness. These tests, in their various ways, measure the relative sequences of certain patterns of *ones* and *zeros* in a section of the sequence. A level of confidence has to be determined for these tests to decide wither a sequence is passed the test or not.

The last two tests, *avalanche tests*, measure the relationship between either the output and the input bits or the key and the output bits.

In these tests we generate non-random sequences as binary plaintexts (e.g. we select all the blocks which contain non-zero, one zero, two zeros, three zeros, and their complements. Those are total of 4162 blocks for DCU64 and 16514 blocks for

DCU128 mode). If the ciphertext is independent of the plaintext, it should appear as a random sequence. The key value was constant during all the above mentioned tests, except for the key-ciphertext avalanche effect text.

## 5.2.1 Frequency Test

The frequency test *(FT)* is the simplest randomness test which is used to determine whether a generator is biased and is based on the model $BMS_p$[2] with one parameter. The number of 1's in a random sequence $R_N = R_1, ..., R_N$ is distributed according to a *binomial distribution* which is very well approximated by the normal distribution with mean *N/2* and variance *N/4* since $E[R_i] = 1/2$ and $Var[R_i] = 1/4$ for all $1 \leq i \leq N$. Thus, the probability distribution of $f_{FT}(R^N)$ is for large enough *N* well approximated by the normal distribution with *zero* mean and variance 1.

In other words, the frequency test is a statistical test which decide between the null hypothesis,

$H_0$ : The number of *zeros* and *ones* in the output sequence of the cryptographic function are *equal.*

and the alternate hypothesis,

$H_1$ : The number of *zeros* and *ones* in the output sequence of the cryptographic function are *different.*

Suppose that sequences have length *n* (e.g. in *DCU* cipher *n* is either *64* or *128* bits). Let $n_0$ and $n_1$ be the number of zeros and ones respectively in the sequence. To accept the null hypothesis or reject it, the $\chi^2$ test is applied as follows:

$$\chi^2 = \frac{(n_0 - n_1)^2}{n} \qquad (5.3)$$

Clearly, if $n_0 = n_1$ always then $\chi^2 = 0$ and the larger the value of $\chi^2$ the greater the discrepancy between the observed and the expected frequencies. To decide

[2]$BMS_p$: is Binary Memoryless Source model of a bit generator, which outputs statistically independent and identically distributed binary random variables and is characterized by a single parameter, *p* denotes the probability of emitting 1's.

if the value obtained is good enough for the sequence to pass, we have merely to compare our value with a table of the $\chi^2$ distribution, for one degree of freedom.

The results of the frequency tests are illustrated in the table 5.1, where it shows the rejection rate of the frequency test with levels of significance $\alpha$ =0.01 and $\alpha$=0.05 for both DCU64 and DCU128 modes of the DCU cipher algorithm. The Result of this test is also presented in Figure 5.1, where the histogram represents the observed values while the expected values are represented by the line graph.

| Cipher | $\alpha$ = 5 % | $\alpha$ = 1 % |
|---|---|---|
| | % of $\chi^2$> 3.84 | % of $\chi^2$ > 6.63 |
| DCU-128 | 4.257 | 0.968 |
| DCU-64 | 3.363 | 0.816 |

***Table 5.1*** *The results of the frequency test on DCU-cipher*

## 5.2.2 Serial Test

The serial test is another statistical test which used to ensure that the transition probabilities are reasonable; i.e. the null hypothesis:

$H_0$: The probability of consecutive entries being *equal* or *different* is about the *same*.

and the alternative hypothesis:

$H_1$: The probability of consecutive entries being *equal* or *different* is about *different*.

This test gives some level of confidence that each bit is independent of its predecessor. Let

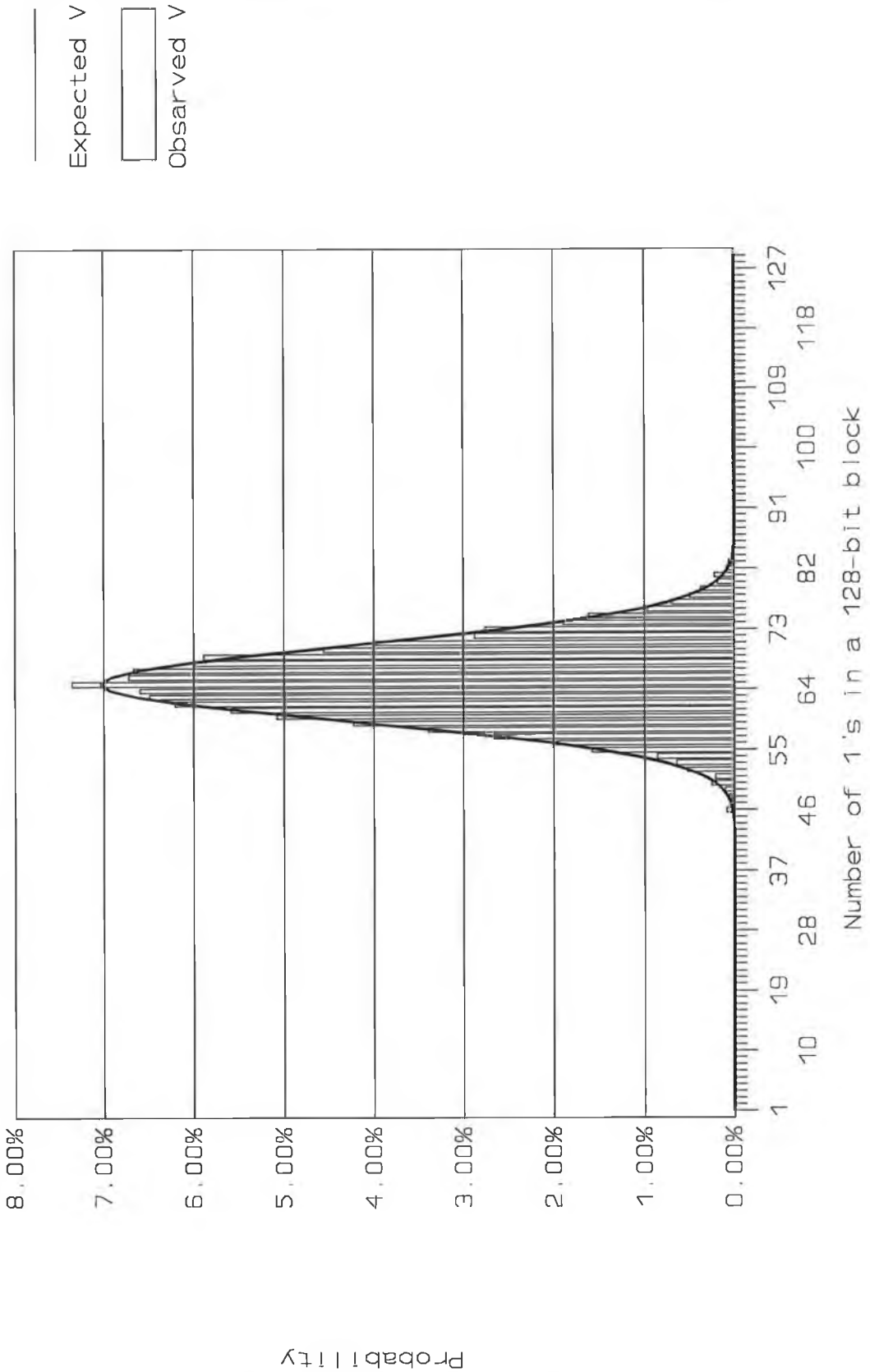



***Figure 5.1*** *The frequency test's results*

$n_{00}$ be the number of *00* entries

$n_{01}$ be the number of *01* entries.

$n_{10}$ be the number of *10* entries.

$n_{11}$ be the number of *11* entries.

Ideally we want $n_{00} = n_{11} = n_{10} = n_{01} = (n-1)/4$. The $\chi^2$ distribution for two degree of freedom is given in [BP82] by the following formula:

$$\chi^2 = \frac{4}{n-1}\sum_{i=0}^{1}\sum_{j=0}^{1}(n_{i,j})^2 - \frac{2}{n}\sum_{i=0}^{1}(n_i)^2+1 \qquad (5.4)$$

The following table, table 5.2, shows the rejection rate of the serial test with levels of significance $\alpha$ =0.01 and $\alpha$=0.05 for both *DCU64* and *DCU128* of the *DCU-cipher* algorithm.

| Cipher | $\alpha$ = 5% | $\alpha$ = 1% |
|---|---|---|
| | % of $\chi^2$>5.99 | %of $\chi^2$>9.21 |
| DCU-128 | 4.868 | 1.04 |
| DCU-64 | 5.141 | 1.057 |

***Table 5.2*** *The results of the serial test on DCU-cipher*

## 5.2.3 Runs Test

If $S_t$ is any binary sequence then a *run* is a string of consecutive identical sequence elements which is neither proceeded nor succeeded by that same symbol. A run of zeros is called *gap* while a run of ones is a *block* [BP82].

For the runs test we divide the sequence into *blocks* and *gaps*. Let $n_{0i}$ be the number of *gaps* of length *i* and $n_{1i}$ be the number of *blocks* of length *i*. If $n_0$ and $n_1$

are the number of *gaps* and *blocks* respectively, then

$$n_0 = \sum_{i=1}^{i=n} n_{0i} \quad , \quad n_1 = \sum_{i=1}^{i=n} n_{1i} \tag{5.5}$$

This test is only applied if the sequences has already passed the serial test in which case the number of gaps and blocks are within acceptable limits. From Golomb's postulate [BP82], we expect about half of the gaps (or blocks) to have the length 1, quarter to have length 2 and so on.

The number of runs is normally distributed with

$$Mean = 1 + \frac{2\, n_0\, n_1}{n} \tag{5.6}$$

$$Variance = \frac{(Mean - 1)(Mean - 2)}{n - 1} \tag{5.7}$$

$$z = \frac{Runs - Means}{\sqrt{Variance}} \tag{5.8}$$

Table 5.3 shows the percentage of the rejected values of runs test for levels of significance $\alpha = 0.01$ and $\alpha = 0.05$.

| Cipher | $\alpha = 5\%$ | $\alpha = 1\%$ |
|---|---|---|
| | % of $-1.96 \geq Z \geq +1.96$ | % of $-2.575 \geq Z \geq +2.575$ |
| DCU-128 | 5.104 | 1.017 |
| DCU-64 | 5.23 | 0.913 |

***Table 5.3*** *The runs test results*

## 5.2.3 The Universal Test

The Universal test is a new statistical test for random bit generators introduced in 1992 by U. Maurer [Mau92]. This test is universal in the sense that it can detect any significant deviation of a device's output statistics from the statistics of a *truly* random bit source when a device can be modeled as an ergodic stationary source[3] with finite memory but arbitrary state transition probabilities. The test hence measure the cryptographic badness of a device's possible defect. The main advantage of this test over the previous tests is, its able to detect any one of the very general class of statistical defects that can be modeled by an ergodic stationary source with finite memory, which includes all those detected by the tests applied in the previous sections.

The Universal test (UT) is specified by the three positive integer valued parameters L, Q and K. To perform the test *UT*, each output sequence of the cipher system, ciphertext block, is partitioned into eight sub-blocks of length L (e.g. L =N/8 where N is the size of the ciphertext block. In our case L =8 or 16). The first bit of these sub-blocks are collected together forming one byte as a generated random number.

The algorithm of this test tried to find the occurrence of each of the eight-bit value. if a value does not generate during the Q times, the cipher algorithm will be considered as a bad bit-random generator. Otherwise, the test will run for K times to check if there is a cycle. Therefore the total length of the sample sequence $s^N$ is N= (Q+K)L bits, where K is the number of steps of the test and Q is the number of initialization steps. Let

$$b_n(s^N) = [s_{L(n-1)+1}, \ldots, s_{Ln}]$$

for $1 \leq n \leq Q+k$ denote the *n*th block of the length L of the sample sequence $s^N = s_1,\ldots,s_N$. For $n = Q+1, \ldots, Q+K$, the sequence is scanned for the most recent

[3] A random process generating *x(t)* is *ergodic* if and only if the probability associated with every stationary sub-ensemble is either 0 or 1. This process has the property that the *t* average of every measurable function $f[x(t_1), \ldots, x(t_n)]$ equal its ensemble average with probability of *one*.

occurrence of the block $b_n(s^N)$,i.e., the last positive integer $i \leq n$ is determined such that $b_n(s^N) = b_{n-i}(s^N)$. Let the integer-valued quantity $A_n(s_N)$ be defined as taking on the value $i$ if the block $b_n(s^N)$ has previously occurred and otherwise $A_n(s_N) = n$.
The test function is defined as the average of the logarithm to the base 2 of the K terms $A_{Q+1}(s^N)$, ..., $A_{Q+K}(s^N)$. Formally:

$$f_{T_U}(s^N) = \frac{1}{K} \sum_{n=Q+1}^{Q+K} \log_2 A_n(s^N) \qquad (5.9)$$

where, for $Q+1 \leq n \leq Q+K$, $A_n(s^N)$ is defined by

$$A_n(S^N) = \begin{cases} n & \text{if there exist no positive } i \leq n \\ & \text{such that } b_n(S^N) = b_{n-i}(S^N) \\ \min(i : i \geq 1,\ b_n(S^N) = b_{n-i}(S^N)) & \text{Otherwise} \end{cases} \qquad (5.10)$$

Rather than scanning the previous blocks for the most recent occurrence of the block, for every $n$, the test $UT$ can be implemented much more efficiently by using a table of size $V = 2^L$ that stores for each L-bit block the time index of its most recent occurrence.

The following is the pseudo-code for the Universal test ($UT$) algorithm:

```
UNIVERSAL_T( )
Begin
 L = 8, V= 2^L
 Q = 10000, K = 100000L                    /* The total number of tests Q+L */
 MEAN = 7.1836656                          /* The Mean Value        */
 DEV = 1.5*SQRT(3.238 / K)                 /* The deviation value*/
 TAB: array of size V                      /* The table  */
 i, n :          integer
 Sum, FTU:       real
 Begin
        Repeat For i = 0 to V
                TAB[i]= -1                 /* Initialization */
        Repeat For n = 0 to n =Q
                TAB[GEN()] = n                  /* Initialization, where GEN() is a
                                                   random bit generator(the DCU-cipher
                                                   encryption function with packing the
                                                   first bit of each of the 8 output sub-
                                                   blocks into byte */
        Repeat For i = 0 to V
                Begin
                        If (TAB[i]= -1)
                        PRINT (" This is a BAD random generator")
                        Exit.
                End
        Sum = 0.0
        Repeat for n = Q to n= Q+K-1
        Begin                              /* Scan byte sequence  */
                i = GEN()
                Sum = Sum +ln(n-TAB[i])
                TAB[i] = n
        End
        FTU = (Sum / K) /ln (2.0)
        IF (FTU>(MEAN + DEV) or (FTU<(MEAN + DEV))
                        PRINT ("This is a BAD random generator")
        ELSE
                        PRINT ("This is a GOOD random generator")
```

*End*

*End.*

## 5.2.1 Avalanche Test

For a given transformation, to exhibit the avalanche effect, an *average* of *one half* of the output bits should change whenever a *single* input bit is complemented.

In order to determine whether a given cryptographic function *f* satisfies this requirement, the $2^n$ plaintext vectors, $P_r$, for $r = 1,... n$, must be divided into $2^{n-1}$ pairs $p_r$ and $p_r^j$ such as $p_r^j$ be plaintext vectors that differ from $P_r$ only in the *jth* coordinate. For a fixed key, let $c_r$ and $c_r^j$ be ciphertext vectors that result from $P_r$ and $p_r^j$ respectively. Define avalanche vectors $V_r^j = c_r \oplus c_r^j$ where '⊕' is exclusive-or addition. If this procedure is repeated for all $j$ such that $j = 1,...,n$, and *one half* of the avalanche variables (bits) are equal to 1 for each $j$, then the function $f$ has a good avalanche effect.

In our case $n$, the size of the plaintext/ciphertext block, is either 64 bits or 128 bits, the number of plaintext vectors are too large (especially when n=128). So we have implemented this test by taking 1000 random sample of plaintext vectors $P_r$ and for each value of $r$ we calculate all the avalanche vectors $v_r^j$ .

The avalanche effect test has been carried out on our cryptographic function $f(p_r,k)= Cip(p_r,OUT,KEY, k)$ where, $p_r$ is a random selected plaintext vector, *OUT* is the corresponding encrypted block, *KEY* is a fixed key block with 128-bits long and $k = 1,...,8$ is the number of the rounds in the DCU-Cipher encryption procedure *Cip()*.

The test on one-round *DCU*-cipher shows that on average 43% of the output bits are changed when an input bit is complemented, while the resulted test of avalanche effect on the DCU cipher with two or more rounds showed that on average around 50% (*one half*) of the output bits have been changed when only one input bit is complemented. In other words, DCU cipher function reaches the good avalanche effect requirement after two rounds only. The graph in Figure 5.2 illustrates the avalanche effect test on the *DCU* cipher. The following is the pseudo code for the Avalanche test:

```
AVALANCHE-PC( )
Begin
 Bk1, Bk2, OUT1, OUT2, AVL   : array of size N/8.
 Binary                      : array of size N.        /* N is the block size in Bits*/
 K, N, Bit-no, R-no          : integer
 Blck_AVAL, T_AVAL           : array of size 8.   /* For storing the results of each round*/

 Repeat K times
        Begin
        Getblock( Bk1)                              /* Get a random block text Bk1 */
        Repeat N times                              /* repeat N times (the block's size)*/
                Begin
                Gblock1(BK1,BK2, N)          /* generate BK2 that differs in 1-bit from BK1,
                                                the N bit.                          */
                Repeat for R-no = 1 TO 8.
                        Begin
                          Cip(Bk1,OUT1,Key,R-no)     /* Encrypt the block Bk1      */
                          Cip(BK2,OUT2,Key,R-no)     /* Encrypt the block Bk2      */

                          AVL=(OUT1 ⊕ OUT2)          /* Calculate avalanche vector for
                                                     OUT1 and OUT2 */
                         Bin-Rep(AVL,Binary)         /* Generate the AVL's binary code */
                          Blck_AVAL[R-no] =Blck_AVAL[R-no]+ Σi Binary[i]
                                                     /* Number of 1's in AVL vector using
                                                     Cip with R-no rounds */
                        End
                End
         Blck_AVAL = Blck_AVAL / N
         T_AVAL = Blck_AVAL + T_AVAL.
        Blck_AVAL = 0                               /* Reset Blck_AVAL           */
        End
  T_AVAL = T_AVAL / k                               /* The Avalanche effect results */
End.
```

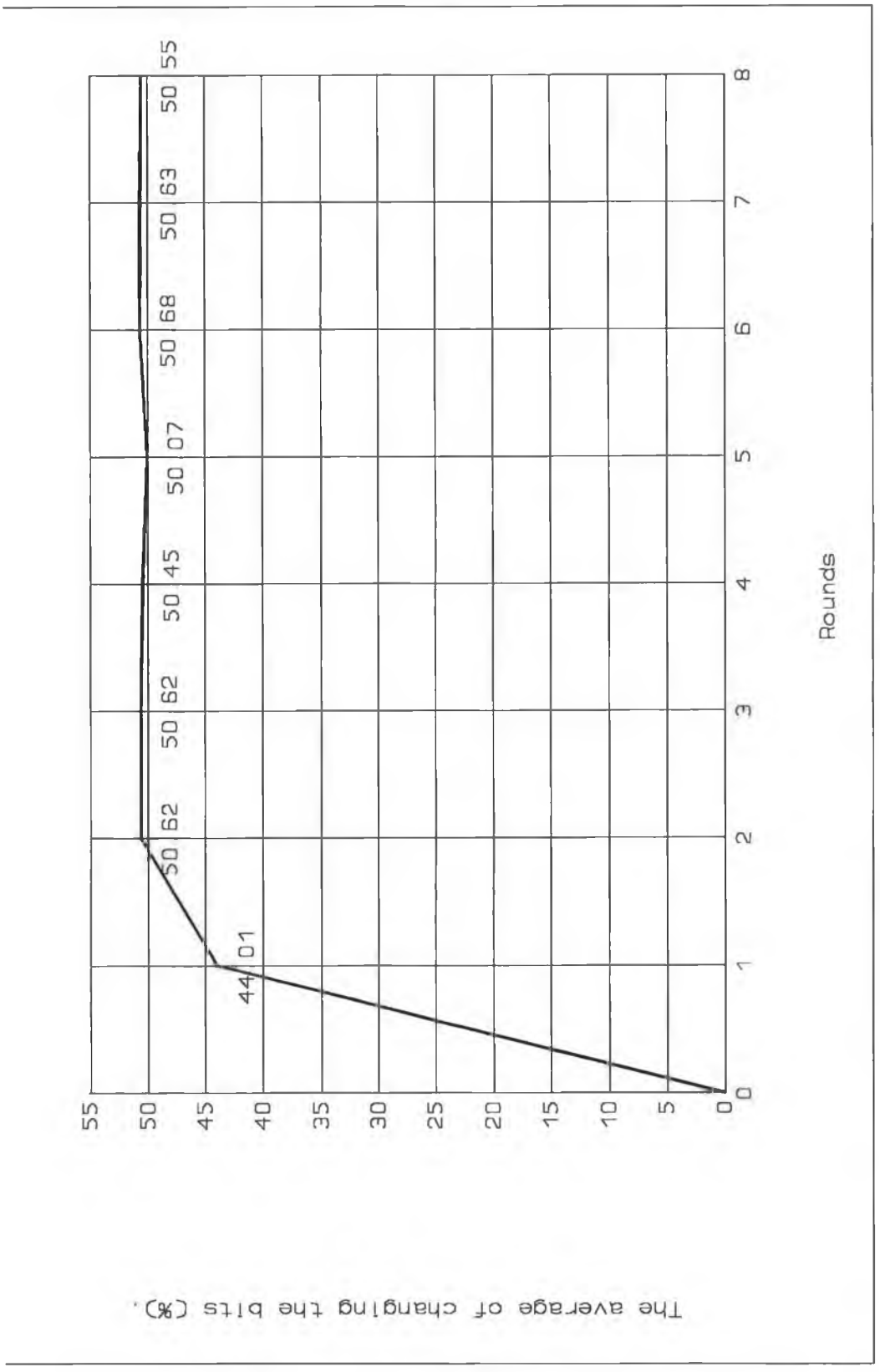



***Figure 5.2. Avalanche test in changing the plaintext***

Comparing the results of implementing the avalanche test in our cipher system, with the results of the same test on other block cipher systems evaluates the advantages of the DCU-cipher structure.

The designer of CALC ran this test on their cipher system and showed that CALC cipher need *eight* rounds to reach a steady value of avalanche effect [OT92].

FEAL-8 reaches a steady value of avalanche effect, 50% of the output bits are changed whenever one input bit is complemented, after *four* rounds, while DES needs at least *five* rounds.

### 5.2.2 Strict Avalanche Criterion test (SAC)

Webster and Tavares introduced the concept of the *strict avalanche criterion* as a combination of the completeness and the avalanche effect concepts [WT86]. If a cryptography function is satisfy the strict avalanche criterion, then *each* output bit should change with probability of *one half* whenever a *single* input bit is complemented.

There are two types of strict avalanche effects which can be examined:

I) The plaintext-ciphertext avalanche effect.

II) The key-ciphertext avalanche effect.

#### *5.2.2.1 Plaintext-Ciphertext Avalanche Effect*

A block cipher satisfies the *plaintext-ciphertext strict avalanche effect* if each ciphertext bit changes with probability of *one half* whenever a single plaintext is changed.

To measure the plaintext-ciphertext strict avalanche effect for a block cipher of length $n$ generate a large number of random plaintext vectors, $P_r$, for $r = 1,... k$. Let $p_r^j$ for $j = 1,...,n$ be plaintext vectors that differ from $P_r$ in the *jth* coordinate. For a fixed key, let $c_r$ and $c_r^j$ be ciphertext vectors that result from $P_r$ and $p_r^j$ respectively. Define avalanche vectors $V_r^j = c_r \oplus c_r^j$ for $r = 1,...,k$ and $j = 1,...,n$ where '$\oplus$' is exclusive-or addition.

Define an $n$ x $n$ dependence matrix, $D$, as follows:

for $r = 1,...,K$ add the $n$ entries of $V_r^j$ to each corresponding entry in column $j$ of $D$, where the initial values of $D$ are all zero. These entries of $D$, $d_{i,j}$, where $i,j = 1,...,n$, will give the total number of *ones* for each ciphertext bit corresponding to each of the avalanche vectors for all plaintext strings. The entries refer to total number of changes in the ciphertext position $i$ when each bit $j$ is changes in the plaintext string, for all $m$ plaintext strings.

The dependence matrix $D$ for the plaintext-ciphertext avalanche effect can be used to decide whether a block cipher is *complete* and is *non-affine* in relation to the key used to define the dependence matrix. A block cipher is said to be *complete* if each ciphertext bit depends on *all* the plaintext bits [Koh86]. Clearly a non-zero entry $d_{i,j}$ in the matrix $D$ indicates that ciphertext bit $i$ depends on plaintext bit $j$. As shown in [Koh86] if a block cipher is complete then the cipher is non-affine.

The following is the plaintext-ciphertext Strict Avalanche effect test algorithm:

```
STRICT-AVALANCHE-PC()
Begin
 Bk1, Bk2, OUT1, OUT2, AVL   : array of size N/8.
 Binary                      : array of size N.
 K, N,                       : integer
 Depend:                     : array of size N x N.

 Repeat K times
        Getblock( Bk1)                          /* Get a random block text Bk1 */
        Cip(Bk1,OUT1,Key)                       /* Encrypt the block Bk1       */
        Repeat N times                          /* repeat N times (the block's size)*/
              Begin
              Gblock1(BK1,BK2, N)       /* generate BK2 that differs in 1-bit from BK1,
                                           the N bit.                  */
              Cip(BK2,OUT2,Key)                 /* Encrypt the block Bk2       */

              AVL=(OUT1 ⊕ OUT2)         /*  Calculate the strict avalanche vector for
                                           OUT1 and OUT2          */
              Bin-Rep(AVL,Binary)       /* Generate the binary code of AVL     */
```

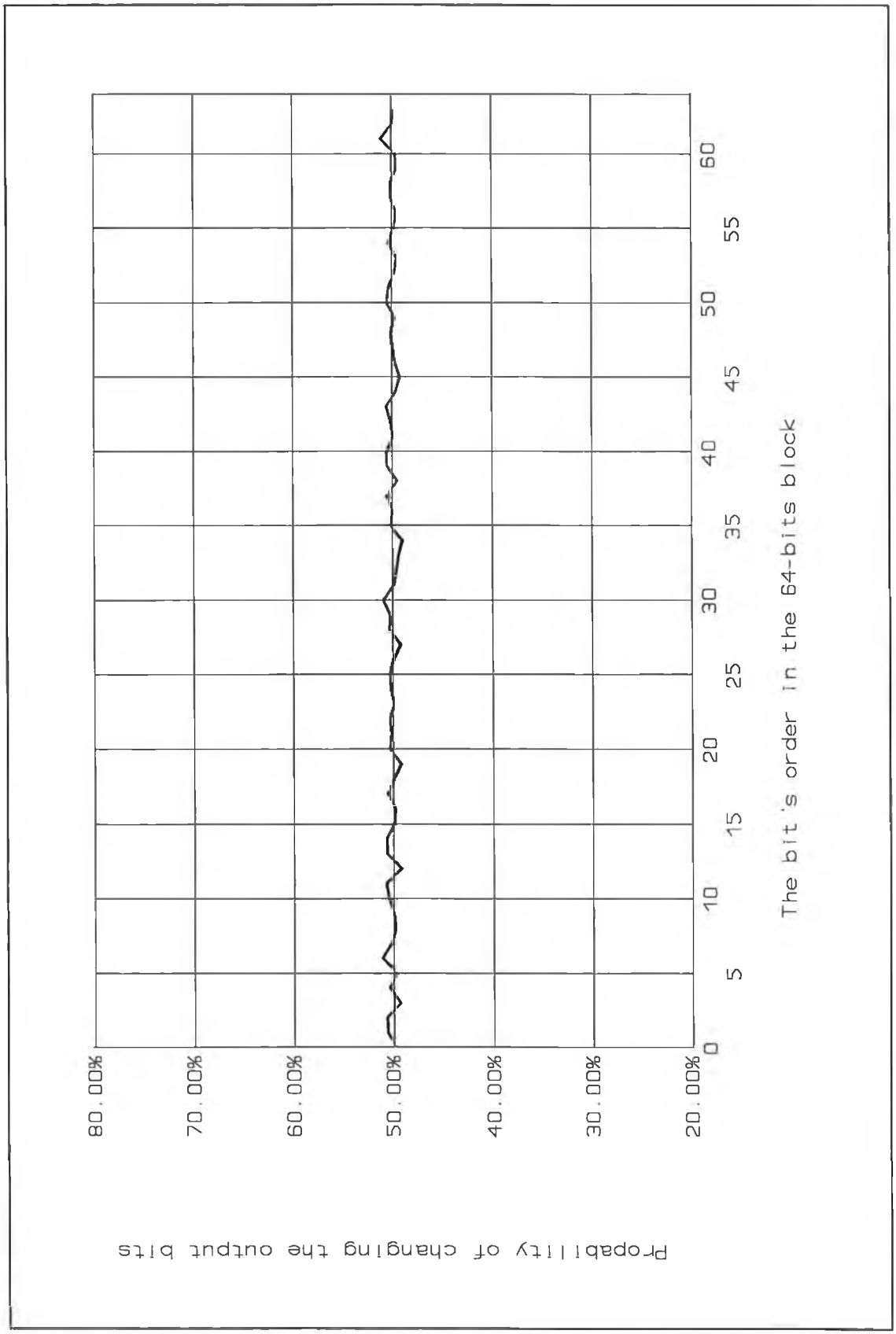



***Figure 5.3.*** *Strict Avalanche test's results*

```
                Depend[ ][N] = Depend [ ][N] + Binary   /* Add the entries of Binary to the
                                                           corresponding entries of Depend in
                                                           column N */
                End
        End
End.
```

Figure 5.3 illustrates graphically the results of implementing the strict avalanche test on the DCU128 mode of the cipher. The results of implementing this test on DCU64 gives approximately the same results. This figure shows that, changing a bit in a plaintext each output bit is changed with probability around 50%.

The dependency matrix D of the DCU cipher with 4-rounds, is illustrated in the appendix D. It appears from the dependency matrix that all the entries have non-zero values which indicate that the DCU-cipher is complete and non-affine function.

#### *5.2.2.2 Key-Ciphertext Avalanche Effect*

A block cipher satisfies the *key-ciphertext strict avalanche effect* if each ciphertext bit changes with probability of *one half* whenever a single key bit is changed.

To measure the key-ciphertext strict avalanche effect for a block cipher of length $n$ generate a large number of random key vectors $k_r$ for $r = 1,...,m$. Let $k_r^j$ for $j = 1,...,l$ be key vectors (where the key length is $l$, e.g. $l = 128$ in the *DCU* cipher) that differ from $k_r$ in the *jth* coordinate. Encrypt a fixed plaintext string $P$ and let $c_r$ and $c_r^j$ be ciphertext vectors that result from $k_r$ and $k_r^j$ respectively. Define avalanche vectors $V_r^j = c_r \oplus c_r^j$ for $r = 1,..,m$ and $j = 1,...,l$, where '$\oplus$' is the exclusive-or addition. Obtain the dependence matrix $A$ as defined in the previous section where in this case $A$ is an $n \times l$ matrix. The entries of $A$ refer to the total number of changes in the ciphertext position $i$ when each bit $j$ is changed in the key string, for all $m$ key strings.

In general a block cipher should satisfy a key-ciphertext complete property in

that every ciphertext bit should depend on every key bit. *A* non-zero entry $a_{i,j}$ in the dependence matrix for the key-cipher effect indicates that the ciphertext bit *i* depends on key bit *j*.

The following is the pseudo code for the key-ciphertext strict avalanche effect:

```
STRICT-AVALANCHE-KC()
Begin
 PLAIN, OUT1, OUT2, AVL      : array of size N/8.
 Binary                      : array of size N.
 K, N                        : integer.
 Depend:                     : array of size N x 128.
 KEY1, KEY2                  : array of size 128.

 Repeat K times
        Begin
        Getblock(KEY1)                          /* Get a random key block KEY1 */
        Cip(PLAIN,OUT1,KEY1)                            /* Encrypt the PLAIN using
                                                        KEY1 key*/
        Repeat N=1 TO 128                       /* repeat 128 times (the key size)*/
                Begin
                Gblock1(KEY1,KEY2, N)           /* generate KEY2 differs in 1-bit from
                                                KEY1, in the N bit              */
                Cip(PLAIN,OUT2,KEY2)            /* Encrypt PLAIN using KEY2    */

                AVL=(OUT1 ⊕ OUT2)               /*  Calculate the strict avalanche
                                                vector for OUT1 and OUT2        */
                Bin-Rep(AVL,Binary)             /* Generate the binary code of AVL*/
                Depend[ ][N] = Depend [ ][N] + Binary  /* Add the entries of Binary to the
                                                corresponding entries of Depend in
                                                column N*/

                End
        End
End.
```

Figure 5.4 shows the results of the key-plaintext strict avalanche effect.

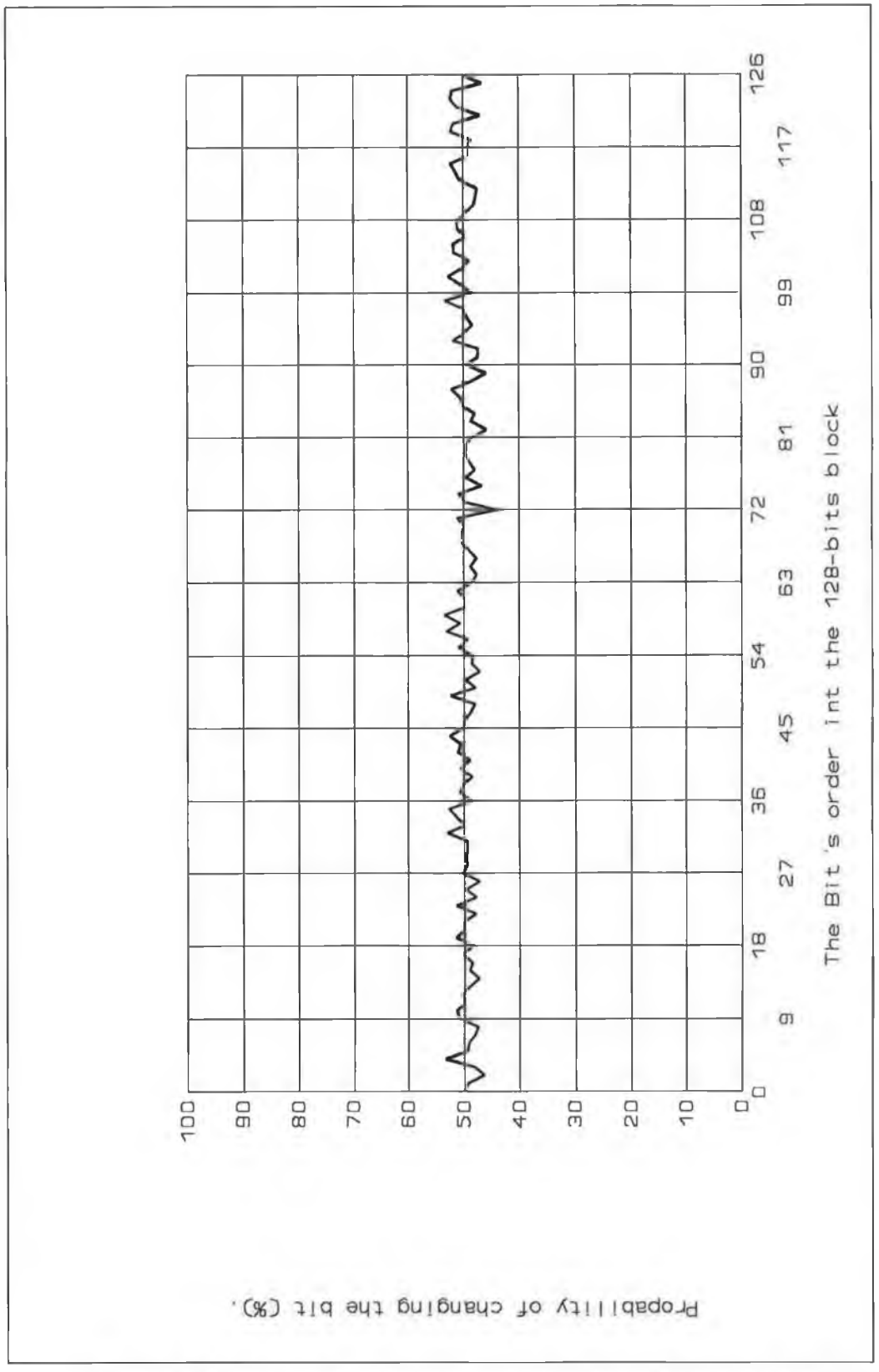



***Figure 5.4.*** *The result of the key-ciphertext Strict Avalanche test.*

## 5.4 Conclusion

Some statistical tests has been implemented on DCU-cipher (in both styles, DCU64 and DCU128). The results of these tests show that DCU-cipher has the property of the avalanche effect within only two rounds. Therefore selecting four-rounds structure appears to be sufficient. The DCU-cipher passed the frequency, serial, runs and the Universal tests, which proves that the DCU-cipher output is a random sequence of bits.

*Chapter 6*

# Concluding Remarks

Conventional cipher systems are the most efficient cryptographic methods for protecting information. The only practical problem in using this type of cipher has been the difficulty of providing a secure way for transferring the secret key from one partner to another, and assuring that the one with whom the secret session is established is the one who it is supposed to be. This problem has been overcome by using the identity-based key exchange protocols by which users in both sides are securely identified. The secret session-key in an identity-based key exchange protocol is based on the two parties' identifications. But the recent discovery of the new type of a chosen-plaintext attack, the *differential cryptanalysis*, which successfully attacked most of the published block cipher algorithms including the *DES*, puts all systems, which are basing their security on such cipher methods, in risk. This highlights the need for a stronger algorithm that stands against the threat of all known-types of attack. In this dissertation, the design and the software implementation of the DCU-cipher algorithm is proposed which appears to be strong against all known attacks including the differential cryptanalysis. The DCU-cipher with 128-bit long cryptographic key is approximately $47 \times 10^{20}$ times stronger than *DES* and about $28 \times 10^{13}$ times stronger than the new *SKIPJACK* cipher algorithm.

Two modes of operation for secure communication and file systems are also suggested here. The threat of known-plaintext differential cryptanalysis on long messages is countered when one of the proposed techniques is used.

We strongly recommend that, when the DCU-cipher algorithm is selected to be used for encrypting long messages, it be implemented in one of the two new proposed modes, either CBC-PX or PCCBC, to avoid any known-plaintext crypt-analytical attack.

# Bibliography

[AMV89] Agnew, G., Mullin, R., and Vanstone, S., "An Interactive Data Exchange Protocol Based on Discrete Exponentiation", *Lecture Notes in Computer Science, Advances in Cryptology- EUROCRYPT'88*, Vol.330,1989, pp.159-166.

[AT90] Adams, C. and Travers, S. " Good S-boxes are easy to find", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'89*, Vol.435,1990, pp.612-615.

[BB92] Boer, den B. and Bosselaers, A., "An Attack on the Last Two Rounds of MD4", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'91*, Vol.476, 1992, pp. 194-203.

[BBD92] Beth, T., Bauspiess, F. and Damm, F., "Workshop on Cryptographic Hash Functions", *E.I.S.S. Report 92/11*, 1992.

[BCS90] Bellare, M., Cowen, L. and Goldwasser, S. "On the Structure of Secret Key Exchange Protocols", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'89*, Vol.435,1990, pp.604-606.

[BD92] Bauspiess, F. and Damm, F., "Requirements for Cryptographic Hash Functions", *E.I.S.S. Report 92/2*, 1992.

[Be89] Beth, T., "Efficient Zero-Knowledge Identification Scheme for Smart Cards", *Lecture Notes in Computer Science, Advances in Cryptology- EUROCRYPT'88*, Vol.330, 1989, pp. 76-84.

[Ber92] Berson, T., "Differential Cryptanalysis Mod $2^{32}$ with Applications to MD5", *Eurocrypt'92 (Extended abstracts pp. 67-76)*, May 24-28, 1992, To appear.

[BFS92] Beth, T., Frisch, M. and Simmons, G (ED), "Public-Key Cryptography: State of Art and Future Directions", *E.I.S.S WorkShop, Oberwolfach, Germany, July 3-6 1991*, Springer-Verlg, 1992.

[BG91] Baritaud, T. and Gilbert, H., "F.F.T. Hashing is not Collision-free", *EUROCRYPT'92, Extended abstracts*, pp.31-40. To appear.

[BK90] Bauspieß,F. and Knobloch, H.J., "How to Keep Authenticity alive in a Computer Network", *Lecture Notes in Computer Science, Advances in*

*Cryptology- EUROCRYPT'89*, Vol.434, 1990, pp. 38-46.

[BMV85] Blake, I.F, Mullin, R.C., and Vanstone, S.A, " Computing Logarithms in GF($2^n$)", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'84*, Vol.196, 1985, pp.73-82.

[Boe89] Boer, B. D., "Cryptanalysis of F.E.A.L.", *Lecture Notes in Computer Science, Advances in Cryptology- EUROCRYPT'88*, Vol.330, 1989, pp.293-300.

[BP82] Beker, H. and Piper, F., *Cipher Sysytem: The protection of Communications*, Northwood Books, 1982.

[BPS91a] Brown, L., Pieprzyk, J. and Seberry, J., "LOKI- A cryptographic Primitive fro Authentication and Secrecy Applications", *Advances in Cryptology - Auscrypt'90*, pp. 229-236, Springer-Verlag, 1991.

[BPS91b] Brown, L., Pieprzyk, J. and Seberry, J., "Improving Resistance to Differential Cryptanalysis & the Redesign of LOKI", *Technical Report CS38/91*, Dept. of Computer Sci., University of South Wales, Australian Defence Force Academey, 1991.

[BS91] Biham, E. and Shamir, A., "Differential Analysis of DES-like Cryptosystems", *Advances in cryptology - Crypto'90*, Springer-Verlag, 1991.

[BS92a] Biham, E. and Shamir, A., "Differential Analysis of FEAL and N-Hash", *Advances in Cryptology - Eurocrypt'91*, pp. 1-16. Springer-Verlag, 1992.

[BS92b] Biham, E. and Shamir, A., "Differential Analysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer"", *Advances in Cryptology - Crypto'91*, pp. 156-171. Springer-Verlag, 1992.

[BS92c] Biham, E. and Shamir, A., "Differential Cryptanalysis of the Full 16-round DES", *CRYPTO'92 ( Extended abstracts, pp.12:1-6)*. To appear.

[CED87] Chaum, D., Evertse, J.H., and Graaf,D., "Demonstrating possession of a discrete logarithm without revealing it", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'86*, Vol.263, 1987, pp.200-212.

[CG92] Corfdir, A. T. and Gilbert, H., " A Known Plaintext Attack of FEAL-4 and FEAL-6", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'91*, Vol.576, 1992, pp.172-181.

[CW91] Cusick, T., and Wood, M. C., " The REDOC-II Cryptosystem",*Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'90*, 1991

[Dam90] Damgard, I. B., "A Design Principle for Hash Functions", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'89*, Vol.435, 1990, pp. 416-427.

[DDQ85] Davio, M., Desmedt, Y. and Quisquater, J.J. "Propagation Characteristics of the DES", *Lecture Notes in Computer Science, Advances in Cryptology-EUROCRYPT'84*, Vol.205, 1985, pp.62-71.

[Den82] Dennings, D. E., *Cryptography and Data Security*, Addision-Wesley, 1982.

[Det85] Davio, M. and et al, "Efficient Hardware and Software Implementations for the DES", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'84*, Vol.196, 1985, pp.144-147.

[DH76] Diffie, D. and Hellman, M., "New Directions in Cryptography", *IEEE Transactions on Information Theory*, Vol. IT-22, Nov. 1976, pp. 644-654.

[DHS85] Davis, J.A., Holdridge, D.B., and Simmons, G.J, "Status Report on Factoring (At the Sandia National Lab.)", *Lecture Notes in Computer Science, Advances in Cryptology-EUROCRYPT'84*, Vol.205, 1985, pp.183-215.

[DK91] Denes, J and Keedwell, A.D., *Latin Squares new Developments in the Theory and Applications*. North-Holland, 1991.

[DP84] Davies, D. W. and Price, W. L., *Security for Computer Networks*, Wiley, 1984.

[DQD85] Desmet,Y., Quisquater, J. J., and Davio, M., "Dependence of Output on Input in DES: Small Avalanche Characteristics", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'84*, Vol.196, 1985, pp.359-376.

[DR90] Devore, J. and Peck, R., *Introductory Statistics*, West Publishing Co.,1992.

[El85] ElGamal, T., "A public key Cryptosystem and Digital Signature scheme Based on Discrete Logarithms", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'84*, Vol.196, 1985, pp.10-18.

[Fah93] Fahn, P., "Answers To Frequently Asked Questions About Today's Cryptography", *RSA Laboratories, a division of RSA Date Security, Inc.,Part #002-903002-200-02f-000,* September 1993.

[Fi90] Fiat, A.,"Batch RSA", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'89,* Vol.435, 1990, pp.175-185.

[FR90] Fereer, J.D, and Rotger, L.H, "Full secure key exchange and authentication with no previously shared secrets", *Lecture Notes in Computer Science, Advances in Cryptology-EUROCRYPT'89*, Vol. 434, 1990, pp. 665-669.

[FS87] Fiat,A. and Shamir, A., " How to Prove Yourself", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'86*, Vol.236, 1987, pp. 189-194.

[GDC92] Gustafson, H., Dawson, E. and Caelli, B., " Comparison of Block Ciphers", *Lecture Notes in Computer Science, Advances in Cryptology-AUSCRYPT'91,* 1993, PP.208-220.

[Gi92] Girault, M., "Self-Certified Public Keys", *Lecture Notes in Computer Science, Advances in Cryptology- EUROCRYPT'91,* 1992, pp. 490-497.

[Gor85] Gordon, J.A., " Strong Primes are Easy to Find", *Lecture Notes in Computer Science, Advances in Cryptology- EUROCRYPT'84,* 1985, Vol.209, pp. 216-223.

[GQ89] Guillou, L.C., and Quisquater, J.J.,"A Practical Zero-Knowledge Protocol Fitted to Security Microprocessors Minimizing Both Transmission and Memory, *Lecture Notes in Computer Science, Advances in Cryptology-EUROCRYPT'88,* Vol. 330, 1989, pp 123-128.

[Gü90] Günther, C.G., "An Identity-Based Key- Exchanges Protocol", *Lecture Notes in Computer Science, Advances in Cryptology- EUROCRYPT'89*, Vol. 434, 1990, pp.29-37.

[Hw91] Hwang, T., "Cryptosystem for Group Oriented Cryptography", *Lecture Notes in Computer Science, Advances in Cryptology- EUROCRYPT'90,* 1991, pp. 352-360.

[Kno89] Knobloch, H.J., " A Smart Card Implementation of the Fiat-Shamir Identification Scheme", *Lecture Notes in Computer Science, Advances in Cryptology- EUROCRYPT'88*, Vol. 330, 1989, pp.87-96.

[Knu92] Knudsen, L.R., "Iterative Characteristics of DES and $s^2$ DES", *CRYPTO'92 (Extended abstracts, pp. 12:6-11)*, August 15-20, 1992, Santra Barbra, CA.

[KO88] Koyama, K. and Ohta, K., "Identity-based key distribution systems", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'87,* Vol.293, 1988, pp.175-184.

[Koh86] Kohnheim, A.G., *"Cryptography: A primer"*, John Wiley & Sons, New York, 1986.

[Koh90] Kohl, J., "The use of Encryption In Kerberos for Network Authentication", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'89,* Vol.435, 1990, pp.35-43.

[Kro86] Kroniakis, E., *Primality and Cryptography*, Wily-Teubner Series in Computer Sci., John-Whily & Sons, 1986.

[LM91] Lai, X. and Massey, J.L., "A Proposal for a new Block Encryption Standard", *Advances in Cryptology - Eurocrypt'90*, pp. 389-404. Springer-Verlag 1991.

[LM92] Lai, X. and Massey, J.M., "Hash Functions Based on Block Ciphers", *Eurocrypt'92, (extended abstracts, pp. 53-66).* To appear.

[LMM92] Lai, X., Massy, J.L. and Murphy, S., "Markov Ciphers and Differential Cryptanalysis", *Advances in Cryptology - Eurocrypt'91*, pp. 17-38. Springer-Verlag, 1992.

[LT85] Leung, A.k. and Tavares, S.E. "Sequence Complexity as a Test for Cryptographic Systems", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPT'84*, Vol. 196, 1985, pp.468-474.

[Mat88] Matsumoto, T., "On the key pre-distribution system: A practical solution to the key distribution problem" *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'87,* Vol.293, Springer-Verlag, 1988, pp.185-193.

[Mau92] Maurer, U. M., " A Universal Statistical Test for Random Bit Generators", *Journal of Cryptography, Vol.5*, 1992, pp.89-105.

[McC88] McCurley, K., "A Key Distribution System Equivalent to Factoring", *Journal of Cryptology*, 1, 1988, pp.95-105.

[McL92] McLaughlin, R., "Yet Another Machine to Break DES", *Cryptologia, Vol. XVI, No.2*, April, 1992.

[Mer90a] Merkle, R., "One Way Hash Functions and DES", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'89,* Vol.435, 1990, pp.428-446.

[Mer90b] Merkle, R., "A Fast Software One-Way Hash Function", *Journal of Cryptology, Vol.3, No. 1*, 1990, pp. 43-85.

[Mer91] Merkle, R.C., " Fast Software Encryption Functions", *Advances in cryptology - Crypto'90*, springer-Verlag, 1991.

[MH78] Merkle, R. and Hellman, M., "Hiding information and signature in trapdoor Knapsacks", *IEEE Trans. Inofrm. Theo. Vol. 24, No. 5*, Sept. 1978, pp. 525-530.

[Miy90] Miyaguchi, S. et al, "Expansion of FEAL Cipher", *NTT Review, Vol. 2, No. 6*, pp. 117 -127, November, 1990.

[MKO90] Miyaguchi, S., Kurihara, S. and Ohta, K.,"Expansion of FEAL Cipher", *NTT Review*, Vol.2, No.6, November, 1990.

[MM82] Meyer, C. H. and Matyas, S. T., *Cryptography: A New dimension in Computer Data Security*, John Wiley & Son, 1982.

[Moo40] Mood,A. M., "The Distribution Theory of Runs", *Ann. Maths. Statist. II.* 1940, pp. 367-392.

[MPW92] Mitchell, C. J., Piper, F. and Wild, P., "Digital Signatures", *Contemporary Cryptology: The Science of Information Integrity*, Simmons, G. J. (Ed.), IEEE Press, 1992, pp.325-278.

[MY91] Maurer, U. and Yacobi, Y., "Non-Interactive Key Cryptography", *Lecture Notes in Computer Science, Advances in Cryptology-EUROCRYPT'91*, Vol. 547,1992, pp. 498-507.

[MY92] Maurer, U. and Yacobi, Y., "A Remarks on a Non-Interactive Key Distribution System", *Lecture Notes in Computer Science, Advances in Cryptology- EUROCRYPT'92*. To appear.

[NBS77] National Bureau of Standards, *Data Encryption Standard*, Federal Information Processing Standards Publication 46, Jan. 1977.

[NEWS1] -Kammer, R. G.,"*The statement of G.Kammer before the subcommittee on the Communications and Finance Committee on Energy and Commerce*, Unpublished manuscript, 29, April, 1993.
-The statement of the secretary of the White House, 16, April, 1993.

[Odl85] Odlyzko, A.M., "Discrete Logarithms in Finite Fields and their Cryptographic Significance", *Lecture Notes in Computer Science, Advances in Cryptology- EUROCRYPT'84,* 1985, Vol.209, pp. 224-316.

[OhO89] Ohta, K., and Okamoto,T. "A Modification of the Fiat-Shamir Scheme", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'88*, Vol. 403, 1989, PP.232-247.

[Ok86] Okamoto, E. "A Proposal for Identity-Based Key Distribution Systems", *El.letters*, Vol. 22, No. 23, 20 Nov. 1986, pp. 1283-4.

[OO91] Okamoto, T. and Ohta, K, "How to utilize the randomness of zero-knowledge proofs", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'90,* 1991, pp. 456-475.

[OT92] Ohtsuka, K. and Taniguchi, T., "A cipherment Algorithm CALC for C Programming Language", *Trans. Inst. Electron. Comm. Eng. Vol. J75 D-I D(1)*, pp.63-66, 1992. (In Japanese).

[Pol78] Pollard, J.,M., "Monte Carlo Methods for Index Computation (mod p).", *Math. Comp. Vol. 32, No.24*, 1978, pp. 1283-1284.

[QD90] Quisquater, J. J. and Delescaille, J. P., "How easy is Collision Search: New results and applications to DES", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'89,* Vol.435, 1990, pp.408-413.

[QG90] Quisquater, J. J. and Girault, M., "2-n Bit Hash-Functions using n-Bit Symmetric Block Cipher Algorithms", *Lecture Notes in Computer Science, Advances in Cryptology-EUROCRYPT'89,* Vol.434, 1990, pp.102-109.

[RD91] Rivest, R. and Dusse, S. "The MD4 Message-Digest Algorithm", *Network Working Group Internet-Draft*, July, 1991.

[RD92] Rivest, R. and Dusse, S. "The MD5 Message-Digest Algorithm", *Network Working Group Internet Draft, RSA Data Security Inc.*, January, 1991.

[RM85] Reeds, J.A., and Manferdelli, J.L., "DES Has No Per Round Linear Factors". *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'84,* Vol.196, 1985, pp.377-392.

[RSA78] Rivest, R.L., Shamir, A., and Adlemen, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Comm. of ACM,* Vol.21, No. 2, Feb. 1978, pp. 120-126.

[Sal90] Salomaa, A., *Public-Key Cryptography*, Springer-Verag, Berlin, 1990.

[SB92] Smid, M.E. and Branstad, D.k., "The Data Encryption Standard Past

and Future", *Contemporary Cryptology The Science of Information Integrity*, Simmons, G.J. (Ed.), IEEE Press, 1992.

[Sch90] Schnorr, C.R., "Effecient Identification and Digital Signature for Smart Cards", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'89,* Vol.435, 1990, pp.239-252.

[Sch94] Schneier, B., *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, 1994.

[Sco92] Scott, M., "File Encryption with no Known Plaintext", *Working Paper: CA-1092*, School of Computer Applications, Dublin City University, 1992.

[SG92] Stubblebine, S.G., and Gligor, V.D., "On Message Integrity in Cryptographic Protocols", *Technical Report No. 2843*, Electrical Eng. Dept. University of Maryland, February, 1992.

[Sha80] Shamir, E., "The Cryptographic Security of Compact Knapsacks", *Proceedings of the Symposium on Privacy and Security*, 1980, pp.95-99

[Sha82] Shamir, E., "A polynomial Time Algorithm for Breaking the Basic Merkle-Hellman Cryptosystem", *Proceedings of the 23rd IEEE Symposioum on Founds. Computer Science,* 1982, pp. 142-152.

[Sha85] Shamir, A., "Identity-Based cryptosystems and signature Schemes", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'84*, Vol.196, 1985, pp.47-53.

[SHS92] ___, "Specifications for a Secure Hash Standard (SHS)", *Federal Information Processing Standards Publication YY, DRAFT*, January, 1992.

[SM88] Shimizu, A., and Miyaguchi, S., "Fast Data Encipherment Algorithm FEAL", *Lecture Notes in Computer Science, Advances in Cryptology-EUROCRYPT'87*, Vol.304, 1988,pp 267-278.

[SP89] Seberry, J. and Pieprzyk, J., *Cryptography: An Introduction to Computer Security*, Prentice Hall, 1989.

[SS92a] Shafa'amry, M. and Scott., M., "On the Identity-Based Key Exchange Protocols", *Working paper:CA-2592*, School of Computer Applications, Dublin City University, 1992.

[SS92b] Scott, M., and Shafa'amry, M., "Implementing an Identity-based Key exchange Algorithm", *Working paper:CA-0992*, School of Computer

Applications, Dublin City University, 1992.

[SS93a] Scott, M. and Shafa'amry, M., "Novel Chaining Methods for Block Ciphers", *Working paper: CA-1993.*, School of Computer Applications, Dublin City University, 1993.

[SS93b] Shafa'amry, M. and Scott, M. "DCU-Cipher : A Secret-Key Block Cipher System", *International Symposium in Computer Science and Applied Mathematics, CSAM'93*, July 1993, To appear.

[SY90] Shimizu, A. and Yamakami, T., "A Fast 32-bit Microprocessor Oriented Data Encipherment Algorithm", *The Transaction of the IEICE, Vol. E 73, No.7*, July 1990.

[TM90] Tatebayashi,M. and Matsuzaki, N., "Key Distribution Protocol for Digital Mobile Communication Systems, *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'89*, Vol.435, 1990, pp.324-332.

[TO90] Tanaka, K. and Okamoto, E., "Key Distribution System using ID-related Information Directory suitable for Mail Systems", *Proc. of SECURICOM'90*, pp.115-122.

[VanT88] Van Tilborg, H. C.A., *An Introduction to Cryptology*, Kluwer Academic Publishers, Boston, 1988.

[Wel88] Welsh, D., *Codes and Cryptography*, Oxford Science of Publication, Clarendon Press- Oxford, 1988.

[WT86] Webster, A.F. and Tavares, E., "On the Design of S-Boxes", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'85*, Vol.218, 1986, pp.523-534.

[YS90] Yacobi, Y and Shmuely, Z. "On Key Distribution Systems", *Lecture Notes in Computer Science, Advances in Cryptology- CRYPTO'89*, Vol.435, 1990, pp.345-355.

[ZTI90] Zheng, Y., Matsumoyo, T. and Imai, H., "On the Construction of Block Ciphers Provably Secure and Not Relying on an Unproved Hypotheses", *Lecture Notes in Computer Science, Advances in Cryptology-CRYPTO'89,* Vol.435, 1990, pp.461-480.

# Appendix - A

The following is the main C-code routines for DCU-Cipher Algorithm.
Note that this code is for the DCU64. The main difference between the DCU64 algorithm's code and the DCU128 is that the input/output sub-blocks are defined as *char* (8-bits each) in the first mode, DCU64, while these sub-blocks are defined as *int* (16-bits each) in the second mode (DCU128).

```
#include <stdio.h>
#include <stdlib.h>
#define     maxim 257          /* maxim = 65537 for DCU128 */
#define     fuyi  256          /* fuyi  = 65536 for DCU128 */
#define     one   255          /* one   = 65535 for DCU128 */
#define     round 4
#define     SIZE  9

/*************************************************************************
Routine Name:     cip64().
Function:         4-rounds DCU-Cipher Encryption Algorithm.
*************************************************************************/
void  cip64(unsigned char IN[SIZE], unsigned char OUT[SIZE],unsigned char
      Z[11][SIZE])
{
   unsigned       i,j,r,e;
   unsigned char  x[SIZE],kk1,kk2,t1,t2,a,C;
   unsigned char  outx[17],temp[17];
   C = '4';                           /* the value that effects the ROR in
                                         the right branch */
   for(i=0;i<SIZE-1;i++)
       x[i] = IN[i] ;
   for ( r = 1; r <= round; r++)      /* No.of rounds. */
    {
                                      /* effecting the input subblocks by the
                                         subkeys*/
      for(i=0;i<2;i++)
         {
          x[i] = mul64(x[i],Z[i+1][r]);
          x[i+6] = mul64(x[i+6],Z[i+7][r]);
          x[i+2]=(x[i+2]+Z[i+3][r]) & one;
          x[i+4]=(x[i+4]+Z[i+5][r]) & one;
       }
       kk1 = mul64(Z[SIZE][r],(x[0]^x[4]));
       t1  = (kk1+(x[1]^x[5]))&one;
       kk2 = mul64(t1,(x[2]^x[6]));
       t2  = (kk2+(x[3]^x[7]))&one;
       t2  = mul64(Z[10][r],RoRn64(t2,C));
       kk2 = (RoRn64(kk2,t2)+t2)&one;
       t1  = mul64(RoRn64(t1,kk2),kk2);
       kk1 = (RoRn64(kk1,t1) +t1)&one;
      x[0] = x[0]^t2; x[1] = x[1]^kk2;
      x[6] = x[6]^t1;x[7] = x[7]^kk1;

       a = x[2]^t1;      x[2] = x[4]^t2;    x[4] = a;
       a = x[3]^kk1;     x[3] = x[5]^kk2;   x[5] = a;
       }
   for(i = 0;i<2;i++)
   {
    OUT[i] = mul64(x[i],Z[i+1][round+1]);
    OUT[i+6] = mul64(x[i+6],Z[i+7][round+1]);
```

```
    OUT[i+2]=(x[i+4]+Z[i+3][round+1]) & one;
    OUT[i+4]=(x[i+2]+Z[i+5][round+1]) & one;
   }
 }

/*************************************************************************
Routine name:     mul64().
Function:   Multipying two chacters mod 257.
*************************************************************************/
unsigned char mul64(unsigned char a,unsigned char b)
{
int               p;
unsigned int      q,d,e;
unsigned char     x,y;

x = a; y = b;
d = (int)x;
e = (int)y;
if(d == 0)  p = maxim-e;
else if(e == 0)   p= maxim-d;
     else
      {
       q = (unsigned int)d*e;
       p = (q & one) - (q>>8);
       if (p <=0) p = p+maxim;
      }
return(char)(p&one);
}

/*************************************************************************
 Routine name:    RoRn64().
 Function   :     Rotates Right  a character X ,by the
                  value of the first 3-bits of n.
*************************************************************************/
unsigned char RoRn64(unsigned char x,unsigned char n)
{
 unsigned char    y,z,w;
 y = z = w = '\0';
 w =n & 7;
 y = x>>w;
 z = x<<(8-w);
 return (unsigned char)( y|z);
 }

/*************************************************************************
Routine name:     nkey()
Function   :      Generating keys for DCU-64, by using addition mode 255
           and variable rotation. (see figure. 4.4)
*************************************************************************/
Void nkeys(unsigned char  userkey[17], unsigned char keys[11][SIZE])
{
unsigned char     A[17];
int               i,j,k,c,start;
c = 0; start = 0;
for(i = 0;i<10;i++)
      for(j =0;j<5;j++)
            keys[i][j] =0;
for(i=0; i<16;i++)
      A[i] = userkey[i];
for(j = 0;j<3;j++)
      {
      for(i=0;i<8;i++)
            A[i] = A[i]+A[i+8] &one;
      for(i =0;i<7;i++)
```

```
            A[i] = RoRn64(A[i],A[i+1]);
      A[7] = RoRn64(A[7],A[0]);
      for(i = 8;i<16;i++)
            A[i] = A[i]+A[i-8]&one;
      for(i = 8;i<15;i++)
            A[i] = RoRn64(A[i],A[i+1]);
      A[15] = RoRn64(A[15],A[8]);
      /* storing the 16-characters of the A[] in the keys[][]array */
      k = 16;
      if(start>0&& k>=10)
        {
            for(i = start;i<10;i++)
                keys[i][c] = A[i-start];
            k = k-(10-start);
            c++;
            start = 0;
            if(k>=10)
            {
                for(i = start;i<10;i++)
                    keys[i][c] = A[16-k+i];
                k = k-10;
                c++;
                for(i = 0;i<k;i++)
                keys[i][c] = A[16-k+i];
                start = k;
            }
          else
            {
                for(i = 0;i<k;i++)
                    keys[i][c] = A[16-k+i];
                start = k;
            }
      }
       else  if(start == 0&&k>=10)
            {
             for(i = start;i<10;i++)
                keys[i][c] = A[i];
             k = k-10;
             c++;
             for(i = 0;i<k;i++)
                keys[i][c] = A[i+10];
             start = k;
             }
          else
            {
             for(i = 0;i<k;i++)
                keys[i][c] = A[i];
             start = k;
             }
    }
}

/*************************************************************************
Routine name:     de-key64();
Function:   Compute the decryption key blocks DK[i][r] from the
            encrypiton key blocks Z[i][r]
*************************************************************************/
void  de_key64(unsigned char z[10][SIZE],unsigned char  DK[11][SIZE])
{
  int             j,i,d,e;
  for(i=1;i<3;i++)
      {
      DK[i][1] = inv64(z[i][5]);
      DK[i+6][1] = inv64(z[i+6][5]);
```

```
      DK[i+2][1] = (fuyi -z[i+2][5]) & one;
      DK[i+4][1] = (fuyi -z[i+4][5]) & one;
      DK[i][5] = inv64(z[i][1]);
      DK[i+6][5] = inv64(z[i+6][1]);
      DK[i+2][5] = (fuyi -z[i+2][1]) & one;
      DK[i+4][5] = (fuyi -z[i+4][1]) & one;
      }
  for( j = 2; j <= round; j++)
      {
        for(i =1;i<3;i++)
        {
            DK[i][round-j+2] = inv64(z[i][j]);
            if(DK[i][round-j+2]<0)
                  DK[i][round-j+2] =DK[i][round-j+2]+fuyi;
            DK[i+6][round-j+2] = inv64(z[i+6][j]);
            if(DK[i+6][round-j+2]<0)
                  DK[i+6][round-j+2]=DK[i+6][round-j+2]+fuyi;
            DK[i+2][round-j+2] = (fuyi -z[i+4][j]) & one;
            DK[i+4][round-j+2] = (fuyi -z[i+2][j]) & one;
        }
         }
  for (j = 1; j<round+1; j++)
        {DK[9][round+1-j] = z[9][j]; DK[10][round+1-j] = z[10][j];}
}
```

# Appendix - B

This appendix contains the C-code for the statistical tests which have been implemented on the DCU-Cipher algorithm.

```
/*************************************************************************
 Programme name:        Utest.c
 Function         :     Applying the Universal test on DCU-Cipher.
*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "mdcu.h"

/* Maurer's Universal test for Random bits */

#define Q 10000          /* >3000  */
#define K 1000000L       /* >100*Q */
#define MEAN 7.1836656
#define DEVIATION  1.5*sqrt(3.238/(double)K)
void    cip128(unsigned    IN[SIZE],unsigned    OUT[SIZE],    unsigned
      Z[11][SIZE]);
unsigned int generator(long V);
/*************************************************************************
Routine name:            maurer().
Function    :            Universal test for randomness.
*************************************************************************/
int maurer()
{
  double           sum,ftu;
  int              i,j;
  long             n;
  static  long     tab[256];

  for(i=0;i<256;i++) tab[i] = (-1); printf("\n table initialized.");
  for(n = 0;n<Q;n++)
      {
      j =generator(n);
      tab[j]=n;
       }
 /*printf("\n program run for  %d values=Q INIT.\n",n);*/

  /*check each byte occurred at least once */
  for(i = 0;i<256;i++) if (tab[i]<0)
  return 0;
  sum = 0.0;
  for(n=Q;n<Q+K;n++)
  {/* scan byte sequence */
    i =generator(n);
    sum +=log((double)n-tab[i]);
    tab[i] = n;
    /*printf("*");*/
   }
  ftu = ((sum/(double)K)/log(2.0));
  printf("\n ftu = %lf DEV = %lf\n",ftu-MEAN, DEVIATION);
```

```
  if(ftu>(MEAN+DEVIATION)||ftu<(MEAN-DEVIATION))return 0;
  return 1;
}

/*********************************************************************
Routine name: generator ()
Function: Using DCU-Cipher algorithm to generate a random byte.
*********************************************************************/
unsigned int generator(long V)

{/*random bit generator/ Pack bits into byte */
int               i,j,x;
unsigned  int     x_in[SIZE], OUT[SIZE], Key[11][SIZE], ;

4=4;
 for(i=0;i<SIZE;i++)
{ x_in[i] = 0;OUT[i]=0;}
                                    /* Generate cipher input blocks */
if(V<65535)
      x_in[1] = (int)V;
else x_in[2] = (int)V;
  for(i =0;i<11;i++)
      for(j=0;j<=SIZE;j++)
            Key[i][j] =1;           /* get a fixed sub-keys. All of
                                    them have the value=1    */
 icip128(x_in,OUT,Key);
                                    /*Collecting the fist bit of each
                                    sub-block to form a random byte*/
for(x=0,i=0;i<8;i++)
      x|=((OUT[i]&1)<<i);
 return (x);
}

/*********************************************************************
 The Main Universal test program
*********************************************************************/
main()
{/* test bit generator for randomness */
  if(maurer())
     printf("This seems to be a GOOD random bit generator \n");
  else printf("This is a BAD random bit generator\n");
}

/*************************************************************
Routine Name: cip128();
Function:   DCU128 Encryption Algorithm
*************************************************************/
void cip128(unsigned IN[SIZE],unsigned OUT[SIZE], unsigned Z[11][SIZE])
 {
   unsigned i,j,r,x[SIZE],kk1,kk2,t1,t2,a,C;
   unsigned  char outx[17], temp[17];

   C = 4;  /* the value that effect the ROR in the right branch */
   for(i=0;i<SIZE-1;i++)
       x[i] = IN[i] ;

for ( r = 1; r <= 4; r++)  /* No.of rounds  */
   {
                                    /* effecting the input sub-blocks
                                    by the sub-keys*/
      for(i=0;i<2;i++)
         {
            x[i] = mul(x[i],Z[i+1][r]);
```

```
            x[i+6] = mul(x[i+6],Z[i+7][r]);
            x[i+2]=(x[i+2]+Z[i+3][r]) & one;
            x[i+4]=(x[i+4]+Z[i+5][r]) & one;
          }
        kk1 = mul( Z[9][r],(x[0]^x[4]));
        t1  = (kk1+(x[1]^x[5]))&one;
        kk2 = mul(t1,(x[2]^x[6]));
        t2  = (kk2+(x[3]^x[7]))&one;
        t2  = mul(Z[10][r],RoRn(t2,C));
        kk2 = (RoRn(kk2,t2)+t2)&one;
        t1  = mul(RoRn(t1,kk2),kk2);
        kk1 = (RoRn(kk1,t1) +t1)&one;
       x[0] = x[0]^t2; x[1] = x[1]^kk2;
       x[6] = x[6]^t1;x[7] = x[7]^kk1;
        a = x[2]^t1;      x[2] = x[4]^t2;
       x[4] = a;
        a = x[3]^kk1;     x[3] = x[5]^kk2;  x[5] = a;
        }
    for(i = 0;i<2;i++)
    {
     OUT[i] = mul(x[i],Z[i+1][round+1]);
     OUT[i+6] = mul(x[i+6],Z[i+7][round+1]);
     OUT[i+2]=(x[i+4]+Z[i+3][round+1]) & one;
     OUT[i+4]=(x[i+2]+Z[i+5][round+1]) & one;
    }
}
```

```
/*********************************************************************
Program Name        : Strict_pc()

Function            : Plaintext-ciphertext Strict Avalanche test effect
                      on DCU64, using K random Plaintext vectors.
*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "dcu64.h"
#define      N    65          /* Block size +1  */
#define      n    (N-1)/8           /* Number of sub-blocks */
#define      K    100         /* Number of the tests */
/*#define  R_no   4*/

void  cip(unsigned char IN[SIZE], unsigned char OUT[SIZE],unsigned char
      Z[11][SIZE], unsigned int R_no);
void  Getblock(unsigned char X[n+1]);
void  Gblock (unsigned char Bk1[n+1], unsigned char Bk2[n+1], unsigned
      int B_no);

main()

{
unsigned int      k,i,j,l, R_no;
unsigned char     Binary[N];
unsigned char     Bk1[n+1],Bk2[n+1],OUT1[n+1],OUT2[n+1],    AVL[n+1],
Key[11][n+1];
unsigned double   Blck_AVAL[9],T_AVAL[9];
unsigned int      Depend[N][N];
R_no = 4;
for(i = 0;i<N; i++)
      for(j=0;j<N;j++) Depend[i][j] = 0;        /*    Init.    The
                                                Dependency array */
for (i =0;i<=n;i++){
Bk1[i] = 0;Bk2[i] =0; OUT1[i] = 0;
OUT2[i] =0; AVL[i]=0; Blck_AVAL[i] =0.0; T_AVAL[i]=0.0;}
for(i =0;i<11;i++)
      for(j=0;j<=n;j++)
            Key[i][j] =1;                  /* get a fixed subkeys all
                                           of them have the value=1*/

for (k=0;k<K;k++)
  {
  Getblock(Bk1);                           /* Generate a random block
                                           BK1   */
  cip(Bk1,OUT1,Key,R_no);                  /*Encrypt Bk1      */
  for(i = 0; i < N-1;i++)
      {
      Gblock(Bk1,Bk2,i);                   /* Generate BK2 that differs
                                           in 1-bit from BK1*/
      cip(Bk2,OUT2,Key,R_no);              /*Encrypt BK2      */
      for(j=0;j<8;j++)
            AVL[j] = OUT1[j]^ OUT2[j];
      Bnry_rep64(AVL,Binary);
      for(j=0;j<N-1;j++)
            {
            Depend[i][j] =Depend[i][j] +(int)(Binary[j]-48);
            }
      }
      /* printf("Test No. %d\n",k);*/
  }
  for (i=0;i<N;i++)
      {for(j=0;j<N;j++)
            printf("%d  ",Depend[i][j]);
```

```
        printf("\n");
        }
}

/*******************************************************
Routine Name:      Getblock()
Function:        Generate a random block of the length n
*******************************************************/
void  Getblock(unsigned char X[n+1])
{
unsigned int       i;
for (i = 0;i<=n;i++)
      X[i]='\0';
randomize();
for(i = 0;i<n;i++)
X[i] = random(255);
X[n] = '\0';
}

/*******************************************************
Routine Name:      Gblock()
Function:    Generate a block BK2 that differs in 1-bit
             from BK1, the bit B_no.
*******************************************************/
void   Gblock(unsigned char Bk1[n+1], unsigned char Bk2[n+1], unsigned
      int B_no)
{
unsigned      int i,j, z;

for(i =0;i<=n;i++)
      Bk2[i] = Bk1[i];
i = B_no/8;                    /* to specify in which byte the bit
                               which will be complemented*/
j = B_no%8;                    /* J is the number of bit in the byte
                               (sub-block)*/
if(j==0) j=8;                  /* this means the last bit in the
                               sub-block */
z = 1<<(j-1);
if(Bk1[i]&z ==z) Bk2[i]=Bk1[i]-z;
else  Bk2[i] = Bk1[i]+z;
}
```

```
/*****************************************************************
Program Name        : Aval_pc()
Function            : Avalanche test effect on DCU64, using K random
                      Plaintext vectors.
Output              : An array T_AVAL [i] of 8 values. Each of them
                      represents the result of the avalanche effect test
                      on DCU64 with number of rounds=i, where i = 1..8.
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "dcu64.h"
#define       N     65          /* Block size +1  */
#define       n     (N-1)/8     /* Number of sub-blocks */
#define       K     1000        /* Number of the tests */

void  cip(unsigned char IN[SIZE], unsigned char OUT[SIZE],unsigned
      char Z[11][SIZE], unsigned int R_no);
void  Getblock(unsigned char X[n+1]);
void  Gblock (unsigned char Bk1[n+1], unsigned char Bk2[n+1],
      unsigned int B_no);
unsigned int      Binary_rep64(unsigned char IN[n+1]);
main()
{
unsigned int            k,i,j,l,R_no,B;
unsigned char           Binary[N];
unsigned char           Bk1[n+1],Bk2[n+1],OUT1[n+1],OUT2[n+1],
                        AVL[n+1], Key[11][n+1];
unsigned double         Blck_AVAL[9],T_AVAL[9];

B = 0;
for (i =0;i<=n;i++){
Bk1[i] = 0;Bk2[i] =0; OUT1[i] = 0;
OUT2[i] =0; AVL[i]=0; Blck_AVAL[i] =0.0; T_AVAL[i]=0.0;}

for(i =0;i<11;i++)
      for(j=0;j<=n;j++)
            Key[i][j] =1;           /* get a fixed sub-keys all of
                                       them have the value=1*/
for (k=0;k<K;k++)
  {
  Getblock(Bk1);                    /* Generate a random block BK1*/
  for(i = 0; i < N-1;i++)
      {
      Gblock(Bk1,Bk2,i);            /* Generate BK2 that differs in
                                       1-bit from BK1*/
      for(R_no =1;R_no<=8;R_no++)
            {
            cip(Bk1,OUT1,Key,R_no); /*Encrypt Bk1     */
            cip(Bk2,OUT2,Key,R_no); /*Encrypt BK2     */
            for(j=0;j<8;j++)
                  AVL[j] = OUT1[j]^ OUT2[j];
            B = Binary_rep64(AVL);
            Blck_AVAL[R_no] = Blck_AVAL[R_no] +B;
            B=0;
            }
      }
  for(l =0;l<=n;l++)
      {
      Blck_AVAL[l] = Blck_AVAL[l]/(N-1);
      T_AVAL[l] = Blck_AVAL[l]+T_AVAL[l];
      Blck_AVAL[l] = 0.0;
      }
  printf("TEST-No: %d\n", k);
```

```
  }
for(l = 0;l<=n;l++)

      {
      T_AVAL[l] = T_AVAL[l]/K;
      printf("%d = %5.3f\n",l,T_AVAL[l]/64);
      }
      printf("\n");
}

unsigned int Binary_rep64( unsigned char   x[SIZE])

{
  unsigned int                  i,j,k;
  unsigned char                 temp;

   k=0;
   for (i = 0;i<n;i++)
   {
   temp = x[i];
   for(j = 0;j<n;j++)
       {
      k=k+((temp>>j)&1);
      }
    }
    return (k);
  }
```

```
/*********************************************************************
Program Name:     r&ser64.c
Function:    serial and run tests DCU64 by using chi-sqare test.It
             calls the file INPUT64.DAT which contains all the
             plaintext which has 64, 63& 62 ones/and zeros.
             (using the routine ser_tst() for calculating chi-seq)
             See Cipher Systems  book by H. Beker & F. Piper .
*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "dcu64.h"
#define  BLOCKSIZE      64
#define      L 24
#define      M 40
#define  ZL -1.96
#define      ZM  1.96
#define  chi_ref  5.99  /*for  2 degree of freedom and a=0.05 */

main()
{
unsigned char     x_in[SIZE], x_out[SIZE],keys[11][SIZE];
unsigned short    keyblock[SIZE];
unsigned char     block1[65];
unsigned  int     i,j,k,ll,kk, rr,PTRN[5],n0,n1,l;
float             chi_sq,Z;
FILE              *fp;

i =0; ;j = 0;ll = 0;kk = 0;l =0;rr =0;
chi_sq = 0.00;Z = 0.0;
for(i =0;i<4;i++)
      {
      PTRN[i] = 0;
      keyblock[i] = keyblock[i+4] = 0;
      }
for(i = 0;i<SIZE;i++)
      {
       keyblock[i] = i+1;
       x_in[i] =0;
       x_out[i] = 0;
       }
for(i =0;i<10;i++)
      for(j = 0;j<SIZE;j++)
            keys[i][j] =0;
key64(keyblock,keys);
if((fp =fopen("input64.dat","rb"))==NULL)
      {
      printf("\n The data file INPUT64.DAT is not found\n");
      exit(0);
      }
i =1;
i =fread(x_in,SIZE-1,1,fp);
while(i ==1)
      {
      icip64(x_in,x_out,keys);
      chi_sq =ser_tst64(x_out);
      Z =rtst64(x_out);
      if(chi_sq>chi_ref)
            ++kk;
      Bnry_rep64(x_out,block1);
      n1 = no_of_changes64(block1);
      n0 = BLOCKSIZE-n1;
      if(n1<L||n1>M)
            ++l;
```

```
        if(Z<ZL||Z>ZM)
              {++rr;
              printf("+");}

         ++ll;
        i =fread(x_in,SIZE-1,1,fp);
        }/*End of while*/
printf("KK = %u\nLL = %u\nthe AVRG ofX^2 = %f\n", kk, ll,
((float)(kk)/(float)(ll)));

printf("\n freq-test result:\n R = %d \t R/N= %f\n",l,
(float)l/(float)ll);

printf("\n the run test result:\n R = %d \t R/N = %f\n",
rr,(float)rr/(float)ll);
} /* end of the main */
```

# APPENDIX - C

The results of the Avalanche effect test on DCU-Cipher Algorithm for 1000 random plaintexts of the size 128-bits :

| Changing the input bit bit-NO. | Number of the bits in the DCU-Cipher output that are changed after: 1-round | 2-rounds | 3-rounds | 4-rounds | 5-rounds | 6-rounds | 7-rounds | 8-rounds |
|---|---|---|---|---|---|---|---|---|
| 0 | 38.28 | 51.56 | 51.56 | 48.44 | 49.22 | 51.56 | 56.25 | 53.12 |
| 1 | 35.94 | 55.47 | 50.78 | 49.22 | 57.81 | 45.31 | 55.47 | 44.53 |
| 2 | 31.25 | 49.22 | 53.12 | 51.56 | 50 | 48.44 | 50 | 38.28 |
| 3 | 32.81 | 49.22 | 50 | 49.22 | 42.97 | 53.91 | 57.81 | 50 |
| 4 | 46.88 | 49.22 | 53.12 | 54.69 | 50 | 52.34 | 59.38 | 61.72 |
| 5 | 44.53 | 50 | 49.22 | 51.56 | 41.41 | 56.25 | 53.12 | 49.22 |
| 6 | 41.41 | 52.34 | 60.16 | 54.69 | 45.31 | 49.22 | 42.19 | 48.44 |
| 7 | 40.62 | 50 | 42.19 | 46.09 | 53.12 | 52.34 | 41.41 | 59.38 |
| 8 | 37.5 | 44.53 | 50.78 | 50.78 | 47.66 | 52.34 | 47.66 | 49.22 |
| 9 | 45.31 | 53.91 | 53.12 | 50 | 42.19 | 62.5 | 50.78 | 49.22 |
| 10 | 39.06 | 52.34 | 49.22 | 57.03 | 48.44 | 50 | 55.47 | 59.38 |
| 11 | 37.5 | 56.25 | 46.88 | 50.78 | 51.56 | 50 | 50.78 | 54.69 |
| 12 | 42.97 | 46.09 | 49.22 | 50.78 | 56.25 | 50.78 | 55.47 | 50.78 |
| 13 | 39.84 | 48.44 | 50 | 50.78 | 48.44 | 50 | 52.34 | 52.34 |
| 14 | 35.16 | 53.12 | 53.91 | 49.22 | 44.53 | 43.75 | 48.44 | 50.78 |
| 15 | 36.72 | 50.78 | 49.22 | 46.88 | 54.69 | 56.25 | 48.44 | 49.22 |

| bit-NO. | 1-round | 2-rounds | 3-rounds | 4-rounds |
|---|---|---|---|---|
| 16 | 31.25 | 46.88 | 51.56 | 50 |
| 17 | 46.88 | 49.22 | 52.34 | 62.5 |
| 18 | 32.81 | 53.91 | 42.97 | 53.12 |
| 19 | 42.97 | 59.38 | 59.38 | 53.12 |
| 20 | 39.84 | 51.56 | 55.47 | 50 |
| 21 | 39.84 | 52.34 | 64.06 | 46.88 |
| 22 | 46.88 | 46.88 | 50.78 | 46.88 |
| 23 | 45.31 | 53.12 | 52.34 | 48.44 |
| 24 | 43.75 | 46.88 | 52.34 | 42.97 |
| 25 | 38.28 | 57.03 | 43.75 | 49.22 |
| 26 | 38.28 | 53.91 | 51.56 | 45.31 |
| 27 | 44.53 | 53.12 | 56.25 | 50.78 |
| 28 | 37.5 | 53.91 | 53.12 | 49.22 |
| 29 | 50 | 46.88 | 42.97 | 54.69 |
| 30 | 50.78 | 47.66 | 46.09 | 53.91 |
| 31 | 39.84 | 46.88 | 50 | 54.69 |
| 32 | 53.91 | 55.47 | 62.5 | 53.12 |
| 33 | 45.31 | 53.12 | 54.69 | 55.47 |
| 34 | 53.91 | 60.94 | 50 | 46.09 |
| 35 | 54.69 | 51.56 | 53.91 | 64.06 |
| 36 | 46.09 | 48.44 | 50 | 42.19 |
| 37 | 53.12 | 53.12 | 53.91 | 53.91 |

| 5-rounds | 6-rounds | 7-rounds | 8-rounds |
|---|---|---|---|
| 53.12 | 47.66 | 55.47 | 53.12 |
| 47.66 | 42.19 | 52.34 | 49.22 |
| 52.34 | 42.19 | 56.25 | 53.12 |
| 50 | 47.66 | 46.88 | 52.34 |
| 56.25 | 43.75 | 47.66 | 55.47 |
| 45.31 | 51.56 | 50 | 50 |
| 44.53 | 51.56 | 53.12 | 42.97 |
| 47.66 | 56.25 | 57.03 | 50.78 |
| 46.88 | 49.22 | 51.56 | 49.22 |
| 54.69 | 53.91 | 53.12 | 47.66 |
| 50.78 | 50 | 38.28 | 50 |
| 42.19 | 46.09 | 56.25 | 52.34 |
| 55.47 | 47.66 | 47.66 | 50.78 |
| 45.31 | 53.12 | 53.91 | 50 |
| 46.88 | 51.56 | 46.09 | 51.56 |
| 50.78 | 46.09 | 46.09 | 60.94 |
| 41.41 | 54.69 | 54.69 | 48.44 |
| 50.78 | 43.75 | 53.91 | 42.19 |
| 57.81 | 48.44 | 51.56 | 42.97 |
| 49.22 | 57.03 | 42.97 | 48.44 |
| 53.91 | 48.44 | 50 | 50 |
| 46.09 | 46.88 | 46.88 | 51.56 |

| bit-NO. | 1-round | 2-rounds | 3-rounds | 4-rounds |
|---|---|---|---|---|
| 38 | 58.59 | 46.09 | 45.31 | 44.53 |
| 39 | 46.88 | 50.78 | 49.22 | 44.53 |
| 40 | 53.91 | 38.28 | 50.78 | 59.38 |
| 41 | 48.44 | 54.69 | 46.09 | 43.75 |
| 42 | 49.22 | 53.12 | 54.69 | 50.78 |
| 43 | 43.75 | 42.19 | 52.34 | 42.97 |
| 44 | 47.66 | 56.25 | 56.25 | 48.44 |
| 45 | 50.78 | 44.53 | 46.88 | 56.25 |
| 46 | 37.5 | 56.25 | 49.22 | 52.34 |
| 47 | 42.19 | 56.25 | 45.31 | 47.66 |
| 48 | 39.84 | 47.66 | 38.28 | 50.78 |
| 49 | 52.34 | 54.69 | 48.44 | 48.44 |
| 50 | 56.25 | 46.09 | 44.53 | 39.06 |
| 51 | 55.47 | 47.66 | 58.59 | 46.88 |
| 52 | 43.75 | 49.22 | 40.62 | 53.91 |
| 53 | 35.94 | 45.31 | 56.25 | 53.12 |
| 54 | 42.19 | 42.19 | 50 | 52.34 |
| 55 | 53.12 | 53.91 | 46.09 | 50.78 |
| 56 | 53.91 | 50 | 39.06 | 48.44 |
| 57 | 45.31 | 49.22 | 55.47 | 55.47 |
| 58 | 53.91 | 51.56 | 48.44 | 50 |
| 59 | 63.28 | 50.78 | 50 | 51.56 |

| 5-rounds | 6-rounds | 7-rounds | 8-rounds |
|---|---|---|---|
| 49.22 | 51.56 | 56.25 | 50.78 |
| 46.88 | 46.88 | 53.91 | 49.22 |
| 53.12 | 53.91 | 46.88 | 49.22 |
| 53.12 | 52.34 | 43.75 | 49.22 |
| 48.44 | 57.03 | 45.31 | 50.78 |
| 54.69 | 61.72 | 53.12 | 53.91 |
| 46.88 | 53.91 | 55.47 | 55.47 |
| 45.31 | 58.59 | 44.53 | 54.69 |
| 45.31 | 55.47 | 60.16 | 42.19 |
| 45.31 | 45.31 | 53.91 | 54.69 |
| 55.47 | 52.34 | 53.91 | 46.09 |
| 51.56 | 51.56 | 57.81 | 53.91 |
| 47.66 | 47.66 | 48.44 | 53.91 |
| 40.62 | 51.56 | 51.56 | 45.31 |
| 48.44 | 53.12 | 44.53 | 43.75 |
| 54.69 | 47.66 | 43.75 | 52.34 |
| 53.12 | 43.75 | 53.91 | 50 |
| 50 | 50 | 56.25 | 46.88 |
| 50.78 | 54.69 | 46.88 | 50.78 |
| 51.56 | 57.81 | 58.59 | 51.56 |
| 53.91 | 54.69 | 47.66 | 45.31 |
| 53.91 | 55.47 | 48.44 | 54.69 |

| bit-NO. | 1-round | 2-rounds | 3-rounds | 4-rounds |
|---|---|---|---|---|
| 60 | 38.28 | 56.25 | 48.44 | 46.09 |
| 61 | 15.62 | 53.91 | 52.34 | 64.06 |
| 62 | 46.88 | 49.22 | 53.12 | 55.47 |
| 63 | 20.31 | 40.62 | 58.59 | 53.12 |
| 64 | 38.28 | 50.78 | 48.44 | 52.34 |
| 65 | 35.94 | 54.69 | 44.53 | 49.22 |
| 66 | 31.25 | 54.69 | 43.75 | 53.91 |
| 67 | 32.81 | 50.78 | 56.25 | 53.91 |
| 68 | 46.88 | 50 | 50 | 46.88 |
| 69 | 44.53 | 56.25 | 55.47 | 51.56 |
| 70 | 41.41 | 54.69 | 50 | 49.22 |
| 71 | 40.62 | 59.38 | 44.53 | 48.44 |
| 72 | 37.5 | 41.41 | 48.44 | 43.75 |
| 73 | 46.09 | 53.12 | 46.88 | 58.59 |
| 74 | 39.84 | 51.56 | 59.38 | 48.44 |
| 75 | 36.72 | 46.88 | 46.88 | 42.97 |
| 76 | 41.41 | 46.09 | 53.12 | 53.12 |
| 77 | 39.84 | 46.88 | 45.31 | 53.12 |
| 78 | 35.94 | 60.94 | 53.91 | 46.88 |
| 79 | 38.28 | 56.25 | 51.56 | 48.44 |
| 80 | 31.25 | 48.44 | 49.22 | 54.69 |
| 81 | 46.88 | 46.09 | 52.34 | 45.31 |

| 5-rounds | 6-rounds | 7-rounds | 8-rounds |
|---|---|---|---|
| 53.12 | 54.69 | 53.91 | 47.66 |
| 52.34 | 53.91 | 56.25 | 50 |
| 57.81 | 47.66 | 50 | 49.22 |
| 60.16 | 47.66 | 58.59 | 50 |
| 46.88 | 57.03 | 45.31 | 42.97 |
| 54.69 | 53.91 | 46.09 | 51.56 |
| 47.66 | 50.78 | 47.66 | 52.34 |
| 50.78 | 53.12 | 45.31 | 49.22 |
| 48.44 | 55.47 | 56.25 | 50.78 |
| 45.31 | 53.91 | 46.88 | 50 |
| 51.56 | 53.12 | 46.09 | 53.12 |
| 54.69 | 58.59 | 52.34 | 51.56 |
| 49.22 | 46.09 | 45.31 | 40.62 |
| 49.22 | 49.22 | 50 | 46.09 |
| 50.78 | 47.66 | 57.03 | 52.34 |
| 54.69 | 47.66 | 50.78 | 47.66 |
| 46.09 | 51.56 | 53.91 | 51.56 |
| 48.44 | 50.78 | 50 | 50 |
| 48.44 | 48.44 | 46.09 | 51.56 |
| 54.69 | 48.44 | 54.69 | 45.31 |
| 51.56 | 53.91 | 55.47 | 52.34 |
| 41.41 | 57.81 | 46.09 | 52.34 |

| bit-NO. | 1-round | 2-rounds | 3-rounds | 4-rounds |
|---|---|---|---|---|
| 82 | 32.81 | 42.19 | 45.31 | 46.88 |
| 83 | 42.97 | 56.25 | 54.69 | 46.88 |
| 84 | 39.84 | 48.44 | 52.34 | 52.34 |
| 85 | 39.84 | 52.34 | 59.38 | 50 |
| 86 | 46.88 | 54.69 | 50.78 | 52.34 |
| 87 | 45.31 | 52.34 | 51.56 | 58.59 |
| 88 | 43.75 | 53.91 | 43.75 | 51.56 |
| 89 | 38.28 | 43.75 | 49.22 | 46.09 |
| 90 | 37.5 | 53.91 | 50.78 | 52.34 |
| 91 | 44.53 | 47.66 | 46.09 | 55.47 |
| 92 | 38.28 | 44.53 | 54.69 | 49.22 |
| 93 | 50 | 51.56 | 51.56 | 57.81 |
| 94 | 51.56 | 53.91 | 49.22 | 43.75 |
| 95 | 41.41 | 46.09 | 52.34 | 57.03 |
| 96 | 56.25 | 46.09 | 56.25 | 49.22 |
| 97 | 46.88 | 46.88 | 55.47 | 46.09 |
| 98 | 53.12 | 53.12 | 55.47 | 44.53 |
| 99 | 53.12 | 46.88 | 44.53 | 37.5 |
| 100 | 45.31 | 47.66 | 51.56 | 45.31 |
| 101 | 53.12 | 53.91 | 50.78 | 48.44 |
| 102 | 56.25 | 44.53 | 52.34 | 51.56 |
| 103 | 49.22 | 53.91 | 35.94 | 46.88 |

| 5-rounds | 6-rounds | 7-rounds | 8-rounds |
|---|---|---|---|
| 49.22 | 46.09 | 53.12 | 51.56 |
| 48.44 | 49.22 | 44.53 | 46.09 |
| 47.66 | 44.53 | 53.91 | 45.31 |
| 45.31 | 53.12 | 45.31 | 56.25 |
| 51.56 | 46.88 | 43.75 | 57.81 |
| 57.03 | 52.34 | 53.12 | 46.88 |
| 50.78 | 46.88 | 46.09 | 52.34 |
| 53.12 | 39.84 | 44.53 | 53.12 |
| 44.53 | 49.22 | 53.12 | 55.47 |
| 42.19 | 37.5 | 53.91 | 51.56 |
| 37.5 | 54.69 | 49.22 | 50.78 |
| 52.34 | 55.47 | 45.31 | 50.78 |
| 44.53 | 49.22 | 51.56 | 54.69 |
| 43.75 | 45.31 | 50 | 51.56 |
| 50.78 | 49.22 | 41.41 | 46.88 |
| 47.66 | 50 | 56.25 | 50.78 |
| 53.91 | 56.25 | 51.56 | 58.59 |
| 53.12 | 50.78 | 46.88 | 54.69 |
| 50 | 54.69 | 53.12 | 56.25 |
| 53.91 | 60.94 | 45.31 | 55.47 |
| 46.09 | 46.09 | 52.34 | 53.91 |
| 55.47 | 50 | 52.34 | 47.66 |

| bit-NO. | 1-round | 2-rounds | 3-rounds | 4-rounds |
|---|---|---|---|---|
| 104 | 57.81 | 47.66 | 48.44 | 54.69 |
| 105 | 48.44 | 50 | 60.16 | 52.34 |
| 106 | 46.88 | 52.34 | 51.56 | 53.12 |
| 107 | 45.31 | 57.03 | 54.69 | 48.44 |
| 108 | 46.88 | 47.66 | 57.03 | 53.91 |
| 109 | 49.22 | 46.09 | 47.66 | 51.56 |
| 110 | 36.72 | 43.75 | 47.66 | 56.25 |
| 111 | 39.06 | 41.41 | 50.78 | 50.78 |
| 112 | 40.62 | 46.09 | 47.66 | 51.56 |
| 113 | 52.34 | 47.66 | 59.38 | 50.78 |
| 114 | 56.25 | 61.72 | 44.53 | 52.34 |
| 115 | 55.47 | 55.47 | 46.09 | 49.22 |
| 116 | 43.75 | 54.69 | 46.09 | 53.12 |
| 117 | 35.94 | 48.44 | 50.78 | 49.22 |
| 118 | 42.19 | 46.88 | 46.09 | 51.56 |
| 119 | 53.12 | 50 | 50.78 | 50.78 |
| 120 | 53.91 | 50 | 57.03 | 43.75 |
| 121 | 46.09 | 44.53 | 41.41 | 44.53 |
| 122 | 53.12 | 57.81 | 53.12 | 47.66 |
| 123 | 64.84 | 64.06 | 48.44 | 47.66 |
| 124 | 50 | 49.22 | 54.69 | 57.81 |
| 125 | 16.41 | 51.56 | 50.78 | 53.91 |

| 5-rounds | 6-rounds | 7-rounds | 8-rounds |
|---|---|---|---|
| 44.53 | 57.81 | 57.03 | 46.09 |
| 52.34 | 46.09 | 56.25 | 49.22 |
| 46.09 | 42.19 | 57.03 | 47.66 |
| 53.91 | 48.44 | 49.22 | 51.56 |
| 47.66 | 46.09 | 50.78 | 59.38 |
| 57.03 | 50.78 | 52.34 | 42.19 |
| 53.91 | 49.22 | 49.22 | 47.66 |
| 49.22 | 53.12 | 52.34 | 45.31 |
| 46.09 | 43.75 | 50.78 | 53.91 |
| 48.44 | 49.22 | 49.22 | 48.44 |
| 53.91 | 42.97 | 54.69 | 45.31 |
| 53.91 | 49.22 | 48.44 | 60.94 |
| 55.47 | 50.78 | 43.75 | 46.09 |
| 53.12 | 53.12 | 52.34 | 50.78 |
| 53.91 | 47.66 | 51.56 | 49.22 |
| 54.69 | 59.38 | 48.44 | 50.78 |
| 51.56 | 55.47 | 46.09 | 56.25 |
| 46.09 | 43.75 | 55.47 | 53.91 |
| 50.78 | 51.56 | 46.09 | 48.44 |
| 46.09 | 52.34 | 50 | 45.31 |
| 58.59 | 50 | 50.78 | 56.25 |
| 53.91 | 53.91 | 45.31 | 47.66 |

| bit-NO. | 1-round | 2-rounds | 3-rounds | 4-rounds |
|---|---|---|---|---|
| 126 | 46.09 | 49.22 | 53.12 | 42.97 |
| 127 | 44.06 | 50.02 | 52.32 | 50.40 |
| **AVRG:** | **44.01** | **50.62** | **50.62** | **50.45** |

| 5-rounds | 6-rounds | 7-rounds | 8-rounds |
|---|---|---|---|
| 57.81 | 48.44 | 50.78 | 53.12 |
| 59.21 | 49.94 | 55.08 | 52.42 |
| ------ | ------ | ------ | ------ |
| **50.07** | **50.68** | **50.63** | **50.55** |

# Appendix - D

The results of the Strict Avalanche test on DCU-Cipher Algorithm for 1000 random plaintexts of the size 64-bits (the dependency matrix).

| Input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 0 | 487 | 516 | 538 | 547 | 480 | 509 | 494 | 528 | 450 | 528 | 528 | 532 | 472 | 502 | 492 | 471 | 507 |
| 1 | 442 | 513 | 479 | 552 | 464 | 464 | 480 | 453 | 480 | 453 | 568 | 510 | 435 | 491 | 462 | 493 | 496 |
| 2 | 503 | 445 | 554 | 490 | 507 | 420 | 451 | 517 | 512 | 537 | 509 | 572 | 432 | 488 | 480 | 517 | 485 |
| 3 | 487 | 542 | 486 | 552 | 459 | 536 | 558 | 483 | 517 | 541 | 504 | 527 | 509 | 532 | 490 | 523 | 526 |
| 4 | 504 | 502 | 535 | 491 | 563 | 484 | 503 | 476 | 479 | 495 | 527 | 533 | 477 | 557 | 469 | 569 | 499 |
| 5 | 525 | 552 | 507 | 518 | 434 | 481 | 491 | 493 | 538 | 559 | 529 | 472 | 582 | 446 | 519 | 552 | 540 |
| 6 | 515 | 520 | 503 | 507 | 501 | 427 | 476 | 558 | 476 | 505 | 511 | 512 | 494 | 501 | 419 | 469 | 493 |
| 7 | 515 | 525 | 503 | 540 | 533 | 492 | 562 | 531 | 459 | 456 | 493 | 552 | 543 | 442 | 427 | 491 | 502 |
| 8 | 491 | 491 | 455 | 510 | 486 | 515 | 476 | 457 | 490 | 464 | 507 | 527 | 482 | 513 | 494 | 531 | 485 |
| 9 | 522 | 492 | 517 | 467 | 512 | 492 | 544 | 507 | 502 | 531 | 555 | 531 | 501 | 567 | 493 | 508 | 467 |
| 10 | 466 | 504 | 513 | 464 | 516 | 575 | 464 | 567 | 465 | 519 | 506 | 461 | 462 | 460 | 490 | 527 | 479 |

| Input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 11 | 465 | 496 | 485 | 484 | 483 | 488 | 519 | 457 | 493 | 513 | 575 | 532 | 452 | 475 | 487 | 524 | 487 |
| 12 | 501 | 507 | 478 | 458 | 507 | 387 | 543 | 491 | 511 | 544 | 483 | 553 | 497 | 493 | 516 | 488 | 458 |
| 13 | 511 | 516 | 545 | 462 | 535 | 486 | 516 | 513 | 478 | 521 | 511 | 500 | 493 | 491 | 537 | 502 | 557 |
| 14 | 458 | 520 | 544 | 456 | 431 | 559 | 504 | 478 | 497 | 534 | 481 | 513 | 496 | 484 | 495 | 519 | 540 |
| 15 | 517 | 415 | 463 | 522 | 493 | 523 | 551 | 485 | 484 | 516 | 463 | 542 | 537 | 515 | 488 | 491 | 490 |
| 16 | 493 | 518 | 519 | 538 | 483 | 482 | 549 | 456 | 548 | 520 | 519 | 464 | 491 | 538 | 512 | 523 | 553 |
| 17 | 513 | 536 | 495 | 521 | 504 | 514 | 567 | 506 | 556 | 511 | 492 | 537 | 484 | 505 | 463 | 487 | 526 |
| 18 | 437 | 507 | 560 | 500 | 502 | 531 | 501 | 531 | 534 | 520 | 519 | 529 | 498 | 503 | 557 | 547 | 490 |
| 19 | 448 | 421 | 539 | 486 | 498 | 514 | 495 | 492 | 471 | 505 | 531 | 485 | 444 | 519 | 508 | 526 | 504 |
| 20 | 539 | 535 | 468 | 428 | 548 | 481 | 575 | 534 | 465 | 450 | 536 | 453 | 445 | 494 | 514 | 474 | 497 |
| 21 | 472 | 480 | 505 | 480 | 473 | 492 | 536 | 510 | 563 | 459 | 458 | 531 | 518 | 518 | 527 | 535 | 535 |
| 22 | 507 | 550 | 503 | 514 | 460 | 482 | 530 | 514 | 445 | 511 | 569 | 486 | 508 | 529 | 544 | 539 | 457 |
| 23 | 493 | 483 | 537 | 433 | 452 | 478 | 486 | 457 | 560 | 477 | 454 | 510 | 483 | 490 | 537 | 470 | 502 |
| 24 | 514 | 465 | 497 | 457 | 512 | 539 | 499 | 459 | 480 | 520 | 441 | 523 | 434 | 514 | 539 | 543 | 501 |
| 25 | 502 | 541 | 501 | 456 | 507 | 471 | 507 | 509 | 532 | 508 | 533 | 433 | 480 | 554 | 513 | 511 | 426 |
| 26 | 527 | 506 | 499 | 528 | 502 | 519 | 466 | 506 | 532 | 497 | 457 | 448 | 418 | 435 | 472 | 517 | 486 |

| Input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 27 | 441 | 491 | 538 | 547 | 547 | 531 | 528 | 533 | 480 | 489 | 485 | 493 | 516 | 576 | 554 | 479 | 490 |
| 28 | 483 | 542 | 511 | 587 | 555 | 453 | 507 | 483 | 519 | 554 | 468 | 526 | 503 | 469 | 491 | 465 | 482 |
| 29 | 511 | 484 | 552 | 472 | 518 | 491 | 527 | 509 | 559 | 500 | 492 | 498 | 465 | 547 | 521 | 486 | 481 |
| 30 | 479 | 480 | 479 | 498 | 470 | 414 | 551 | 533 | 514 | 508 | 523 | 472 | 506 | 527 | 496 | 513 | 488 |
| 31 | 558 | 535 | 529 | 508 | 582 | 491 | 518 | 527 | 489 | 511 | 430 | 509 | 518 | 481 | 578 | 407 | 480 |
| 32 | 545 | 581 | 449 | 482 | 594 | 516 | 558 | 444 | 554 | 517 | 426 | 487 | 499 | 517 | 496 | 496 | 480 |
| 33 | 499 | 519 | 439 | 534 | 485 | 537 | 446 | 539 | 489 | 417 | 542 | 504 | 484 | 483 | 558 | 463 | 496 |
| 34 | 564 | 467 | 466 | 479 | 477 | 498 | 533 | 532 | 456 | 518 | 464 | 589 | 537 | 520 | 532 | 505 | 558 |
| 35 | 508 | 498 | 539 | 454 | 485 | 514 | 512 | 465 | 446 | 478 | 516 | 435 | 496 | 481 | 534 | 523 | 530 |
| 36 | 521 | 530 | 507 | 458 | 479 | 434 | 532 | 522 | 436 | 511 | 560 | 526 | 470 | 493 | 514 | 475 | 507 |
| 37 | 527 | 512 | 467 | 480 | 557 | 521 | 570 | 500 | 432 | 441 | 530 | 559 | 506 | 485 | 529 | 504 | 494 |
| 38 | 482 | 502 | 504 | 469 | 504 | 505 | 520 | 567 | 462 | 494 | 484 | 514 | 457 | 494 | 456 | 500 | 479 |
| 39 | 510 | 524 | 528 | 524 | 474 | 537 | 451 | 478 | 515 | 474 | 524 | 482 | 544 | 458 | 498 | 438 | 537 |
| 40 | 437 | 523 | 495 | 511 | 474 | 527 | 517 | 557 | 489 | 416 | 495 | 499 | 527 | 558 | 543 | 556 | 458 |
| 41 | 473 | 458 | 467 | 474 | 512 | 522 | 518 | 491 | 467 | 478 | 446 | 524 | 512 | 479 | 528 | 518 | 496 |
| 42 | 477 | 501 | 526 | 462 | 550 | 530 | 449 | 499 | 502 | 486 | 502 | 472 | 549 | 483 | 427 | 523 | 516 |

| Input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 43 | 536 | 569 | 480 | 469 | 520 | 474 | 526 | 546 | 427 | 538 | 502 | 494 | 461 | 497 | 457 | 474 | 489 |
| 44 | 547 | 497 | 495 | 480 | 469 | 466 | 478 | 465 | 495 | 448 | 460 | 518 | 459 | 516 | 486 | 466 | 512 |
| 45 | 512 | 481 | 522 | 465 | 531 | 450 | 552 | 494 | 489 | 514 | 504 | 521 | 556 | 519 | 549 | 490 | 551 |
| 46 | 445 | 537 | 483 | 509 | 520 | 443 | 544 | 497 | 521 | 468 | 478 | 431 | 414 | 543 | 570 | 532 | 479 |
| 47 | 509 | 531 | 498 | 477 | 542 | 531 | 478 | 497 | 496 | 550 | 415 | 521 | 506 | 512 | 393 | 422 | 498 |
| 48 | 546 | 578 | 508 | 499 | 458 | 512 | 500 | 527 | 470 | 544 | 513 | 515 | 507 | 547 | 586 | 588 | 524 |
| 49 | 462 | 496 | 542 | 518 | 590 | 474 | 485 | 495 | 484 | 511 | 533 | 522 | 456 | 501 | 555 | 506 | 441 |
| 50 | 392 | 464 | 568 | 466 | 538 | 530 | 552 | 536 | 476 | 472 | 564 | 499 | 527 | 554 | 506 | 448 | 513 |
| 51 | 535 | 503 | 515 | 502 | 487 | 485 | 584 | 538 | 527 | 461 | 568 | 545 | 579 | 502 | 510 | 495 | 541 |
| 52 | 511 | 504 | 524 | 540 | 469 | 512 | 470 | 468 | 537 | 529 | 432 | 502 | 505 | 514 | 517 | 440 | 474 |
| 53 | 480 | 464 | 490 | 448 | 484 | 486 | 527 | 450 | 520 | 511 | 525 | 483 | 570 | 505 | 490 | 550 | 476 |
| 54 | 530 | 500 | 497 | 541 | 529 | 466 | 489 | 519 | 474 | 570 | 524 | 501 | 470 | 474 | 507 | 468 | 524 |
| 55 | 475 | 513 | 557 | 480 | 484 | 593 | 455 | 512 | 523 | 475 | 488 | 499 | 480 | 511 | 543 | 508 | 444 |
| 56 | 508 | 477 | 537 | 422 | 503 | 525 | 468 | 449 | 505 | 438 | 559 | 471 | 534 | 501 | 523 | 547 | 451 |
| 57 | 503 | 514 | 525 | 471 | 508 | 486 | 552 | 514 | 498 | 470 | 493 | 531 | 407 | 562 | 512 | 529 | 518 |
| 58 | 478 | 529 | 506 | 526 | 510 | 476 | 474 | 475 | 504 | 521 | 500 | 486 | 499 | 470 | 532 | 493 | 456 |

| Input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 59 | 479 | 528 | 488 | 542 | 528 | 540 | 504 | 524 | 505 | 469 | 487 | 514 | 449 | 517 | 525 | 444 | 495 |
| 60 | 511 | 556 | 499 | 558 | 523 | 492 | 486 | 544 | 500 | 524 | 494 | 539 | 458 | 506 | 520 | 455 | 487 |
| 61 | 453 | 466 | 530 | 434 | 498 | 518 | 533 | 511 | 552 | 513 | 547 | 436 | 440 | 511 | 456 | 499 | 531 |
| 62 | 527 | 493 | 515 | 493 | 523 | 440 | 466 | 489 | 517 | 491 | 527 | 510 | 552 | 555 | 502 | 464 | 481 |
| 63 | 536 | 467 | 446 | 440 | 468 | 532 | 562 | 470 | 540 | 515 | 572 | 568 | 520 | 493 | 521 | 456 | 512 |

| Input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| 0 | 481 | 459 | 516 | 489 | 515 | 535 | 502 | 472 | 451 | 459 | 463 | 493 | 481 | 457 | 493 | 538 |
| 1 | 538 | 520 | 471 | 495 | 544 | 587 | 496 | 492 | 520 | 502 | 479 | 458 | 498 | 517 | 478 | 539 |
| 2 | 489 | 469 | 472 | 515 | 542 | 477 | 493 | 509 | 529 | 524 | 448 | 460 | 520 | 520 | 499 | 460 |
| 3 | 533 | 398 | 467 | 538 | 505 | 516 | 553 | 443 | 575 | 412 | 516 | 507 | 477 | 536 | 507 | 549 |
| 4 | 514 | 523 | 470 | 481 | 505 | 482 | 484 | 451 | 519 | 532 | 502 | 483 | 538 | 477 | 554 | 508 |
| 5 | 493 | 478 | 532 | 478 | 514 | 505 | 533 | 534 | 508 | 498 | 538 | 525 | 481 | 581 | 442 | 435 |
| 6 | 489 | 527 | 516 | 483 | 521 | 523 | 523 | 569 | 499 | 465 | 538 | 569 | 508 | 487 | 523 | 495 |
| 7 | 569 | 554 | 513 | 501 | 481 | 484 | 529 | 457 | 492 | 505 | 416 | 456 | 502 | 572 | 532 | 482 |
| 8 | 528 | 494 | 542 | 507 | 529 | 537 | 511 | 447 | 575 | 491 | 460 | 563 | 477 | 476 | 498 | 478 |
| 9 | 512 | 562 | 488 | 471 | 519 | 540 | 468 | 495 | 457 | 421 | 558 | 440 | 538 | 571 | 442 | 475 |
| 10 | 463 | 510 | 507 | 538 | 535 | 547 | 485 | 481 | 558 | 475 | 569 | 498 | 541 | 478 | 478 | 540 |
| 11 | 474 | 473 | 520 | 551 | 508 | 456 | 470 | 465 | 558 | 527 | 480 | 470 | 493 | 461 | 533 | 538 |
| 12 | 473 | 538 | 492 | 500 | 527 | 558 | 461 | 517 | 545 | 456 | 605 | 493 | 519 | 527 | 523 | 477 |
| 13 | 561 | 410 | 472 | 476 | 499 | 552 | 569 | 472 | 458 | 508 | 473 | 563 | 484 | 504 | 452 | 476 |
| 14 | 451 | 428 | 439 | 452 | 499 | 516 | 472 | 485 | 472 | 533 | 569 | 448 | 503 | 488 | 523 | 499 |
| 15 | 502 | 507 | 484 | 496 | 506 | 509 | 468 | 500 | 468 | 450 | 535 | 556 | 495 | 493 | 555 | 438 |

| Input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| 16 | 480 | 503 | 506 | 570 | 510 | 521 | 485 | 463 | 460 | 539 | 459 | 520 | 532 | 542 | 468 | 532 |
| 17 | 533 | 571 | 494 | 506 | 554 | 479 | 468 | 497 | 574 | 497 | 463 | 590 | 500 | 542 | 554 | 504 |
| 18 | 503 | 507 | 492 | 500 | 484 | 499 | 453 | 462 | 483 | 464 | 498 | 439 | 452 | 533 | 557 | 562 |
| 19 | 499 | 579 | 525 | 521 | 518 | 447 | 519 | 513 | 491 | 519 | 468 | 524 | 519 | 600 | 495 | 449 |
| 20 | 530 | 482 | 475 | 517 | 525 | 495 | 461 | 512 | 485 | 495 | 498 | 464 | 553 | 452 | 462 | 496 |
| 21 | 392 | 512 | 467 | 465 | 515 | 483 | 480 | 485 | 486 | 491 | 438 | 535 | 521 | 530 | 438 | 530 |
| 22 | 505 | 510 | 533 | 561 | 522 | 502 | 500 | 580 | 456 | 522 | 528 | 482 | 402 | 447 | 567 | 554 |
| 23 | 529 | 499 | 542 | 494 | 504 | 520 | 465 | 463 | 505 | 497 | 497 | 467 | 474 | 510 | 494 | 473 |
| 24 | 517 | 510 | 542 | 493 | 476 | 563 | 575 | 492 | 526 | 508 | 479 | 523 | 560 | 477 | 523 | 477 |
| 25 | 543 | 461 | 448 | 532 | 478 | 552 | 509 | 504 | 510 | 598 | 454 | 464 | 542 | 517 | 515 | 478 |
| 26 | 576 | 516 | 518 | 485 | 496 | 449 | 494 | 593 | 539 | 498 | 498 | 517 | 520 | 515 | 531 | 537 |
| 27 | 538 | 463 | 486 | 506 | 489 | 485 | 474 | 495 | 499 | 477 | 492 | 493 | 467 | 551 | 442 | 509 |
| 28 | 543 | 465 | 428 | 475 | 426 | 508 | 513 | 512 | 494 | 484 | 458 | 502 | 468 | 496 | 522 | 476 |
| 29 | 479 | 470 | 422 | 583 | 488 | 498 | 485 | 474 | 500 | 489 | 506 | 584 | 449 | 530 | 464 | 450 |
| 30 | 539 | 559 | 467 | 458 | 534 | 516 | 533 | 515 | 582 | 561 | 417 | 490 | 471 | 511 | 510 | 521 |
| 31 | 551 | 454 | 548 | 474 | 424 | 566 | 510 | 551 | 408 | 503 | 409 | 488 | 477 | 472 | 414 | 375 |

| Input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| 32 | 477 | 537 | 434 | 571 | 480 | 491 | 506 | 493 | 492 | 459 | 499 | 509 | 452 | 484 | 562 | 469 |
| 33 | 530 | 525 | 500 | 518 | 506 | 471 | 494 | 543 | 461 | 481 | 491 | 529 | 465 | 456 | 507 | 553 |
| 34 | 497 | 467 | 487 | 477 | 476 | 538 | 518 | 450 | 541 | 522 | 543 | 496 | 537 | 481 | 484 | 459 |
| 35 | 480 | 537 | 541 | 494 | 493 | 473 | 421 | 543 | 516 | 533 | 463 | 480 | 444 | 464 | 493 | 505 |
| 36 | 484 | 515 | 523 | 511 | 478 | 451 | 534 | 517 | 473 | 507 | 536 | 515 | 556 | 443 | 474 | 534 |
| 37 | 509 | 476 | 501 | 508 | 487 | 486 | 535 | 567 | 520 | 552 | 463 | 508 | 557 | 549 | 461 | 528 |
| 38 | 550 | 499 | 504 | 424 | 440 | 477 | 478 | 447 | 499 | 510 | 456 | 509 | 474 | 528 | 525 | 537 |
| 39 | 491 | 513 | 467 | 492 | 507 | 532 | 520 | 474 | 520 | 514 | 465 | 527 | 467 | 517 | 455 | 497 |
| 40 | 493 | 492 | 393 | 543 | 542 | 535 | 458 | 562 | 551 | 490 | 471 | 539 | 479 | 496 | 495 | 509 |
| 41 | 490 | 446 | 481 | 551 | 509 | 522 | 442 | 503 | 535 | 463 | 492 | 504 | 541 | 493 | 504 | 504 |
| 42 | 627 | 511 | 511 | 465 | 460 | 463 | 478 | 526 | 495 | 472 | 541 | 456 | 511 | 473 | 457 | 502 |
| 43 | 515 | 534 | 421 | 513 | 512 | 480 | 486 | 464 | 495 | 523 | 471 | 548 | 529 | 494 | 527 | 469 |
| 44 | 510 | 485 | 501 | 493 | 448 | 504 | 488 | 527 | 599 | 520 | 479 | 480 | 492 | 543 | 531 | 440 |
| 45 | 513 | 527 | 446 | 478 | 475 | 492 | 510 | 485 | 453 | 560 | 473 | 432 | 522 | 500 | 522 | 483 |
| 46 | 521 | 535 | 551 | 524 | 493 | 449 | 451 | 492 | 460 | 496 | 536 | 462 | 532 | 496 | 572 | 559 |
| 47 | 541 | 545 | 498 | 538 | 572 | 485 | 456 | 458 | 497 | 498 | 517 | 468 | 542 | 480 | 567 | 475 |

| Input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| 48 | 525 | 470 | 580 | 523 | 487 | 468 | 573 | 561 | 490 | 540 | 445 | 488 | 446 | 533 | 500 | 478 |
| 49 | 508 | 484 | 471 | 527 | 562 | 490 | 498 | 516 | 471 | 479 | 513 | 457 | 537 | 586 | 545 | 511 |
| 50 | 512 | 500 | 499 | 454 | 469 | 532 | 519 | 534 | 520 | 499 | 445 | 541 | 447 | 490 | 493 | 523 |
| 51 | 507 | 557 | 541 | 499 | 473 | 529 | 476 | 519 | 507 | 436 | 502 | 504 | 512 | 457 | 459 | 536 |
| 52 | 510 | 440 | 522 | 473 | 523 | 471 | 478 | 472 | 484 | 477 | 514 | 515 | 538 | 523 | 492 | 510 |
| 53 | 390 | 503 | 470 | 509 | 525 | 518 | 526 | 458 | 470 | 556 | 483 | 504 | 486 | 543 | 476 | 532 |
| 54 | 471 | 471 | 449 | 539 | 498 | 501 | 480 | 526 | 563 | 506 | 471 | 472 | 544 | 488 | 445 | 514 |
| 55 | 469 | 510 | 536 | 451 | 575 | 484 | 583 | 561 | 471 | 493 | 524 | 516 | 496 | 566 | 514 | 442 |
| 56 | 457 | 452 | 484 | 566 | 476 | 517 | 531 | 520 | 453 | 506 | 479 | 543 | 508 | 513 | 424 | 459 |
| 57 | 456 | 518 | 490 | 476 | 528 | 584 | 463 | 554 | 494 | 476 | 494 | 523 | 534 | 521 | 540 | 549 |
| 58 | 487 | 553 | 487 | 560 | 573 | 522 | 525 | 451 | 540 | 494 | 541 | 476 | 541 | 495 | 481 | 427 |
| 59 | 542 | 515 | 491 | 477 | 477 | 541 | 460 | 536 | 532 | 484 | 430 | 507 | 537 | 511 | 504 | 509 |
| 60 | 500 | 442 | 444 | 489 | 488 | 492 | 582 | 549 | 446 | 552 | 543 | 522 | 483 | 596 | 488 | 471 |
| 61 | 481 | 493 | 504 | 528 | 488 | 441 | 552 | 475 | 474 | 483 | 465 | 552 | 534 | 531 | 497 | 508 |
| 62 | 507 | 536 | 507 | 512 | 497 | 461 | 472 | 450 | 468 | 520 | 531 | 553 | 508 | 523 | 495 | 464 |
| 63 | 500 | 479 | 492 | 471 | 422 | 470 | 506 | 521 | 557 | 485 | 521 | 524 | 468 | 462 | 452 | 476 |

| input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| 0 | 493 | 538 | 543 | 527 | 523 | 483 | 524 | 454 | 475 | 523 | 504 | 509 | 528 | 576 | 521 | 496 | 528 | 481 |
| 1 | 478 | 539 | 475 | 488 | 506 | 481 | 505 | 508 | 476 | 468 | 540 | 447 | 542 | 494 | 519 | 457 | 482 | 508 |
| 2 | 499 | 460 | 510 | 503 | 552 | 519 | 551 | 476 | 507 | 473 | 450 | 414 | 507 | 503 | 473 | 481 | 475 | 516 |
| 3 | 507 | 549 | 521 | 509 | 499 | 466 | 532 | 422 | 492 | 517 | 530 | 471 | 529 | 567 | 512 | 565 | 531 | 450 |
| 4 | 554 | 508 | 465 | 541 | 471 | 500 | 539 | 516 | 527 | 444 | 470 | 547 | 528 | 479 | 463 | 463 | 463 | 505 |
| 5 | 442 | 435 | 459 | 446 | 460 | 434 | 467 | 447 | 450 | 489 | 513 | 493 | 574 | 469 | 483 | 472 | 507 | 563 |
| 6 | 523 | 495 | 541 | 484 | 477 | 511 | 460 | 538 | 557 | 528 | 489 | 497 | 456 | 505 | 513 | 514 | 525 | 475 |
| 7 | 532 | 482 | 521 | 464 | 523 | 478 | 522 | 456 | 534 | 496 | 569 | 495 | 459 | 511 | 508 | 545 | 459 | 502 |
| 8 | 498 | 478 | 440 | 520 | 530 | 461 | 467 | 478 | 537 | 514 | 506 | 483 | 480 | 541 | 424 | 510 | 479 | 506 |
| 9 | 442 | 475 | 476 | 457 | 448 | 517 | 502 | 484 | 532 | 493 | 466 | 515 | 550 | 476 | 479 | 497 | 504 | 485 |
| 10 | 478 | 540 | 449 | 516 | 576 | 454 | 485 | 517 | 500 | 461 | 545 | 506 | 543 | 499 | 521 | 549 | 512 | 584 |
| 11 | 533 | 538 | 420 | 382 | 518 | 449 | 494 | 531 | 457 | 487 | 540 | 531 | 527 | 475 | 546 | 553 | 533 | 413 |
| 12 | 523 | 477 | 414 | 462 | 566 | 510 | 443 | 502 | 525 | 516 | 470 | 469 | 497 | 476 | 435 | 513 | 420 | 484 |
| 13 | 452 | 476 | 495 | 458 | 537 | 535 | 549 | 416 | 531 | 504 | 508 | 522 | 537 | 493 | 506 | 520 | 466 | 501 |
| 14 | 523 | 499 | 499 | 489 | 508 | 530 | 451 | 465 | 439 | 535 | 481 | 549 | 449 | 545 | 485 | 522 | 524 | 528 |
| 15 | 555 | 438 | 470 | 486 | 461 | 480 | 504 | 425 | 472 | 550 | 533 | 431 | 471 | 544 | 544 | 543 | 508 | 503 |

| input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| 16 | 468 | 532 | 498 | 429 | 578 | 529 | 414 | 517 | 517 | 478 | 507 | 461 | 503 | 481 | 463 | 520 | 528 | 538 |
| 17 | 554 | 504 | 510 | 543 | 411 | 519 | 464 | 407 | 449 | 574 | 525 | 538 | 485 | 520 | 472 | 465 | 502 | 498 |
| 18 | 557 | 562 | 563 | 563 | 473 | 501 | 558 | 465 | 492 | 596 | 553 | 524 | 480 | 507 | 476 | 494 | 516 | 532 |
| 19 | 495 | 449 | 565 | 487 | 508 | 523 | 458 | 506 | 489 | 477 | 471 | 460 | 524 | 553 | 476 | 541 | 536 | 539 |
| 20 | 462 | 496 | 473 | 474 | 527 | 509 | 522 | 538 | 464 | 524 | 484 | 472 | 591 | 423 | 527 | 536 | 527 | 517 |
| 21 | 438 | 530 | 489 | 446 | 536 | 453 | 506 | 536 | 478 | 525 | 507 | 481 | 563 | 467 | 439 | 494 | 520 | 469 |
| 22 | 567 | 554 | 508 | 534 | 528 | 509 | 532 | 510 | 483 | 519 | 570 | 475 | 456 | 416 | 462 | 523 | 464 | 481 |
| 23 | 494 | 473 | 501 | 540 | 479 | 483 | 551 | 554 | 480 | 545 | 495 | 530 | 472 | 468 | 454 | 549 | 479 | 414 |
| 24 | 523 | 477 | 465 | 465 | 472 | 480 | 519 | 503 | 568 | 489 | 512 | 520 | 519 | 437 | 474 | 509 | 520 | 464 |
| 25 | 515 | 478 | 518 | 502 | 438 | 560 | 495 | 492 | 539 | 463 | 439 | 585 | 564 | 462 | 477 | 489 | 471 | 489 |
| 26 | 531 | 537 | 491 | 499 | 546 | 581 | 476 | 536 | 524 | 511 | 509 | 496 | 482 | 498 | 475 | 462 | 488 | 504 |
| 27 | 442 | 509 | 508 | 495 | 517 | 470 | 485 | 549 | 453 | 517 | 473 | 460 | 434 | 498 | 511 | 497 | 517 | 545 |
| 28 | 522 | 476 | 455 | 433 | 500 | 436 | 500 | 488 | 472 | 494 | 490 | 526 | 533 | 446 | 491 | 452 | 460 | 545 |
| 29 | 464 | 450 | 512 | 488 | 514 | 532 | 517 | 600 | 500 | 524 | 528 | 521 | 516 | 447 | 495 | 423 | 485 | 469 |
| 30 | 510 | 521 | 537 | 431 | 523 | 490 | 550 | 467 | 508 | 548 | 459 | 465 | 485 | 550 | 502 | 508 | 536 | 480 |
| 31 | 414 | 375 | 504 | 486 | 485 | 465 | 468 | 527 | 531 | 546 | 532 | 422 | 466 | 490 | 470 | 557 | 511 | 518 |

| input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| 32 | 562 | 469 | 494 | 502 | 518 | 517 | 574 | 402 | 483 | 559 | 473 | 579 | 503 | 485 | 501 | 497 | 578 | 496 |
| 33 | 507 | 553 | 509 | 455 | 491 | 486 | 471 | 490 | 514 | 543 | 491 | 493 | 503 | 559 | 538 | 535 | 472 | 559 |
| 34 | 484 | 459 | 422 | 436 | 534 | 506 | 488 | 488 | 539 | 501 | 482 | 485 | 495 | 443 | 470 | 461 | 508 | 493 |
| 35 | 493 | 505 | 430 | 432 | 474 | 482 | 456 | 451 | 490 | 498 | 433 | 459 | 476 | 471 | 495 | 502 | 517 | 477 |
| 36 | 474 | 534 | 478 | 481 | 543 | 458 | 496 | 549 | 510 | 500 | 544 | 517 | 513 | 501 | 545 | 467 | 490 | 499 |
| 37 | 461 | 528 | 467 | 453 | 503 | 512 | 504 | 501 | 578 | 506 | 478 | 523 | 550 | 538 | 473 | 452 | 474 | 474 |
| 38 | 525 | 537 | 522 | 478 | 492 | 486 | 432 | 456 | 521 | 457 | 485 | 527 | 508 | 468 | 457 | 392 | 484 | 531 |
| 39 | 455 | 497 | 522 | 482 | 525 | 466 | 534 | 487 | 530 | 546 | 511 | 530 | 493 | 467 | 520 | 411 | 496 | 591 |
| 40 | 495 | 509 | 549 | 458 | 540 | 520 | 524 | 534 | 541 | 486 | 511 | 465 | 509 | 434 | 538 | 479 | 484 | 467 |
| 41 | 504 | 504 | 497 | 558 | 513 | 489 | 472 | 449 | 493 | 506 | 465 | 531 | 478 | 465 | 475 | 447 | 535 | 495 |
| 42 | 457 | 502 | 442 | 514 | 520 | 566 | 473 | 561 | 490 | 596 | 459 | 449 | 465 | 460 | 472 | 513 | 494 | 503 |
| 43 | 527 | 469 | 414 | 549 | 509 | 513 | 505 | 516 | 547 | 511 | 550 | 485 | 545 | 474 | 508 | 508 | 570 | 485 |
| 44 | 531 | 440 | 503 | 445 | 491 | 480 | 504 | 559 | 518 | 519 | 567 | 504 | 552 | 560 | 461 | 551 | 513 | 517 |
| 45 | 522 | 483 | 434 | 479 | 518 | 488 | 523 | 537 | 545 | 462 | 531 | 538 | 577 | 481 | 485 | 507 | 492 | 486 |
| 46 | 572 | 559 | 503 | 561 | 421 | 462 | 506 | 496 | 508 | 526 | 473 | 534 | 386 | 608 | 522 | 465 | 450 | 533 |
| 47 | 567 | 475 | 543 | 480 | 534 | 527 | 523 | 525 | 485 | 519 | 457 | 535 | 489 | 423 | 455 | 529 | 565 | 476 |

| input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| 48 | 500 | 478 | 517 | 556 | 425 | 555 | 488 | 469 | 549 | 521 | 463 | 484 | 513 | 490 | 522 | 444 | 443 | 524 |
| 49 | 545 | 511 | 570 | 514 | 465 | 430 | 481 | 489 | 522 | 465 | 510 | 482 | 493 | 499 | 513 | 481 | 515 | 478 |
| 50 | 493 | 523 | 475 | 439 | 604 | 535 | 527 | 519 | 481 | 533 | 490 | 535 | 498 | 501 | 487 | 489 | 537 | 562 |
| 51 | 459 | 536 | 522 | 443 | 467 | 538 | 569 | 564 | 486 | 460 | 489 | 469 | 545 | 480 | 471 | 549 | 473 | 424 |
| 52 | 492 | 510 | 528 | 525 | 476 | 452 | 412 | 526 | 535 | 507 | 494 | 518 | 522 | 498 | 526 | 534 | 485 | 558 |
| 53 | 476 | 532 | 505 | 492 | 528 | 514 | 517 | 497 | 516 | 550 | 528 | 522 | 433 | 504 | 443 | 535 | 431 | 500 |
| 54 | 445 | 514 | 460 | 471 | 368 | 510 | 555 | 483 | 522 | 441 | 517 | 517 | 531 | 481 | 478 | 417 | 528 | 448 |
| 55 | 514 | 442 | 459 | 499 | 426 | 475 | 489 | 519 | 481 | 483 | 490 | 512 | 500 | 495 | 532 | 440 | 521 | 493 |
| 56 | 424 | 459 | 509 | 497 | 478 | 497 | 536 | 475 | 505 | 508 | 483 | 563 | 494 | 515 | 523 | 549 | 529 | 548 |
| 57 | 540 | 549 | 539 | 538 | 535 | 530 | 605 | 474 | 510 | 497 | 452 | 524 | 516 | 519 | 442 | 519 | 514 | 509 |
| 58 | 481 | 427 | 467 | 504 | 509 | 497 | 520 | 494 | 467 | 457 | 489 | 502 | 518 | 522 | 607 | 453 | 485 | 488 |
| 59 | 504 | 509 | 494 | 562 | 596 | 527 | 574 | 497 | 531 | 540 | 509 | 489 | 561 | 518 | 508 | 516 | 500 | 500 |
| 60 | 488 | 471 | 525 | 524 | 497 | 518 | 572 | 450 | 536 | 443 | 470 | 507 | 433 | 519 | 436 | 520 | 456 | 513 |
| 61 | 497 | 508 | 568 | 516 | 518 | 492 | 520 | 483 | 558 | 471 | 440 | 483 | 504 | 544 | 505 | 499 | 465 | 478 |
| 62 | 495 | 464 | 514 | 493 | 442 | 569 | 498 | 480 | 478 | 486 | 526 | 506 | 505 | 492 | 534 | 440 | 507 | 568 |
| 63 | 452 | 476 | 453 | 502 | 523 | 555 | 542 | 459 | 431 | 444 | 518 | 528 | 540 | 564 | 467 | 502 | 488 | 440 |

| input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 0 | 484 | 608 | 433 | 520 | 463 | 511 | 509 | 522 | 476 | 442 | 448 | 561 | 473 | 469 | 520 |
| 1 | 507 | 490 | 507 | 451 | 475 | 482 | 535 | 517 | 531 | 475 | 495 | 468 | 501 | 505 | 493 |
| 2 | 535 | 481 | 443 | 472 | 495 | 534 | 431 | 496 | 516 | 509 | 529 | 445 | 480 | 539 | 551 |
| 3 | 489 | 485 | 517 | 507 | 484 | 566 | 464 | 489 | 521 | 547 | 518 | 541 | 487 | 533 | 479 |
| 4 | 500 | 523 | 499 | 506 | 482 | 487 | 429 | 457 | 443 | 415 | 465 | 529 | 468 | 484 | 488 |
| 5 | 523 | 549 | 524 | 480 | 434 | 541 | 494 | 509 | 510 | 357 | 545 | 440 | 540 | 500 | 531 |
| 6 | 452 | 555 | 534 | 511 | 515 | 472 | 510 | 453 | 501 | 490 | 497 | 531 | 577 | 504 | 445 |
| 7 | 494 | 495 | 532 | 478 | 427 | 480 | 542 | 497 | 523 | 530 | 481 | 557 | 454 | 443 | 552 |
| 8 | 454 | 499 | 526 | 523 | 451 | 455 | 439 | 470 | 469 | 460 | 506 | 520 | 450 | 505 | 498 |
| 9 | 493 | 522 | 520 | 495 | 491 | 529 | 493 | 513 | 457 | 531 | 511 | 498 | 529 | 533 | 480 |
| 10 | 551 | 439 | 488 | 510 | 540 | 478 | 528 | 506 | 541 | 478 | 506 | 509 | 560 | 520 | 487 |
| 11 | 489 | 522 | 532 | 472 | 452 | 541 | 520 | 468 | 433 | 513 | 470 | 479 | 555 | 480 | 550 |
| 12 | 446 | 480 | 474 | 504 | 552 | 547 | 499 | 502 | 537 | 481 | 481 | 520 | 499 | 495 | 501 |
| 13 | 516 | 507 | 518 | 430 | 483 | 524 | 503 | 494 | 577 | 506 | 535 | 456 | 502 | 521 | 490 |
| 14 | 572 | 522 | 542 | 474 | 577 | 495 | 523 | 451 | 472 | 526 | 469 | 353 | 466 | 465 | 438 |
| 15 | 521 | 407 | 449 | 427 | 501 | 460 | 529 | 458 | 457 | 466 | 472 | 538 | 500 | 492 | 536 |

| input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 16 | 494 | 527 | 530 | 501 | 496 | 510 | 506 | 515 | 497 | 529 | 431 | 487 | 518 | 501 | 493 |
| 17 | 476 | 550 | 518 | 494 | 550 | 495 | 467 | 487 | 465 | 499 | 535 | 485 | 474 | 493 | 461 |
| 18 | 522 | 497 | 491 | 523 | 502 | 463 | 548 | 441 | 495 | 575 | 487 | 423 | 563 | 488 | 430 |
| 19 | 508 | 444 | 526 | 557 | 508 | 455 | 482 | 514 | 524 | 521 | 477 | 499 | 542 | 454 | 475 |
| 20 | 405 | 461 | 516 | 496 | 567 | 469 | 481 | 474 | 435 | 493 | 476 | 498 | 539 | 458 | 499 |
| 21 | 486 | 495 | 540 | 480 | 490 | 505 | 458 | 587 | 507 | 510 | 430 | 530 | 536 | 536 | 495 |
| 22 | 423 | 517 | 554 | 484 | 457 | 508 | 517 | 573 | 538 | 520 | 503 | 526 | 440 | 452 | 522 |
| 23 | 588 | 523 | 500 | 523 | 465 | 500 | 466 | 402 | 524 | 480 | 530 | 532 | 514 | 500 | 437 |
| 24 | 482 | 520 | 523 | 514 | 399 | 469 | 524 | 492 | 544 | 551 | 532 | 456 | 553 | 557 | 506 |
| 25 | 503 | 484 | 509 | 503 | 474 | 533 | 479 | 482 | 492 | 436 | 513 | 484 | 516 | 467 | 461 |
| 26 | 460 | 477 | 490 | 545 | 553 | 493 | 500 | 499 | 466 | 502 | 454 | 525 | 532 | 513 | 488 |
| 27 | 503 | 509 | 505 | 516 | 420 | 489 | 574 | 519 | 549 | 481 | 515 | 487 | 458 | 527 | 534 |
| 28 | 510 | 543 | 461 | 488 | 523 | 461 | 452 | 509 | 446 | 570 | 417 | 552 | 520 | 484 | 469 |
| 29 | 486 | 485 | 532 | 524 | 473 | 497 | 514 | 514 | 533 | 510 | 538 | 450 | 513 | 440 | 523 |
| 30 | 501 | 507 | 461 | 490 | 550 | 482 | 532 | 502 | 496 | 480 | 474 | 465 | 486 | 482 | 505 |
| 31 | 519 | 512 | 492 | 418 | 575 | 482 | 514 | 399 | 518 | 463 | 436 | 487 | 452 | 518 | 468 |

| input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 32 | 482 | 481 | 531 | 469 | 545 | 483 | 549 | 487 | 429 | 499 | 485 | 470 | 507 | 483 | 536 |
| 33 | 538 | 475 | 490 | 497 | 433 | 482 | 569 | 478 | 492 | 500 | 543 | 534 | 502 | 491 | 450 |
| 34 | 524 | 496 | 507 | 478 | 484 | 473 | 447 | 597 | 520 | 538 | 579 | 511 | 480 | 529 | 505 |
| 35 | 468 | 522 | 521 | 460 | 502 | 578 | 416 | 445 | 516 | 524 | 480 | 545 | 489 | 482 | 539 |
| 36 | 587 | 481 | 526 | 559 | 551 | 524 | 480 | 512 | 466 | 466 | 515 | 512 | 550 | 527 | 511 |
| 37 | 512 | 539 | 501 | 555 | 488 | 512 | 507 | 492 | 498 | 523 | 517 | 473 | 440 | 483 | 523 |
| 38 | 521 | 463 | 435 | 545 | 475 | 498 | 436 | 566 | 517 | 532 | 494 | 477 | 507 | 519 | 481 |
| 39 | 523 | 479 | 538 | 484 | 501 | 508 | 528 | 532 | 517 | 493 | 495 | 463 | 546 | 505 | 407 |
| 40 | 444 | 424 | 507 | 517 | 506 | 537 | 520 | 468 | 540 | 495 | 523 | 447 | 445 | 513 | 521 |
| 41 | 492 | 472 | 475 | 504 | 489 | 570 | 518 | 473 | 514 | 595 | 451 | 516 | 579 | 550 | 465 |
| 42 | 456 | 517 | 488 | 466 | 503 | 534 | 433 | 475 | 498 | 514 | 528 | 488 | 586 | 455 | 564 |
| 43 | 464 | 537 | 545 | 472 | 394 | 516 | 527 | 527 | 533 | 493 | 517 | 488 | 458 | 486 | 559 |
| 44 | 496 | 560 | 450 | 435 | 462 | 518 | 467 | 504 | 503 | 454 | 530 | 522 | 498 | 523 | 516 |
| 45 | 512 | 495 | 505 | 484 | 537 | 557 | 563 | 455 | 523 | 470 | 511 | 516 | 524 | 520 | 440 |
| 46 | 467 | 490 | 502 | 489 | 492 | 513 | 522 | 509 | 540 | 544 | 548 | 521 | 493 | 489 | 498 |
| 47 | 510 | 526 | 445 | 510 | 502 | 557 | 490 | 466 | 481 | 444 | 482 | 468 | 555 | 521 | 466 |

| input bit-no. ▼ | Number of the changes in the output bits (for 1000 Strict Avalanche tests) Output bit No. ▼ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 48 | 487 | 536 | 529 | 553 | 489 | 451 | 523 | 476 | 564 | 494 | 547 | 477 | 545 | 495 | 471 |
| 49 | 519 | 514 | 501 | 458 | 473 | 523 | 489 | 478 | 449 | 507 | 505 | 530 | 456 | 523 | 526 |
| 50 | 505 | 470 | 464 | 541 | 491 | 496 | 480 | 540 | 452 | 517 | 445 | 521 | 448 | 408 | 462 |
| 51 | 514 | 517 | 511 | 541 | 431 | 495 | 442 | 477 | 453 | 480 | 507 | 513 | 534 | 512 | 543 |
| 52 | 459 | 577 | 532 | 492 | 508 | 513 | 495 | 529 | 524 | 531 | 505 | 436 | 514 | 538 | 469 |
| 53 | 479 | 467 | 528 | 476 | 476 | 516 | 492 | 538 | 508 | 517 | 512 | 425 | 513 | 520 | 523 |
| 54 | 534 | 569 | 516 | 504 | 495 | 576 | 485 | 511 | 541 | 532 | 502 | 478 | 527 | 429 | 555 |
| 55 | 498 | 553 | 491 | 585 | 532 | 509 | 534 | 492 | 524 | 489 | 457 | 517 | 496 | 501 | 471 |
| 56 | 444 | 504 | 497 | 529 | 487 | 550 | 451 | 487 | 483 | 505 | 482 | 472 | 560 | 522 | 452 |
| 57 | 453 | 412 | 504 | 456 | 487 | 555 | 481 | 466 | 494 | 516 | 491 | 534 | 578 | 555 | 541 |
| 58 | 542 | 570 | 540 | 484 | 570 | 414 | 497 | 433 | 520 | 486 | 509 | 505 | 543 | 532 | 487 |
| 59 | 521 | 537 | 524 | 490 | 503 | 424 | 535 | 511 | 445 | 513 | 463 | 519 | 582 | 504 | 534 |
| 60 | 490 | 492 | 520 | 469 | 532 | 482 | 488 | 505 | 541 | 479 | 528 | 509 | 552 | 481 | 543 |
| 61 | 462 | 527 | 485 | 493 | 516 | 474 | 571 | 578 | 502 | 460 | 477 | 525 | 484 | 490 | 453 |
| 62 | 530 | 457 | 459 | 559 | 557 | 466 | 501 | 498 | 501 | 571 | 472 | 483 | 532 | 538 | 526 |
| 63 | 491 | 547 | 468 | 494 | 469 | 521 | 507 | 501 | 483 | 580 | 451 | 527 | 510 | 523 | 482 |