CERTIFICATE

This is to certify that the thesis titled "**Customizing the Cellular Message Encryption Algorithm**" submitted by **Mallika Tyagi** (2K2\EC\641), **Neha Gupta** (2K2\EC\649), **Pallavi Tyagi** (2K2\EC\654) and **Piyush Kharbanda** (2K2\EC\656) to the Department of **Electronics and Communications** in partial fulfillment of the requirements of the award of the degree of Bachelor of Engineering, is a bonafide record of the work carried out by them under my supervision and guidance.

The results embodied in this thesis have not been submitted to any other university or Institute for the award of any degree or diploma.

Ms. Rajeshwari Pandey
Dept. of Electronics and Communications
Delhi College of Engineering
Bawana Road, New Delhi- 110062

ACKNOWLEDGEMENT

We take this opportunity to acknowledge the encouragement and support given to us by our respected Project guide, Ms. Rajeshwari Pandey. We thank her for providing us with the opportunity to work on this project and making all the necessary resources available to us.

We would also like to express our gratitude to all the persons who have helped us along the way, in the least bit by thought or action.

Mallika Tyagi
2K2/EC/641

Neha Gupta
2K2/EC/649

Pallavi Tyagi
2K2/EC/654

Piyush Kharbanda
2K2/EC/656

INDEX

ABSTRACT

Real time applications like mobile handsets and the Personal Digital Assistants (PDA's) have become an important organ of our system design. Data security has gained utmost importance because of the rapid growth of the data communication industry and for military purposes. As the cellular telephony boomed, the need for security has increased: both for privacy and fraud prevention.

Since the birth of the cellular industry, security has been a major concern for both service providers and subscribers. Service providers are primarily concerned with security to prevent fraudulent operations such as cloning or subscription fraud, while subscribers are mainly concerned with privacy issues.

With the advent of second-generation digital technology platforms like TDMA/CDMA, operators were able to enhance their network security by using improved encryption algorithms and other means. CDMA2000 systems use the standardized CAVE (Cellular Authentication and Voice Encryption) algorithm for authentication purposes. The Cellular Message Encryption Algorithm (CMEA), a block cipher is used to encrypt the control channel. The CMEA has been broken and it has been proven that it is extremely vulnerable to cryptographic attacks. In the present thesis, the properties of CMEA that render it vulnerable have been identified and the algorithm has been accordingly modified. Further, a cryptanalysis of the new or customized algorithm has been carried out to prove that it is indeed secure to previous specialized attacks as well as standard attacks like linear and differential cryptanalysis.

CHAPTER 1
INTRODUCTION

INTRODUCTION

Cryptography is concerned with the introduction of schemes that should be able to withstand any abuse. Such schemes are constructed so as to maintain a desired functionality, even under malicious attempts aimed at making them deviate from their desired functionality.

Security of information results from the need for private transmission of both military and diplomatic messages. The ancient Greeks and Spartans enciphered their military messages. For the Chinese merely writing the message made it private since very few people knew the language. The first electronic computers were built during the Second World War to help with cracking codes. The first computers were physically massive and slow. But the advent of the transistor and the development of technology have made the computers smaller and faster.

Today's era of communication has increased the importance of financial data exchange, image processing, biometrics, etc. Thus there has been a shift in the modern day cryptology. Thus cryptology today not only provides authentication, data integrity and non repudiation, but has also the added task of providing security in menacing environments.

The design of cryptographic schemes is a daunting task. One cannot rely on intuition regarding the typical state of the environment. The adversary attacking the system will try to manipulate the environment into conducive states. In a nutshell the attacker will try to break a system in an unconventional way by adopting strategies which the designer may not have envisioned. A secured cryptographic scheme has to withstand such types of attacks. This tussle of the cryptographer (designers of cryptographic algorithms) and the cryptanalyst (those who practice the art and science of breaking codes) has continued for ages.

1.1 Motivation

With ever increasing growth of data communication in the field of E-commerce transactions, data security has gained utmost importance. Several cryptosystems like DES, RSA and AES have been developed to protect secured data. With the advent of wireless communication and other handheld devices like Personal Digital Assistants security provided by cryptographic algorithms has attained importance of new dimensions.

As cellular telephony industry has boomed, the need for security has increased: both for privacy and fraud prevention. Because all cellular communications are sent over a radio link, anyone with the appropriate receiver can passively eavesdrop on all cell phone transmissions in the area without fear of detection. The cellular telephony industry players in particular are especially concerned with fraud prevention.

Cryptographic mechanisms are one obvious way to combat cloning fraud, and indeed, the industry is turning to cryptography for protection. In 1992, the TR-45 working group within the Telecommunications Industry Association (TIA) developed a standard for integration of cryptographic technology into tomorrow's digital cellular systems, which has been updated at least once. Some of the most recent cellphones to hit the market already include these cryptographic protection mechanisms. The TIA standard [1] describes four cryptographic primitives for use in the CDMA2000 digital cellular systems:

- **CAVE, a mixing function, is intended for challenge-response authentication protocols and for key generation.**
- **A repeated XOR mask is applied to voice data for voice privacy.**
- **ORYX, a LSFR-based stream cipher intended for wireless data services.**
- **CMEA (Control Message Encryption Algorithm), a block cipher, is used to encrypt the control channel.**

The Voice Privacy Mask algorithms, the ORYX algorithm as well as CMEA have been found to be insecure. This thesis focuses on the weaknesses of the CMEA algorithm [2] since it is not used to protect voice communications; instead, it is intended to protect sensitive control data, such as the digits dialed by the cell phone user. A successful break of CMEA might reveal user calling patterns. Also sent CMEA-encrypted are digits dialed (all DTMF tones) by the remote endpoint and alphanumeric personal pages received by the cell phone user. Finally, compromise of the control channel contents could lead to any confidential data the user types on the keypad: calling card PIN numbers may be an especially widespread concern, and credit card numbers, bank account numbers, and voicemail PIN numbers. The proven insecurity of such widely used encryption algorithms once again raises a question mark over the practice of closed door encryption algorithm design.

1.2 Problem Definition

The aim of the present thesis is to improve the security of the CMEA algorithm. This thesis performs a thorough analysis of the weaknesses identified in [2] in the CMEA and proposes a modified version of the algorithm which we shall call the "Customized-CMEA" or "C-CMEA". Thus the thesis focuses on the following:

- **Identify the properties of the CMEA leading to its inherent insecurity. Modify the algorithm so as to remove the causes of the easy cryptanalysis of the CMEA.**
- **Carry out the cryptanalysis of the C-CMEA against specialized attacks [2] as well as standard cryptanalytic attacks.**
- **Implement the C-CMEA in hardware.**

1.3 Summary of our work

The work in the thesis may be summarized as follows:

1) Identifying the properties of the CMEA leading to its inherent insecurity. Modify the algorithm so as to remove the causes of the easy cryptanalysis of the CMEA:

The thesis focuses on the parts of the CMEA that lead to the cryptanalysis of the algorithm as given in [2]. The steps that cause these weaknesses have been studied, and accordingly the algorithm is modified so that these weaknesses are rectified. An effort has been made to ensure that the algorithm retains its original structure and only necessary changes have been made.

2) Carrying out the cryptanalysis of the C-CMEA against specialized attacks [2] as well as standard cryptanalytic attacks:

Proving the security of any new algorithm is a difficult task. First we verified both logically as well as computationally that the C-CMEA is indeed resistant to the types of attacks discussed in [2]. The next logical step was to test the algorithm's strength against standard attacks like linear and differential cryptanalysis. The confusion and diffusion properties were checked to make sure that the algorithm provided sufficient amount of confusion as well as diffusion as required by any good cryptographic algorithm. Also finer issues were kept in mind against future attacks like Meet in the Middle attack.

1.4 Thesis Overview

The thesis is organized in the following sections, including the introduction present in this section.

1) Survey

The evolution of cryptography has been briefed in section 2. The various techniques and principles of the art have been presented. The final objective is to explain the lineage of the current work in the vast and interesting world of ciphers.

2) Cellular Message Encryption Algorithm

Section 3 contains a description of the Cellular Message Encryption Algorithm (CMEA). It lists out some salient points of the algorithm and its weaknesses. It also describes the various attacks that have been successfully mounted on the algorithm in [2].

CHAPTER 2
CRYPTOGRAPHY – A SURVEY

2.1 Introduction - The development of modern day cryptography

Security of information results from the need for private transmission of both military and diplomatic messages. The need is as old as civilization itself. The art of keeping messages secret is called *cryptography*, and is practiced by *cryptographers*. Cryptography is used to protect information from illegal access if possible. The primitive operation of cryptography is called *encryption*. The operation transforms messages into representation that is meaningless for all parties other than the intended receiver. Almost all cryptosystems rely upon the difficulty of reversing the encryption transformation in order to provide security to communication. *Cryptanalysis* is the art and science of breaking the encrypted message. The branch of science encompassing both cryptography and cryptanalysis is cryptology and its practitioners are cryptologists. In short cryptology evolves from the long lasting tussle between the cryptographer and the cryptanalyst.

For many years cryptography was the exclusive domain of the military. After the world wars there was a shift of focus of cryptography as it became of interest to the research community in general. The development if the world of communications and the mass awareness of cryptography soon made it a very useful tool of modern day technology. A large number of cryptographic papers laid to the rebirth of the science. Horst Fiestel began the development of the Data Encryption Standard (DES) (which is a private key algorithm) and laid the foundation of Fiestel Networks in general. Martin Hellman and Whitefield Diffie developed the public key cryptography in 1975.

The modern day cryptographer does more than merely providing security by jumbling up the message. He has to look into the application areas that suit the present day world. The development of VLSI technology has made the once cumbersome computers faster and smaller. The modern day cryptographer has the added task of providing security amidst the conflicting requirements of throughput, power and area.

2.2 Cryptography – a few technical details

The aim of the cryptographer is to find methods to secure and authenticate messages. The original message is called the *plaintext* and the encrypted output is called the *ciphertext*. A secret key is employed to generate the cipher text from the plaintext. The process of converting the plaintext to the ciphertext is called *encryption* and the reverse is called *decryption*. The cryptographer tries to keep the message secret from the attacker or intruder. A cryptosystem is a communication system encompassing a message source, an encryptor, an insecure channel, a decryptor, a message destination and a secure key transfer mechanism. This is represented in figure 2.1

A ***Ciphertext only*** attack is an attack where the cryptanalyst has access to the ciphertexts generated using a given key but has no access to corresponding plaintext or the key.

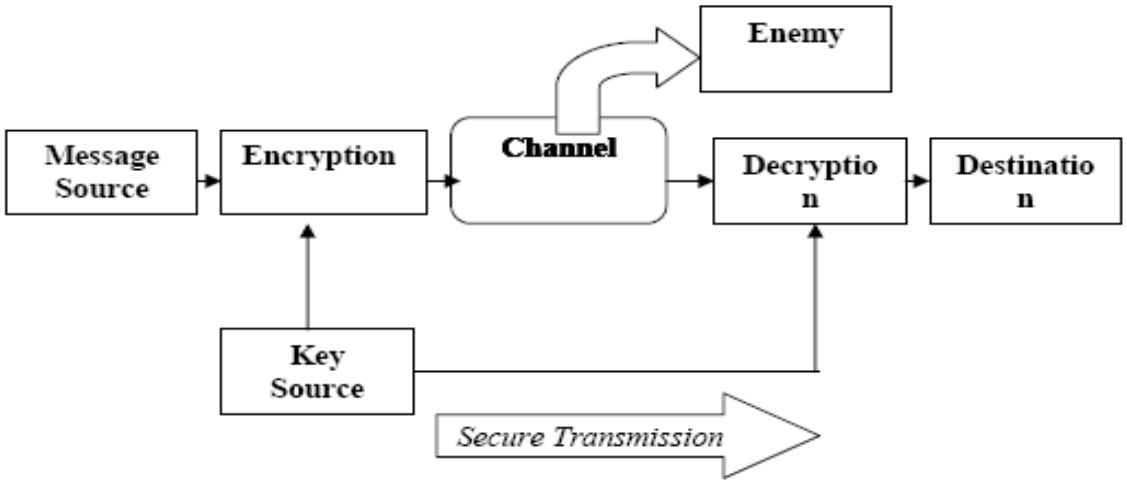


FIGURE 2.1 Secret Key Cryptosystem

A ***Known-plaintext*** attack is an attack where the cryptanalyst has access to ciphertexts as well as the corresponding plaintexts, but not the key.

A ***Chosen-plaintext*** attack is an attack where the cryptanalyst can choose plaintexts to be encrypted and has access to the resulting ciphertexts, again their purpose being to determine the key.

These attacks are measured against a worst case referred to as the ***brute force method***. This method is a trial and error approach, where by every possible key is tried until the correct one is found. Any attack that permits the recovery of the key faster than the brute force method is considered successful.

In modern cryptography we have two distinct types of ciphers

- **Private Key(or secret key or symmetric key) ciphers**
- **Public Key ciphers.**

These types differ in the manner in which the keys are shared. In private –key cryptography both the encryptor and the decryptor use the same key. Thus key must somehow be securely exchanged before secret key communication can begin.

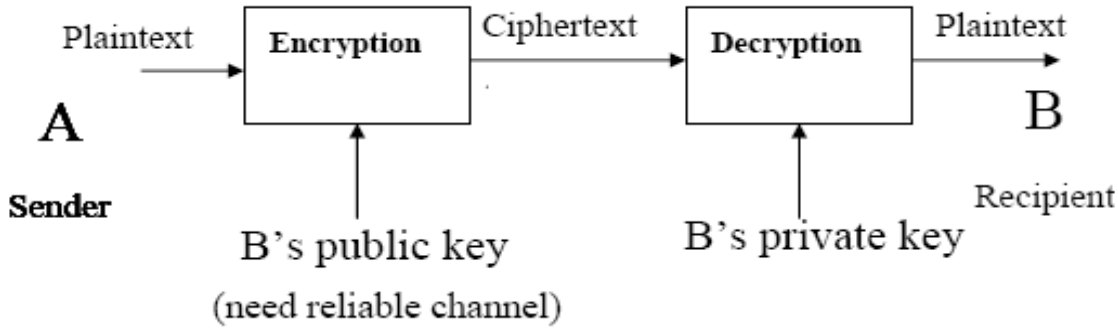


FIGURE 2.2 Public Key Cryptosystem

In public key cryptography the encryption and decryption keys are different. Thus in such algorithms we have a key pair consisting of:

- Private Key (or Symmetric key) which must be secret and is kept to decrypt messages.
- Public key which can be freely distributed and is used to encrypt messages.

Public and Private keys algorithms have complementary advantages and disadvantages. Thus they have their specific application areas. Private Key ciphers have higher data throughput but the key must remain secret at both ends. Thus in a large network there are many key pairs that must be managed. Sound cryptographic practice dictates that the key should be changed frequently for each communication session. The throughputs of most popular public key encryption methods are several methods of magnitudes smaller than the most symmetric key schemes. In a large network the number of key pairs to be maintained is much smaller and there is no need of frequent key changes. In practice public key cryptography is used for efficient key management while symmetric key encryption algorithms are used for bulk data encryption. Since the present thesis deals with private key algorithms, it has been dealt with exclusively in the subsequent section.

2.2.1 Private Key Cryptography

The private key algorithms can be divided into two types:

- **Block ciphers**
- **Stream ciphers**

Block ciphers operate on blocks of plaintexts and ciphertexts. Identical plaintext blocks always encrypt to the same cipher text blocks for a given key. The algorithm DES uses 64 bit block size where as Rijndael uses 128 bits. The CMEA is also a block cipher operating on n number of octets at a time where n>=2. The security of the data depends on the block sizes for the data and the key. As the key space increases, the probability of success of the adversary reduces increasing the security of the scheme. Stream ciphers operate on streams of plaintext one at a time. Thus a stream cipher may be imagined as a block cipher where the bloc length is one. The topic of our present work is block ciphers and we focus on such type of cryptographic schemes.

2.2.2 Block Ciphers

The block ciphers are a widely researched topic in the present crypto world. Security and efficient implementations are two of the most important design objectives of such ciphers. Shannon's principle of *confusion* and *diffusion* are applied in the designs of block ciphers. Confusion obscures the relationship between the plaintext and the ciphertext. This works to make the relationship between the statistics of the plaintexts and the ciphertexts as complicated as possible. Diffusion dissipates the redundancy of the plaintext by spreading it over the ciphertext. Product or iterated cipher is one where confusion and diffusion is achieved by the repeated application of the same single ciphers. This helps the iterated cipher achieve security goals while being easily implementable at the same time. The cipher structure which is repeated is called a *round*. In modern day cryptography there are two different types of confusion, a key dependent and a key independent function. Key dependent confusion is obtained by the bit wise exclusive-or of key bits and plaintexts. Non linear transformations are used to obtain key independent confusion. The non linear step is often implemented by means of a look up table called a *Substitution Box* or an *S-Box* The S-Box is one of the most important aspects of the design of ciphers. If the S-Box did not provide non-linearity to the cipher the cascading of the rounds could be represented by a single step. Thus with a linear S-box any number of rounds is equivalent to a single application of a different S-Box.

2.2.3 Cryptanalysis

The strength of a cipher can be measured only by its resistance to known cryptanalytic attacks. Knowledge of this technique is important from the point of view of crypto algorithm designs. Some of the well known cryptanalytic techniques are:

- **Differential Cryptanalysis.**
  In 1990 Eli Biham and Ali Shamir introduced the method. Let us consider a pair of known texts (x1 and x2) maintaining a fixed difference, measured by the bit

wise exclusive or of x1 and x2. Due to the nature of the round transform the corresponding difference in the cipher text depends upon the key. Thus, by encrypting a large number of pairs of plaintexts, all with a given difference and then examining the difference in the plaintexts one can gain knowledge about the key.

- **Linear Cryptanalysis.**

  Linear Cryptanalysis proposed by Mitsuru Matsui in 1993. The approach was based on linear approximations to the non-linear S-box. The basic premise is that the linear approximation of a non linear S-box will hold with a certain probability. By chaining such linear approximations one may approximate the entire cipher through a linear equation.

The cryptanalyst uses a combination of algebraic and statistical methods to evaluate the security of ciphers.

2.2.4 Avalanche Effect

A desirable property of any encryption algorithm is that a small change in either the plaintext or the key should produce a significant change in the cipher text. In particular, a change in one bit of the plaintext or one bit of the key should produce a change in many bits of the cipher text. If the change is small this might provide a way to reduce the size of the plaintext or the key space to be searched.

CHAPTER 3
CELLULAR MESSAGE ENCRYPTION ALGORITHM

The CMEA is used in CDMA2000 systems along with the Cellular Authentication and Voice Encryption (CAVE) Algorithm which is used to generate the CMEA keys. As mentioned earlier it is used to encrypt sensitive control data and hence its security is an important issue. The functional schematic of the various algorithms used in CDMA2000 systems[3] is given in figure 3.1.

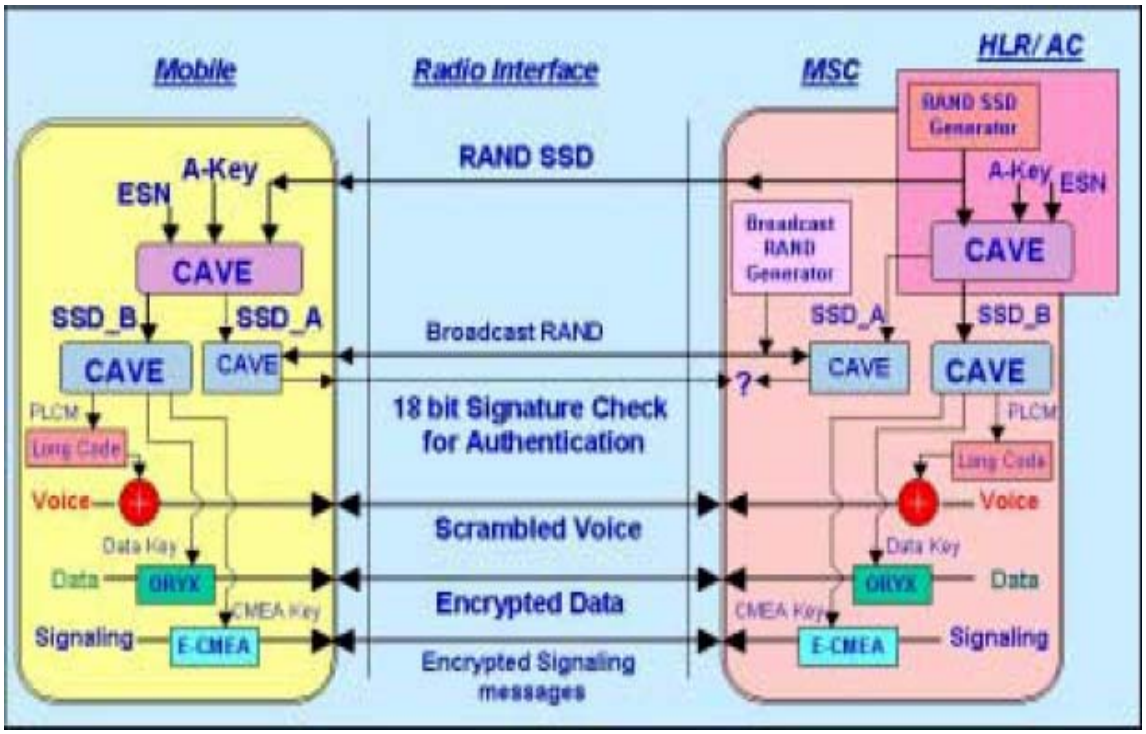


FIGURE 3.1: CDMA Encryption and Authentication

The CAVE algorithm [4], which is nothing but a nonlinear mixing function, as shown above ,is used to for authentication and as well as for generating keys for the ORYX as well as CMEA algorithm. Bruce and Wagner in their work [2] have shown that the CMEA is deeply flawed and have described an attack on CMEA which requires 40 to 80 known plaintexts, has time complexity about 2^{24} to 2^{32}, and finishes in minutes or hours of computation on a standard workstation. We shall

3.1 A description of CMEA

We describe the CMEA specification fully here for reference. CMEA is a byte oriented variable-width block cipher with a 64 bit key. Block sizes may be any number of bytes. In practice, US cellular telephony systems typically apply CMEA to 2-6 byte blocks, with the block size potentially varying without any key changes. CMEA is quite simple, and appears to be optimized for 8-bit microprocessors with severe resource limitations.

CMEA consists of three layers. The first layer performs one non-linear pass on the block, this effects left-to-right diffusion. The second layer is a purely linear, unkeyed operation intended to make changes propagate in the opposite direction. One can think of the second step as (roughly speaking) XORing the right half of the block onto the left half. The third layer performs a final nonlinear pass on the block from left to right; in fact, it is the inverse of the first layer.

CMEA obtains the non-linearity in the first and third layer from an 8-bit keyed lookup table known as the *T*-box. The *T*-box calculates its 8-bit output as

$$T(x) = C(((C(((C(((C((x \oplus K_0) + K_1) + x) \oplus K_2) + K_3) + x) \oplus K_4) + K_5) + x) \oplus K_6) + K_7) + x$$

Given input byte *x* and 8-byte key $K_{0...7}$. In this equation *C* is an unkeyed 8-bit lookup table known as the CaveTable; all operations are performed using 8-bit arithmetic.

hi \ lo	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.a	.b	.c	.d	.e	.f
0.	d9	23	5f	e6	ca	68	97	b0	7b	f2	0c	34	11	a5	8d	4e
1.	0a	46	77	8d	10	9f	5e	62	f1	34	ec	a5	c9	b3	d8	2b
2.	59	47	e3	d2	ff	ae	64	ca	15	8b	7d	38	21	bc	96	00
3.	49	56	23	15	97	e4	cb	6f	f2	70	3c	88	ba	d1	0d	ae
4.	e2	38	ba	44	9f	83	5d	1c	de	ab	c7	65	f1	76	09	20
5.	86	bd	0a	f1	3c	a7	29	93	cb	45	5f	e8	10	74	62	de
6.	b8	77	80	d1	12	26	ac	6d	e9	cf	f3	54	3a	0b	95	4e
7.	b1	30	a4	96	f8	57	49	8e	05	1f	62	7c	c3	2b	da	ed
8.	bb	86	0d	7a	97	13	6c	4e	51	30	e5	f2	2f	d8	c4	a9
9.	91	76	f0	17	43	38	29	84	a2	db	ef	65	5e	ca	0d	bc
a.	e7	fa	d8	81	6f	00	14	42	25	7c	5d	c9	9e	b6	33	ab
b.	5a	6f	9b	d9	fe	71	44	c5	37	a2	88	2d	00	b6	13	ec
c.	4e	96	a8	5a	b5	d7	c3	8d	3f	f2	ec	04	60	71	1b	29
d.	04	79	e3	c7	1b	66	81	4a	25	9d	dc	5f	3e	b0	f8	a2
e.	91	34	f6	5c	67	89	73	05	22	aa	cb	ee	bf	18	d0	4d
f.	f5	36	ae	01	2f	94	c3	49	8b	bd	58	12	e0	77	6c	da

FIGURE 3.2: The CAVE Table

We now provide a specification of CMEA. The algorithm encrypts an *n*-byte message $P_{0.....n-1}$ to a cipher text $C_{0,....,n-1}$ under the key $K_{0...7}$ as follows:

$y_0 \leftarrow 0$
for $i \leftarrow 0, \ldots, n-1$
$\quad P'_i \leftarrow P_i + T(y_i \oplus i)$
$\quad y_{i+1} \leftarrow y_i + P'_i$

for $i \leftarrow 0, \ldots, \lfloor \frac{n}{2} \rfloor - 1$
$\quad P''_i \leftarrow P'_i \oplus (P'_{n-1-i} \vee 1)$

$z_0 \leftarrow 0$
for $i \leftarrow 0, \ldots, n-1$
$\quad z_{i+1} \leftarrow z_i + P''_i$
$\quad C_i \leftarrow P''_i - T(z_i \oplus i)$

Here all operations are byte-wide arithmetic: + and - are addition and subtraction modulo 256, $\oplus$ represents a logical bitwise exclusive or, V represents a logical bitwise or, and the keyed *T* function is as described previously.

3.2 Observations

First, we list some preliminary observations made in [2]:

- CMEA is its own inverse. In other words, every key is a “weak key" (in the strict sense, from the DES nomenclature, of being self-inverse). This was apparently originally a design goal, for unknown reasons.

- CMEA is typically used to encrypt short blocks. Because the cellular telephony specification does not use random IVs, does not use block chaining modes, and encrypts short blocks under CMEA, codebook attacks could be a threat. On the other hand, the cell phone specifications require the CMEA key to be re-derived (using CAVE as a pseudo-random generator) for every call, so the amount of text required for a codebook attack may often be unavailable. (In a codebook attack, one obtains the encryption of every possible plaintext, records those pairs in a lookup table, and uses it to completely decrypt future messages without needing to know the key.) Codebooks attacks may still be possible though. practice. In some contexts, each digit dialed will be encrypted in a separate CMEA block (with fixed padding), because CMEA is used in ECB mode, the result is a simple substitution cipher on the digits 0-9. Techniques from classical cryptography may well suffice to recover useful information about the dialed digits, especially when side information is available.

- One bit of the plaintext leaks. The LSB (least-significant bit) of the ciphertext is the complement of the LSB of the plaintext.

- The *T*-box has some key equivalence classes. Simultaneously complementing the MSB (most significant bit) of K_0 and K_1 leaves the action of the *T*-box unchanged; the same holds for K_{2i} and K_{2i+1} for $i = 0, 1, 2, 3$. Therefore for the rest of the paper we take the MSBs of K_0, K_2, K_4, and K_6 to all be 0, without loss of generality, and we see that the effective key length of CMEA is at most 60 bits.

- Recovering the value of all 256 of the *T*-box entries suffices to break CMEA, even if the key $K_{0..7}$ is never recovered.

- The value of $T(0)$ occupies a position of special importance. $T(0)$ is always used to obtain $C0$ from $P0$. One cannot trivially predict where other *T*-box entries are likely to be used. Knowing $T(0)$ lets one learn the inputs to the *T*-box lookups that modify the second byte in the message.

- The CaveTable has a much skewed statistical distribution. It is not a permutation; 92 of the 256 possible 8-bit values never appear. Some values appear as many as four times. The distribution appears to be consistent with that of a random function. The skew in the CaveTable means that the *T*-box values are skewed, too: we know $T(i) - i$ must appear in the CaveTable, so for any input to the *T*-box, we can immediately rule out 92 possibilities for the corresponding *T*-box output without needing any knowledge of the CMEA key.

3.3 Attacks on CMEA

The attacks on CMEA have been briefed underneath:

3.3.1 A chosen-plaintext attack

CMEA is weak against chosen-plaintext attacks: one can recover all of the *T*box entries with about 338 chosen texts (on average) and very little work. This attack works on any fixed block length $n > 2$; the attacker is not assumed to have control over n.

The attack proceeds in two stages, first recovering $T(0)$, and then recovering the remainder of the *T*-box entries, the CMEA key itself is never identified. First, one learns $T(0)$ with (256 _ 92)=2 = 82 chosen plaintexts (on average). For each guess x at the value of $T(0)$, obtain the encryption of the message

$$P = (1 - x;\ 1 - x;\ 1 - x;\ ....),$$

e.g. the message P where each byte has the value $1 - x$, if the result is of the form $C = (-x;....)$ then we can conclude with high probability that indeed $T(0) = x$. False alarms occasionally occur, but they can be ruled out quickly in the second phase because of the skewed CaveTable distribution. Note that there are only 256 - 92 = 164 possible values of $T(0)$, since $T(0)$ must appear in the CaveTable, and therefore we expect to identify the correct value after about 164=2 = 82 trials, on average.

In the second phase of the attack, one learns all of the remaining *T*-box entries with 256 more chosen plaintexts. For each byte j, to learn the value of $T(j)$, let $k = ((n\text{-}1) \oplus j)\text{-}(n\text{-}2)$, where the desired blocks are n bytes long. Obtain the encryption of the message

$$P = (1 - T(0);\ 1 - T(0);....;\ 1 - T(0);\ k - T(0);\ 0),$$

If the result is of the form $C = (t\text{-}T(0);....)$, then we may conclude that $T(j) = t$, except for a possible error in the LSB. A more sophisticated analysis can resolve the uncertainty in the LSB of the *T*-box entries.

In practice, chosen-plaintext queries may be available in some special situations. Suppose the targeted cell phone user can be persuaded to a call a phone number under the

attacker's control perhaps a memorized survey, answering machine, or operator. The phone message the user receives might prompt the user to enter digits (chosen in advance by the attacker), thus silently enabling a chosen plaintext attack on CMEA. Alternatively, the phone message might send chosen DTMF tones to the targeted cell phone user, thus mounting chosen-plaintext queries at will.

3.3.2 A known-plaintext attack

We now describe a known plaintext attack on CMEA needing about 40-80 known texts. The attack assumes that each known plaintext is enciphered with a 3-byte block width. The (unoptimized) implementation has a time complexity of 224 to 232, and can be easily parallelized.

The cryptanalysis has two phases. The first phase gathers information about the *T*-box entries from the known CMEA encryptions, eliminating many possibilities for the values of each *T*-box output. In this way we reduce the problem to that of cryptanalysis of the *T*-box algorithm, given some partial information about *T*-box input/output pairs. In the second phase, we take advantage of the statistical biases in the CaveTable to cryptanalyze the *T*-box and recover the CMEA key $K_{0...7}$, using pruned search and meet-in-the-middle techniques to enhance performance.

The first phase is implemented as follows. Because *T*(0) occupies a position of special importance, we exhaustively search over the 164 possibilities for *T*(0). (Remember that *T*(0) must appear in the CaveTable, and so there are only 256_92 = 164 possibilities for it.) For each guess at *T*(0), we set up a 256 *X* 256 array $p_{i,j}$ which records for each *i, j* whether *T*(*i*) = *j* is possible. All values for *T*(*i*), *i* > 0 are initially listed as possible. Since *T*(*i*) - *i* is a CaveTable output and the CaveTable has an uneven distribution, we can immediately rule out 92 values for *T*(*i*).

Next, we gradually eliminate impossible values using the known texts as follows. The general idea is that each known plaintext/ciphertext pair lets us establish several implications of the form

$$T(0) = t_0 \ , T(i) = j \qquad => \qquad T(i') = j' \qquad (1)$$

If we have already eliminated $T(i') = j'$ as impossible, then we can conclude that $T(i) = j$ is also impossible via the contra positive of (1). In this way, we successively rule out more and more possibilities in the $p_{i\ ,j}$ array, until we either reach a contradiction (in which case we start over with another guess at $T(0)$) or until we run out of logical deductions to make (in which case we proceed to the second phase).

The second phase recovers the CMEA key from the information about T previously accumulated in the $p_{i\ ,j}$ array. Our simplest key recovery algorithm is based on pruned search. First, one guesses K_6 and K_7. Then, we peel of the effect of the last 1/4 of the T-box, and check whether the intermediate value is a possible CaveTable output. The intermediate value must always be one of the 164 possible CaveTable outputs when we find the correct K_6 ,K_7; because the CaveTable is so heavily skewed, incorrect K_6 ,K_7 guesses will usually be quickly identified by this test, if we have knowledge about a number of T-box entries. Next, one continues by guessing K_4, K_5, pruning the search as before, and continuing the pruned search until the entire key is recovered. This technique is very effective if enough information is available in the $p_{i,j}$ array. Unfortunately, pruned search very quickly becomes extremely computationally intensive if too few known texts are available: at each stage, too many candidates survive the pruning, and the search complexity grows exponentially.

We have a more sophisticated key recovery algorithm which can reduce the computation workload dramatically in these instances. The basic idea is that the T-box is subject to a classic meet-in-the-middle optimization: one can work halfway through the T-box given only $K_{0...3}$, and one can work backwards up to the middle given just $K_{4...7}$. This enables us to precompute a lookup table that contains the intermediate value corresponding to each

$K_{0...3}$ value. Then, we try each possible $K_{4...7}$ value, work backwards through some known *T*-box outputs, and look for a match in the precomputed lookup table. Of course the search pruning techniques can be applied to $K_{4...7}$ to further reduce the complexity of the meet-in-the-middle algorithm. The combination of pruned search and meet-in-the-middle cryptanalysis allows us to efficiently recover the entire CMEA key with as few as 40-80 known plaintexts.

3.4 Feasibility of the attacks

The known plaintext attack is much more devastating than the chosen plaintext attack described in Section 3.31. Chosen plaintext may be difficult to obtain in practice, but known plaintext is likely to be much easier to acquire.

There are a number of realistic ways that the required known plaintext can be collected in practice. Dialed digits are typically CMEA-encrypted with 3-byte blocks, typically each block will contain only one digit, and often the telephone number dialed will be known. DTMF tones sent on the line will usually be CMEA-encrypted. If the user can be persuaded to dial a number under adversarial control, using their calling card, then the DTMF tones and user-dialed digits will be known to the attacker, providing a ready source of known plaintext, after recovering the CMEA key in a known-plaintext attack, the attacker could decrypt the calling card number and make false calls billed to the victim's name. Furthermore, alphanumeric pages sent to cellular phones are becoming increasingly common, and alphanumeric pages are sent over the control channel. These pages may have a large known component, which will provide some known plaintext. It should be clear that known plaintext may be available from a number of potential sources.

CHAPTER 4
WEAKNESS of the CMEA ALGORITHM

Introduction

In this chapter, we shall analyze the properties of the CMEA which cause its weakness. The recovery of all the 256 values of the T-Box is equivalent to breaking of the cipher, so the strength of the T-Box requires special attention and has been treated subsequently in details in this and the next chapter.

4.1 Why is CMEA weak?

We have already made some observations in Chapter 3 regarding CMEA. There we pointed out some of the weaknesses in the algorithm. It should be noted that some of the properties lead to the successful breaking of the cipher, whereas others, though they give away some information do not contribute to the success of the two types of attacks given in Section 3.3. First let us consider the properties that lead to the successful cryptanalysis of the cipher.

4.1.1 Property 1:

If the plaintext is of the form :

$$P = \{1\text{-}x, 1\text{-}x, \ldots 1\text{-}x\}$$

And the ciphertext is of the from :

$$C = \{-x, ...\}$$

Then there is a very high probability that T(0)=x, where T(i) is the output of the T-Box corresponding to input 'i'.

Analysis:

$P'_0 = P_0 + T(0)$

$= 1 - x + T(0).$

If $T(0) = x$, we have $P'_0 = 1$.

Thus , $y_1 = y_0 + P'_0 = 0 + 1.$

Likewise,

$P'_1 = P_1 + T(1 \oplus 1)$

$= 1 + 1 = 2$

Thus , $y_2 = y_1 + P_1$

$= 1 + 1 = 2.$

Thus continuing we have

$P'_{n-1} = 1$

So, $P'' = P'_0 - T(0)$

$= - T(0) = - x$

Hence,

$C_0 = P''_0 - T(0)$

$= - T(0) = - x$

The probability when using the CaveTable is dependent on the fact that the initial guess for T(0) is correct and the possible number of trails is thus only (256-92)/2 = 82 on the average.

4.1.2 Property 2:

If the plaintext is of the form :

$P = \{1\text{-}T(0), 1\text{-}T(0),\ldots.., 1\text{-}T(0), k\text{-}T(0),0\}$

And the ciphertext is

$C = \{t\text{-}T(0),\ldots\}$

Where $k = ((n-1) \oplus j) - (n-2)$, then there is a very high probability that **t = T(j).**

Analysis:

Now $P'_{n-2} = P_{n-2} + T(y_{n-2} \oplus n\text{-}2)$

$= P_{n-2} + T(0)$, since $y_{n-2} = n-2$

$= k - T(0) + T(0) = k$

Using this fact,

$y_{n-1} = y_{n-2} + P'_{n-2}$

$= (n-2) + k$

$= (n-1) \oplus j.$

Therefore,

$P'_{n-1} = P_{n-1} + T(y_{n-1} \oplus n\text{-}1)$

$= 0 + T(j)$.

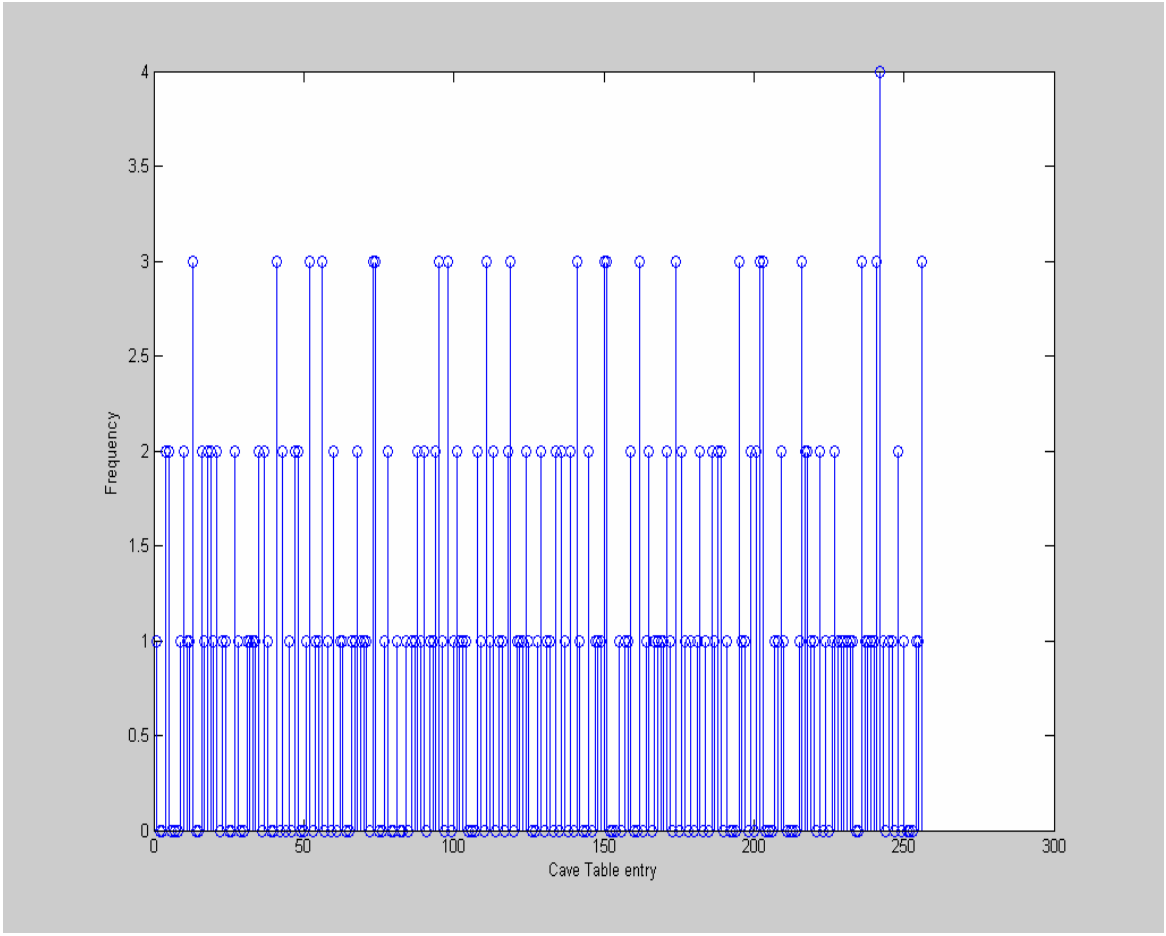
Thus, $C_0 = t - T(0) = P''_0 - T(0)$

$= P'_0 \oplus (P'_{n-1} \text{ V } 1) - T(0)$

Thus, $t = 1 \oplus (T(j) \text{ V1})$

$= T(j)$, with very high probability, with some confusion in the LSB.

4.1.3 Property 3:



The CMEA algorithm uses a very skewed CaveTable[2]. The CaveTable is not a permutation and 92 of the 256 values never occur. The CaveTable was shown in FIGURE 3.2. The frequency distribution of the CaveTable entries is given in FIGURE 4.1. We see that some of the values occur as many as 3 and 4 times. For every repeated values some value has to be missing. This property of the CaveTable considerably reduces the number of plain texts needed in both the attacks given in Section 3.3.

4.1.4 Property 4:

The CMEA algorithm uses a four round T-Box which can be subjected to a meet in the middle attack [2].

4.1.5 Property 5:

The Least Significant Bit of the ciphertext is always the complement of the Least Significant Bit of the plaintext. i.e. one bit always leaks.

Analysis:

Using original CMEA algorithm we get:

$$C_0=((P_0 + T(0)) \oplus (P'_2 \text{ or } 1)) - T(0)$$

P_0	T_0	$(P_0 \oplus T_0) \oplus 1$	C_0 (=P'_0)
0	0	1	1
0	1	0	1
1	0	0	0
1	1	1	0

TABLE 4.1 Truth Table for LSB of P_0

Consider LSB's: As far as (P(0)+T(0))'s LSB is concerned it is an equivalent exclusive or operation on P(0) and T(0)'s LSB's (neglecting carry). The LSB output can be obtained as shown in TABLE 4.1. We see that the LSB of ciphertext is always the complement of the LSB of plaintext,

4.1.6 Property 6:

The T-Box has some Key-Equivalence classes. As mentioned in Section 3.2, simultaneously complementing the Most Significant Bits of K_{2i} and K_{2i+1} for i = 0,1,2 leaves the action of the T-Box unchanged. This reduces the key length of the CMEA to

60 bits instead of 64 bits since we can assume the Most Significant Bits of K_0, K_2, K_4 and K_6 to all be 0 or 1.

Analysis :

In the T-Box Function :

$$T(X)=C(((C(((C(((C((X \oplus K_0)+K_1)+X) \oplus K_2)+K_3)+x) \oplus K_4)+K_5)+X) \oplus K_6)+K_7)+X,$$

consider the function : $f(X)=((X \oplus K0)+K1)$. Let us consider the Most Significant Bits of all the three terms. In order to consider the effect of the other Least Significant Bits., we include a carry term which can be zero or one. The invariance after simultaneous complementing the Most Significant Bits of the pairs of keys is because the carry is not being accounted for. In f(x), the MSB is being neglected. This can be seen from the TABLE 4.2. In case of both *a1* and *a2* after complementing the Most Significant Bits of K_0 and K_1 the difference is in the New Carry which is being neglected irrespective of the Carry (which is due the other less significant bit operations).

				CARRY = 0	CARRY = 1
	X	K0	K1	f(X)	f(X)
a1	0	0	0	0	1
	0	0	1	1	0
	0	1	0	1	0
a2	0	1	1	0	1
	1	0	0	1	0
	1	0	1	0	1
	1	1	0	0	1
	1	1	1	1	0

Table 4.2

4.2 Analysis of the properties

Using the above properties one can explain why the CMEA algorithm is weak against the chosen plaintext and the known plain text attacks. The causes of the attacks are enlisted below:

1. *Chosen Plain Text Attack:* The CMEA is weak against this attack because of properties 1 and 2 (and to an extent property 3).
2. *Known Plain Text Attack:* The CMEA is weak against this attack because of properties 3 and 4.

Properties 5 and 6 give away some information to the attacker which though not used in the cryptanalysis of the algorithm as described in Section 3.3, can be possible weak points of the algorithm. The next chapter deals with ways of overcoming some of these weaknesses in the CMEA.

CHAPTER 5

CUMOSTIZED CELLULAR MESSAGE ENCRYPTION ALGORITHM

Introduction

In this section the weaknesses that are given in Chapter 3 will be removed by elimination their causes as identified in Chapter 4. Each property that causes a dent in the CMEA's weakness will be dealt with so that none of the attacks that have been described in Chapter 3 work against the algorithm. In addition, ways to deal with some of the properties that could be weak points in the CMEA but are not used explicitly in the two attacks are also covered in this chapter.

5.1 Modifications in the CMEA

In this section the modifications needed in the CMEA corresponding to each of the Properties identified in Section 4.1 are presented.

5.1.1 Modification 1:

The update equation of P' needs to be changed so that Properties 1 and 2 do not work. Thus the modified equation is of the form:

$$P'_i = P_i + T(y_i \oplus f(i,n))$$

Such that as we vary i from 0 to n-1 (where n is the number of byte blocks in the plaintext) the T-Box is not predictably accessed. In the original CMEA property 1 exists because for a particular nature of the input plaintext and the key the T-Box was always referred at the point 0. So, the function f(i,n) should be such that the T-Box is accessed at different points. After considering several forms of the function f(i,n) the proposed function is f(i,n) = 2i%n, hence the update equation becomes :

$$P'_i = P_i + T(y_i \oplus 2i \% n)$$

Thus the algorithm is transferred to:

$$y_0 = 0$$

$$
\begin{aligned}
&for(i = 0; i < n; i++)\\
&\{\\
&\quad P'_i = P_i + T(y_i \oplus 2i\%n)\\
&\quad y_{i+1} = y_i + P'_i\\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&for(i = 0; i < \lfloor n/2 \rfloor; i++)\\
&\quad P''_i = P'_i \oplus (P'_{n-i-1} \bigvee 1)
\end{aligned}
$$

$$
\begin{aligned}
&z_0 = 0\\
&for(i = 0; i < n; i++)\\
&\{\\
&\quad z_{i+1} = z_i + P''_i\\
&\quad C_i = P''_i - T(z_i \oplus 2i\%n)\\
&\}
\end{aligned}
$$

5.1.2 Modification 2:

The CaveTable is replaced with the Advanced Encryption Standard's (AES) S-Box which can be efficiently implemented. Thus the distribution is no more skewed and all the possible 256 values appear as a possibility.

After modifications 1 and 2 the Chosen Plain text attack is nullified and the Known Plain text attack is also evaded easily. The following is the explanation:

For 50,000 variations of the key, plaintexts of the form (1 – T(0), 1 – T(0),…..,1 – T(0)) gives ciphertext of the form (-T(0),….) only 0.766 % of the time .However to prove that no similar attack of the same type is possible, we need a more rigorous approach :

Let the P_0 block of plain text be (1-x).

Thus $P'_0 = P_0 + T(y_0 \oplus 0)$

$= 1 – x + T(0 \oplus 0)$

$= 1- x + T(0)$

Let $x = T(0)$. So $P'_0 = 1$ and $y_1 = y_0 + P'_0 = 1$

Similarly ,

$P'_1 = P_1 + T(1 \oplus 2)$

$= P_1 + T(3)$.

Hence if we have $P_1 = 1 – x_1$ and let $x_1 = T(3)$.

So, $P'_1 = 1$ and $y_2 = y_1 + P'_1 = 1 + 1 = 2$

Likewise,

$P'_2 = P_2 + T(y_2 \oplus 4)$

$= P_2 + T(2 \oplus 4)$

$= 1 – x + T(6)$, if $P_2 = 1- x$

$= 1$, using the guess that $x = T(6)$.

$y_3 = y_2 + P'_2 = 2 + 1 = 3$

For the fourth block,

$P'_3 = P_3 + T(y_3 \oplus 6)$

$= P_3 + T(3 \oplus 6)$

$= 1 – x_2 + T(5)$, if $P_3 = 1- x_2$

$= 1$, using the guess $x_2 = T(5)$.

Thus if we have four blocks in the plaintext (without loss generality) then

$P''_0 = P'_0 \oplus P'_3 V 1) = 0$.

Thus for four input blocks if one obtains chosen plaintexts of the form

$P = (1-T(0), 1 – T(3), 1 – T(6), 1- T(5))$

Then the ciphertext is of the form $C = (-T(0),….)$

The number of trials on an average required is $(256^4)/2$ which is equivalent to a brute force search on the entire plain text space and is much larger than that required for original CMEA. Note that as the CaveTable has been replaced by the S-Box of the Rijndael-AES the number of possible values of each T-Box access is 256.

The above proof for block four length plaintext can be extended to plaintext of any length. In order to reduce the order of the plaintext required in the modified CMEA to carry out the above attacks, there should be a repetition in the point at which the T-Box is accessed . Supposed we have i = i1 and i = i2 for which the T-Box is accessed at the same point. Thus

$$i1 \oplus 2i\%n = i2 \oplus 2i \% n$$

or $$(i1 \oplus i2) = 2(i1 \oplus i2) \%n$$

If, $2(i1 \oplus i2) < n$, then the equation is possible only if i1 = i2, contradicting our initial assumption.

Also, if $2(i1 \oplus i2) = kn + r > n$ (where $k >= 1$ and $r < n$), we have

$$(kn + r)/2 = (kn + r) \% n = r,$$

$$\text{or } kn = r,$$

Which is not possible as r<n. Thus we have a contradiction and hence the T-Box is not accessed at the same point. Thus the attack does not work against the modified CMEA.

Also the number of plaintexts grows exponentially with the number of blocks. For an n byte block the number of chosen plaintexts is of the order of $256^{n.}$ Thus the number of plaintexts to be investigated is equal to that in a brute force search on the entire plaintext space. Such a large number of plaintext requirements make the attack ineffective against the modified CMEA.

As the CaveTable has been replaces by the AES S-Box the skewness of the CaveTable no longer exists. All of the 256 values may appear.

5.1.3 Modification 3:

The T-Box previously had only four rounds. The number of rounds in the T-Box has been increased to eight rounds to prevent meet-in-the-middle attack. For this purpose, the output of the four round T-Box is recycled again through the T-Box.

5.1.4 Modification 4:

The or with 1 in the second stage of the CMEA is removed. This removes the property that the LSB of the ciphertext is always the complement of the LSB of the plaintext. This can be explained by using the modified version and resorting to its truth table.

Using the modified version:

$$C_0 = ((P_0 + T(0)) \oplus P'_2) - T(0)$$

From the truth table (TABLE 5.1) we see that the Least Significant Bits of the plaintext and the cipher text are no longer related.

P[0]	T[0]	P'[2]	(P[0]xorT[0])XORp'[2])	C[0]
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	1
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	0

TABLE 5.1 Truth Table for LSB of plaintext and ciphertext

.

5.1.5 Modification 5:

From TABLE 4.2 we saw that by complementing the Most Significant Bits of K_0 and K_1 or a similar pair of odd and even keys, the output does not change. However the Carry out in the 2 cases is different. In order to incorporate the effect of the carry from the MSB, it was exclusive or-ed with the Least Significant Bit of the resultant f(X). The Truth Table showing the carry over is given in TABLE 5.2.

For 50,000 random combination of data and keys, we get the average number of changes in cipher texts (byte wise) by simultaneously changing the Most Significant Bits of all four pairs of Equivalence Class Keys as approximately 99.5 % as opposed to 0% of the times earlier.

				CARRY = 0		CARRY = 1	
	X	K0	K1	f(X)	New Carry	f(X)	New Carry
a1	0	0	0	0	0	1	0
	0	0	1	1	0	0	1
	0	1	0	1	0	0	1
a2	0	1	1	0	1	1	1
	1	0	0	1	0	0	1
	1	0	1	0	1	1	1
	1	1	0	0	0	1	0
	1	1	1	1	0	0	1

Table 5.2

5.2 Modifications to be kept

It was mentioned that the fourth and fifth modifications in the CMEA did not contribute towards preventing the Chosen plain text and the Known Plain text attacks. So it was decided to not include these changes while performing dedicated cryptanalysis of the new algorithm so as to not change the algorithm more than absolutely necessary.

In a nutshell, the first three modifications are absolutely necessary in order that the CMEA's use in CDMA2000 systems is continued without compromising with the customer's privacy, while the other modifications might be used in case an attack exploiting these properties is devised in the future. Also, since we have carried out dedicated cryptanalysis of the modified algorithm, we ourselves have checked that the properties do not make the cipher insecure. A reduction in the Key set from 2^{64} to 2^{60} does not make much difference practically and by knowing the Least Significant Bit of the ciphertext and hence the plaintext the attacker will not be able to break the algorithm.

5.3 Confusion and Diffusion in the Modified CMEA

This section of the chapter deals with the Diffusion and Confusion of the customized CMEA algorithm. Before continuing to more exhaustive and rather rigorous measures like differential and linear cryptanalysis, it is essential to first verify the confusion and diffusion property of the customized CMEA. For this purpose, the algorithm has been subjected to Avalanche Attack. A function has good Avalanche effect when a change in one bit of the input results in a change in half of the output bits.

Diffusion criteria require that a change in a single bit of the plain text should cause a change in several bits of the cipher text, keeping the key constant. In order to test the diffusion property the customized CMEA has been subjected on pairs of plain texts that differ by one bit. The number of bits affected should have a mean of n/2 where n is the number of bits in the cipher. In other words it is expected that for a good cipher approximately half of the output bits should be affected. The experiments have been performed on block size of three bytes (24bits). In FIGURE 5.1, the frequency of the number of bits affected has been plotted versus the number of bits affected. The plot shows that around 12 bits are affected for a maximum number of cases. Also the computed average is around 11.98. The plot shows that the algorithm provides sufficient diffusion property.

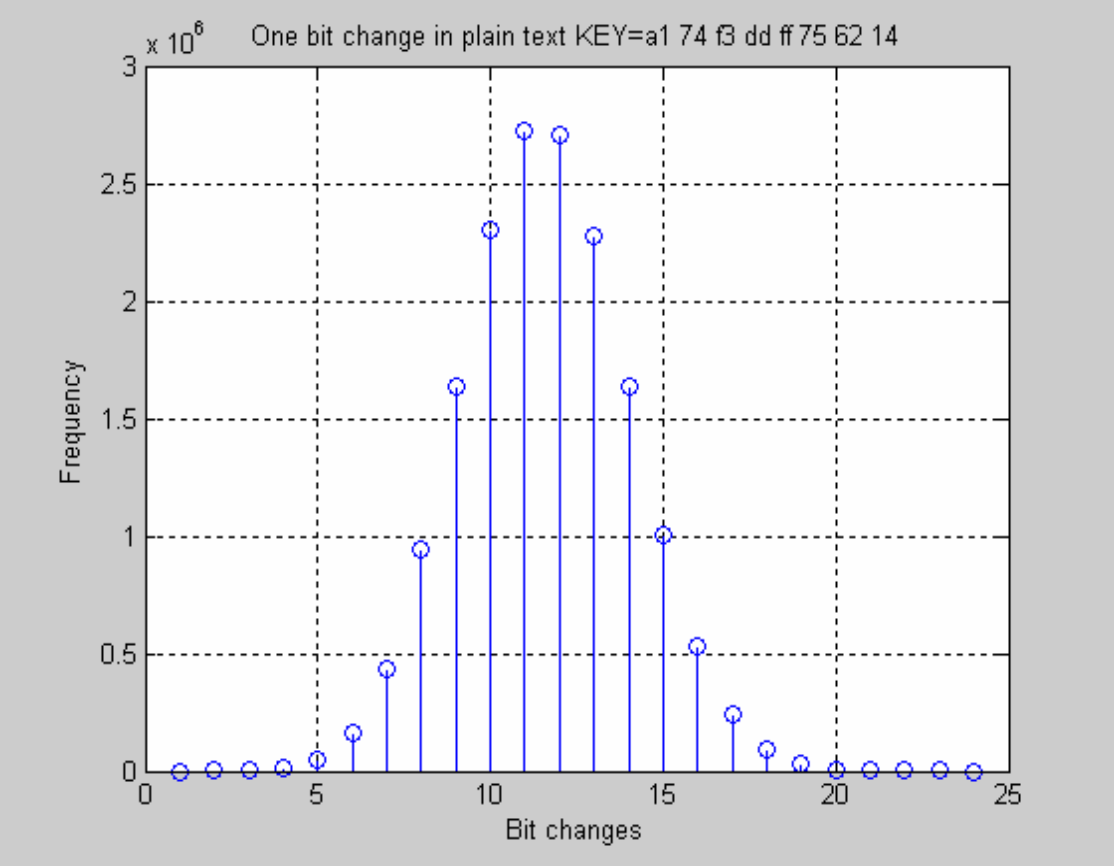


FIGURE 5.1 Diffusion in Customized CMEA

A confusion criterion requires that a change in a single bit in the key should cause a change in several bits of the ciphertext, keeping the plaintext constant. In order to test the confusion property of the customized CMEA has been used to encrypt plaintexts with pair of keys which differ in only one bit. The number of output bits affected according to the Avalanche Criterion should be around n/2 where n is the number of bits of the cipher. The experiments have been performed again on block size of three-bytes (twenty four bits). In FIGURE 5.2 the frequency of the number of bits affected has been plotted versus the number of bits affected. The plot shows that around 12 bits are affected for a maximum number of cases.

Also, the computed average is around 11.91. Thus the plots show that the confusion property is satisfied by the Customized CMEA.

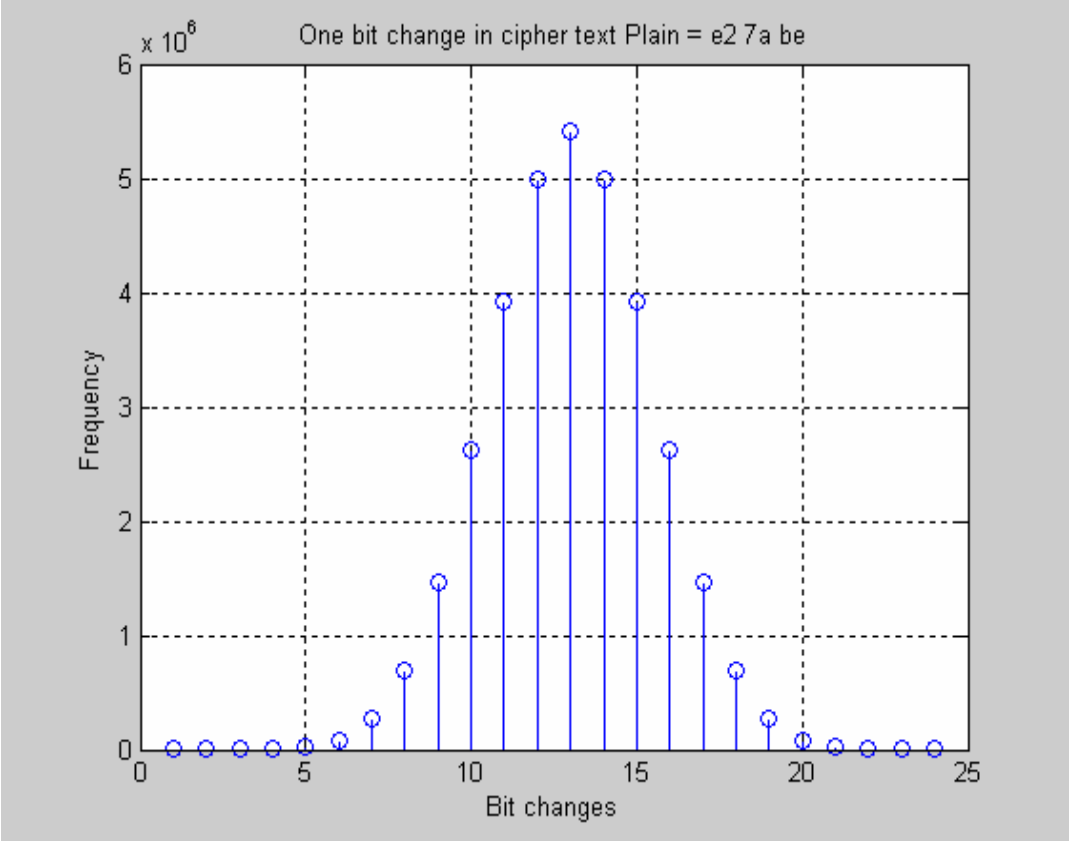


FIGURE 5.2 Diffusion in Customized CMEA

In the next chapter we shall consider dedicated Cryptanalysis of the Customized CMEA, wherein the Linear and Differential Cryptanalysis of the algorithm shall be carried out.

BIBLIOGRAPHY

[1] Cryptoanalysis of CMEA, David Wagner, University Of Californai Berkeley

[2] Cryptoanalysis of CMEA, Bruce scheineier Counterpane systems

[3] www.firewall.cx

[4] Network Security, William Stallings

APPENDIX
C- CODE

PROGRAM 1

```
//This program finds the number of keys that are affected by a given dx/dy pair and saves
them in the data1 - data9 files :small execution time
#include<stdio.h>

void filewrite(char *Filename,int dy[256][256],int Beg,int End)
{FILE *fp;
 int i,j;
 fp=fopen(Filename,"w");
 fprintf(fp,"   ");
 for(i=Beg;i<=End;i++)
   fprintf(fp,"%3d",i);
 fprintf(fp,"\n\n");
 for(i=0;i<256;i++)
   { fprintf(fp,"%3d",i);
     for(j=Beg;j<=End;j++)
       fprintf(fp,"%3d",dy[i][j]);
     fprintf(fp,"\n");
   }
 fclose(fp);
}

int main()
{unsigned char k,x1,x2,dx,y1,y2;
 int dy[256][256];
 int i,j,keycnt,sum,max,probreqd[256],keyprob[256][256],keytemp[256][256];
 for(i=0;i<256;i++)//Initialising
   for(j=0;j<256;j++)
     keyprob[i][j]=0;
 for(keycnt=0;keycnt<256;keycnt++)
{//For all possible keys
 k=(unsigned char)keycnt;
 for(i=0;i<256;i++)
   for(j=0;j<256;j++)
     dy[i][j]=0;
 //printf("\nKEY = %0x\n",k);
 for(j=0;j<256;j++)      //for all possible dx
   {dx=(unsigned char)j;
   //printf("Check ");
   //printf("%0x ",j);
   for(i=0;i<256;i++)
     {x1=(unsigned char)i;
     y1=x1+k;
     x2=x1^dx;
     y2=x2+k;
```

```
    dy[j][y2^y1]++;     //DDT entry
    }
  for(i=0;i<256;i++)
    if(dy[j][i])
      {//printf("fount   ");
      keyprob[j][i]++;
      }
 }
 }//end of key round
 /*printf("\n\n");
  for(i=0;i<256;i++)
  for(j=0;j<256;j++)
  printf("%d ",keyprob[i][j]);*/
  // printf("Key = %d, prob of dx = %d, dy= %d ==>%d
\n",i,chkprobdx,chkprobdy,probreqd[i]);

 filewrite("data1",keyprob,0,30);
 filewrite("data2",keyprob,31,60);
 filewrite("data3",keyprob,61,90);
 filewrite("data4",keyprob,91,120);
 filewrite("data5",keyprob,121,150);
 filewrite("data6",keyprob,151,180);
 filewrite("data7",keyprob,181,210);
 filewrite("data8",keyprob,211,240);
 filewrite("data9",keyprob,241,256);
 printf("\n\n Reqd = %d",keyprob[230][34]);
}
```

PROGRAM 2

```
//This program is used to demonstrate the T[0] vulnerability of the CMEA algorithm
//It can be shown that by using the methods described we can overcome the weakness by
making a very small change
//in the CMEA function.

#include<stdio.h>
#define PLAIN1 0x02
#define PLAIN2 0xd2
#define PLAIN3 0xf3
#define CMEAK0 0xa1
#define CMEAK1 0x74
#define CMEAK2 0x26
#define CMEAK3 0xdd
#define CMEAK4 0x02
#define CMEAK5 0x75
```

```
#define CMEAK6 0x69
#define CMEAK7 0x14
#define no_of_octets 3
unsigned char cmeakey[8];
void lastbits(unsigned char *,unsigned char *);
void assignkey(int);
unsigned char CaveTable[256] =
{0xd9, 0x23, 0x5f, 0xe6, 0xca, 0x68, 0x97, 0xb0,
0x7b, 0xf2, 0x0c, 0x34, 0x11, 0xa5, 0x8d, 0x4e,
0x0a, 0x46, 0x77, 0x8d, 0x10, 0x9f, 0x5e, 0x62,
0xf1, 0x34, 0xec, 0xa5, 0xc9, 0xb3, 0xd8, 0x2b,
0x59, 0x47, 0xe3, 0xd2, 0xff, 0xae, 0x64, 0xca,
0x15, 0x8b, 0x7d, 0x38, 0x21, 0xbc, 0x96, 0x00,
0x49, 0x56, 0x23, 0x15, 0x97, 0xe4, 0xcb, 0x6f,
0xf2, 0x70, 0x3c, 0x88, 0xba, 0xd1, 0x0d, 0xae,
0xe2, 0x38, 0xba, 0x44, 0x9f, 0x83, 0x5d, 0x1c,
0xde, 0xab, 0xc7, 0x65, 0xf1, 0x76, 0x09, 0x20,
0x86, 0xbd, 0x0a, 0xf1, 0x3c, 0xa7, 0x29, 0x93,
0xcb, 0x45, 0x5f, 0xe8, 0x10, 0x74, 0x62, 0xde,
0xb8, 0x77, 0x80, 0xd1, 0x12, 0x26, 0xac, 0x6d,
0xe9, 0xcf, 0xf3, 0x54, 0x3a, 0x0b, 0x95, 0x4e,
0xb1, 0x30, 0xa4, 0x96, 0xf8, 0x57, 0x49, 0x8e,
0x05, 0x1f, 0x62, 0x7c, 0xc3, 0x2b, 0xda, 0xed,
0xbb, 0x86, 0x0d, 0x7a, 0x97, 0x13, 0x6c, 0x4e,
0x51, 0x30, 0xe5, 0xf2, 0x2f, 0xd8, 0xc4, 0xa9,
0x91, 0x76, 0xf0, 0x17, 0x43, 0x38, 0x29, 0x84,
0xa2, 0xdb, 0xef, 0x65, 0x5e, 0xca, 0x0d, 0xbc,
0xe7, 0xfa, 0xd8, 0x81, 0x6f, 0x00, 0x14, 0x42,
0x25, 0x7c, 0x5d, 0xc9, 0x9e, 0xb6, 0x33, 0xab,
0x5a, 0x6f, 0x9b, 0xd9, 0xfe, 0x71, 0x44, 0xc5,
0x37, 0xa2, 0x88, 0x2d, 0x00, 0xb6, 0x13, 0xec,
0x4e, 0x96, 0xa8, 0x5a, 0xb5, 0xd7, 0xc3, 0x8d,
0x3f, 0xf2, 0xec, 0x04, 0x60, 0x71, 0x1b, 0x29,
0x04, 0x79, 0xe3, 0xc7, 0x1b, 0x66, 0x81, 0x4a,
0x25, 0x9d, 0xdc, 0x5f, 0x3e, 0xb0, 0xf8, 0xa2,
0x91, 0x34, 0xf6, 0x5c, 0x67, 0x89, 0x73, 0x05,
0x22, 0xaa, 0xcb, 0xee, 0xbf, 0x18, 0xd0, 0x4d,
0xf5, 0x36, 0xae, 0x01, 0x2f, 0x94, 0xc3, 0x49,
0x8b, 0xbd, 0x58, 0x12, 0xe0, 0x77, 0x6c, 0xda };

//////////////////Beginning of tbox//////////////////////
unsigned char tbox(unsigned char z){
        int k_index,i;
        unsigned char result,result1,result2;
        //printf(" %d size %d",cmeakey[1],sizeof(cmeakey));
        k_index = 0;
```

```
        result = z;

        for (i = 0; i < 4; i++)
        {
             result =result^cmeakey[k_index];
             result =result + cmeakey[k_index+1];
            /* result1 =result^cmeakey[k_index];
             result2 =result + cmeakey[k_index+1];
                if((result2<cmeakey[k_index+1])||(result2<result1))
                        result2^=0x01;
                result = z + CaveTable[result2];*/
                result = z + CaveTable[result];
                k_index += 2;
        }
        return(result);
}

//////////////////////End of tbox//////////////////////
void CMEA(unsigned char *msg_buf)
{
        const int octet_count=no_of_octets;
        int msg_index,half;
        unsigned char k,z;
        /* first manipulation (inverse of third) */
        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
             //k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
                msg_buf[msg_index] += k;
                z += msg_buf[msg_index];
        }
/* second manipulation (self-inverse) */

        half = octet_count/2;
        for (msg_index = 0; msg_index < half; msg_index++)
        {
                msg_buf[msg_index] ^=
                //msg_buf[octet_count - 1 - msg_index] | 0x00;
                msg_buf[octet_count - 1 - msg_index] | 0x01;
        }

/* third manipulation (inverse of first) */

        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
```

```
        {
            k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
            //k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
            z += msg_buf[msg_index];
            msg_buf[msg_index] -= k;
        }
}
//////////////////End of CMEA/////////////////////
void CMEA2(unsigned char *msg_buf)
{
        const int octet_count=no_of_octets;
        int msg_index,half;
        unsigned char k,z;
        /* first manipulation (inverse of third) */
        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
                k = tbox((unsigned char)((octet_count-z-1) ^ (msg_index & 0xff)));
                //k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
                msg_buf[msg_index] += k;
                z += msg_buf[msg_index];
        }
/* second manipulation (self-inverse) */

        half = octet_count/2;
        for (msg_index = 0; msg_index < half; msg_index++)
        {
                msg_buf[msg_index] ^=
                //msg_buf[octet_count - 1 - msg_index] | 0x00;
                msg_buf[octet_count - 1 - msg_index] | 0x01;
        }

/* third manipulation (inverse of first) */

        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
             //k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
             k = tbox((unsigned char)((octet_count-z-1) ^ (msg_index & 0xff)));
                z += msg_buf[msg_index];
                msg_buf[msg_index] -= k;
        }
}
//////////////////End of CMEA2/////////////////////

unsigned long int checker(unsigned char *msg,unsigned char *plain){
```

```
unsigned long int countmatches,total,countrite=0;
int i;
unsigned int count1,count2,count3;
countmatches=0;
for(count1=0;count1<=255;count1++){
        for(count2=0;count2<=255;count2++){
                for(count3=0;count3<=255;count3++){
                        msg[0]=plain[0]=(unsigned char)count3;
                        msg[1]=plain[1]=(unsigned char)count2;
                        msg[2]=plain[2]=(unsigned char)count1;
                        CMEA2(msg);
                        CMEA2(msg);
                        for(i=0;i<3;i++)
                                if(plain[i]==msg[i])
                                        countrite++;

                }
         }
    }
total=count1*count2*count3;
return total-countrite/3;

}
//////////////////////EO Check data////////////////////////////////

void assignkey(int swch)
{
        int i;
        if(swch==0)
        {       cmeakey[0]=CMEAK0;
                cmeakey[1]=CMEAK1;
                cmeakey[2]=CMEAK2;
                cmeakey[3]=CMEAK3;
                cmeakey[4]=CMEAK4;
                cmeakey[5]=CMEAK5;
                cmeakey[6]=CMEAK6;
                cmeakey[7]=CMEAK7;
        }
        else if(swch==1)
        {       //printf("\t\tKeychanged!!\t\t ");
                for(i=0;i<8;i++)
                        cmeakey[i]=rand()%256;
        }
        else if(swch==2)
        {     for(i=0;i<7;i+=2)
                {       //printf("\n%dth cmeakey previous =%0x",i,cmeakey[i]);
```

```
                        cmeakey[i]=cmeakey[i]^0x80;
                        cmeakey[i+1]=cmeakey[i+1]^0x80;
                        //printf("\t\t%dth cmeakey changed =%0x",i,cmeakey[i]);
                }
        }

}
////////////////////////////////////////EO assignkeys//////////////////////////////

void checkT0()
{       int i,j;
        unsigned char msg[3];
        unsigned char plain[3];
        unsigned char t0;
        long int countcodebreaks=0;
        int key_changes;
        for(key_changes=1;key_changes<=500000;key_changes++)
        {       assignkey(1);
                t0=tbox(0);
                for(i=0;i<no_of_octets;i++)
                        msg[i]=plain[i]=(1-t0)&0xff;
                /*msg[0]=plain[0]=msg[2]=plain[2]=1-tbox(2);
                msg[1]=plain[1]=1-tbox(0);*/
                CMEA2(msg);
                if(msg[0]==(unsigned char)(-t0))
                {       countcodebreaks++;
                        //printf("\nCode broken !! C[1-t0] = msg[0] = %d !!\n\n",msg[0]);
                }

        }

        printf("\n\nAmbiguity  = %d",countcodebreaks);
        printf("\n\nFraction of keys satisfying the equality =
%f",(float)countcodebreaks/key_changes);
}
////////////////////////////////////EO checkT0/////////////////////////////////////////////




int main()
{
 long unsigned int i,j;
 unsigned int count1,count2,count3;
```

```
unsigned char msg[no_of_octets];
unsigned char plain[no_of_octets];
int nochange,countchanges;


if(!(checker(msg,plain)))
        printf("\n\nSuccess!! \n");

else
        printf("\n\nDecrypted value not matching with plain text!!\n");



checkT0();
```

PROGRAM 3

```
//This program is used to demonstrate the LSB leakage of the CMEA algorithm
//It can be shown that by using the methods described we can overcome the weakness by
making a very small change
//in the CMEA function.

#include<stdio.h>
#define PLAIN1 0x02
#define PLAIN2 0xd2
#define PLAIN3 0xf3
#define CMEAK0 0xa1
#define CMEAK1 0x74
#define CMEAK2 0xf6
#define CMEAK3 0xdd
#define CMEAK4 0x02
#define CMEAK5 0x75
#define CMEAK6 0x69
#define CMEAK7 0x14
#define no_of_octets 3
unsigned char cmeakey[8];
void lastbits(unsigned char *,unsigned char *);
void assignkey(int);
unsigned char CaveTable[256] =
{0xd9, 0x23, 0x5f, 0xe6, 0xca, 0x68, 0x97, 0xb0,
0x7b, 0xf2, 0x0c, 0x34, 0x11, 0xa5, 0x8d, 0x4e,
0x0a, 0x46, 0x77, 0x8d, 0x10, 0x9f, 0x5e, 0x62,
0xf1, 0x34, 0xec, 0xa5, 0xc9, 0xb3, 0xd8, 0x2b,
0x59, 0x47, 0xe3, 0xd2, 0xff, 0xae, 0x64, 0xca,
0x15, 0x8b, 0x7d, 0x38, 0x21, 0xbc, 0x96, 0x00,
0x49, 0x56, 0x23, 0x15, 0x97, 0xe4, 0xcb, 0x6f,
```

```
0xf2, 0x70, 0x3c, 0x88, 0xba, 0xd1, 0x0d, 0xae,
0xe2, 0x38, 0xba, 0x44, 0x9f, 0x83, 0x5d, 0x1c,
0xde, 0xab, 0xc7, 0x65, 0xf1, 0x76, 0x09, 0x20,
0x86, 0xbd, 0x0a, 0xf1, 0x3c, 0xa7, 0x29, 0x93,
0xcb, 0x45, 0x5f, 0xe8, 0x10, 0x74, 0x62, 0xde,
0xb8, 0x77, 0x80, 0xd1, 0x12, 0x26, 0xac, 0x6d,
0xe9, 0xcf, 0xf3, 0x54, 0x3a, 0x0b, 0x95, 0x4e,
0xb1, 0x30, 0xa4, 0x96, 0xf8, 0x57, 0x49, 0x8e,
0x05, 0x1f, 0x62, 0x7c, 0xc3, 0x2b, 0xda, 0xed,
0xbb, 0x86, 0x0d, 0x7a, 0x97, 0x13, 0x6c, 0x4e,
0x51, 0x30, 0xe5, 0xf2, 0x2f, 0xd8, 0xc4, 0xa9,
0x91, 0x76, 0xf0, 0x17, 0x43, 0x38, 0x29, 0x84,
0xa2, 0xdb, 0xef, 0x65, 0x5e, 0xca, 0x0d, 0xbc,
0xe7, 0xfa, 0xd8, 0x81, 0x6f, 0x00, 0x14, 0x42,
0x25, 0x7c, 0x5d, 0xc9, 0x9e, 0xb6, 0x33, 0xab,
0x5a, 0x6f, 0x9b, 0xd9, 0xfe, 0x71, 0x44, 0xc5,
0x37, 0xa2, 0x88, 0x2d, 0x00, 0xb6, 0x13, 0xec,
0x4e, 0x96, 0xa8, 0x5a, 0xb5, 0xd7, 0xc3, 0x8d,
0x3f, 0xf2, 0xec, 0x04, 0x60, 0x71, 0x1b, 0x29,
0x04, 0x79, 0xe3, 0xc7, 0x1b, 0x66, 0x81, 0x4a,
0x25, 0x9d, 0xdc, 0x5f, 0x3e, 0xb0, 0xf8, 0xa2,
0x91, 0x34, 0xf6, 0x5c, 0x67, 0x89, 0x73, 0x05,
0x22, 0xaa, 0xcb, 0xee, 0xbf, 0x18, 0xd0, 0x4d,
0xf5, 0x36, 0xae, 0x01, 0x2f, 0x94, 0xc3, 0x49,
0x8b, 0xbd, 0x58, 0x12, 0xe0, 0x77, 0x6c, 0xda };

//////////////////Beginning of tbox//////////////////////
unsigned char tbox(unsigned char z){
        int k_index,i;
        unsigned char result,result1,result2;
        //printf(" %d size %d",cmeakey[1],sizeof(cmeakey));
        k_index = 0;
        result = z;

        for (i = 0; i < 4; i++)
        {
          /* result =result^cmeakey[k_index];
             result =result + cmeakey[k_index+1];*/
               result1 =result^cmeakey[k_index];
                    result2 =result + cmeakey[k_index+1];
                            if((result2<cmeakey[k_index+1])||(result2<result1))
                                    result2^=0x01;
                result = z + CaveTable[result2];
                // result = z + CaveTable[result];
                k_index += 2;
        }
```

```
        return(result);
}

//////////////////////End of tbox//////////////////////
void CMEA(unsigned char *msg_buf)
{
        const int octet_count=no_of_octets;
        int msg_index,half;
        unsigned char k,z;
        /* first manipulation (inverse of third) */
        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
            //k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
                msg_buf[msg_index] += k;
                z += msg_buf[msg_index];
        }
/* second manipulation (self-inverse) */

        half = octet_count/2;
        for (msg_index = 0; msg_index < half; msg_index++)
        {
                msg_buf[msg_index] ^=
                //msg_buf[octet_count - 1 - msg_index] | 0x00;
                msg_buf[octet_count - 1 - msg_index] | 0x01;
        }

/* third manipulation (inverse of first) */

        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
             k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
             //k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
             z += msg_buf[msg_index];
             msg_buf[msg_index] -= k;
        }
}
//////////////////End of CMEA/////////////////////
void CMEA2(unsigned char *msg_buf)
{
        const int octet_count=no_of_octets;
        int msg_index,half;
        unsigned char k,z;
        /* first manipulation (inverse of third) */
```

```
        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
                k = tbox((unsigned char)((octet_count-z-1) ^ (msg_index & 0xff)));
                //k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
                msg_buf[msg_index] += k;
                z += msg_buf[msg_index];
        }
/* second manipulation (self-inverse) */

        half = octet_count/2;
        for (msg_index = 0; msg_index < half; msg_index++)
        {
                msg_buf[msg_index] ^=
                msg_buf[octet_count - 1 - msg_index] | 0x00;
                //msg_buf[octet_count - 1 - msg_index] | 0x01;
        }

/* third manipulation (inverse of first) */

        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
             //k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
             k = tbox((unsigned char)((octet_count-z-1) ^ (msg_index & 0xff)));
                z += msg_buf[msg_index];
                msg_buf[msg_index] -= k;
        }
}
//////////////////End of CMEA2//////////////////////

unsigned long int checker(unsigned char *msg,unsigned char *plain){
unsigned long int countmatches,total,countrite=0;
int i;
unsigned int count1,count2,count3;
countmatches=0;
for(count1=0;count1<=0;count1++){
        for(count2=0;count2<=0;count2++){
                for(count3=0;count3<=0;count3++){
                        msg[0]=plain[0]=(unsigned char)count3;
                        msg[1]=plain[1]=(unsigned char)count2;
                        msg[2]=plain[2]=(unsigned char)count1;
                        CMEA2(msg);
                        CMEA2(msg);
                        for(i=0;i<3;i++)
                                if(plain[i]==msg[i])
```

```
                                        countrite++;

                        }
                }
        }
total=count1*count2*count3;
return total-countrite/3;

}
//////////////////////EO Check data////////////////////////////////


void assignkey(int swch)
{
        int i;
        if(swch==0)
        {       cmeakey[0]=CMEAK0;
                cmeakey[1]=CMEAK1;
                cmeakey[2]=CMEAK2;
                cmeakey[3]=CMEAK3;
                cmeakey[4]=CMEAK4;
                cmeakey[5]=CMEAK5;
                cmeakey[6]=CMEAK6;
                cmeakey[7]=CMEAK7;
        }
        else if(swch==1)
        {       //printf("\t\tKeychanged!!\t\t ");
                for(i=0;i<8;i++)
                        cmeakey[i]=rand()%256;
        }
        else if(swch==2)
        {     for(i=0;i<7;i+=2)
                {       //printf("\n%dth cmeakey previous =%0x",i,cmeakey[i]);
                        cmeakey[i]=cmeakey[i]^0x80;
                        cmeakey[i+1]=cmeakey[i+1]^0x80;
                        //printf("\t\t%dth cmeakey changed =%0x",i,cmeakey[i]);
                }
        }

}
//////////////////////////////////////EO assignkeys////////////////////////////

void checkT0()
{     int i,j;
        unsigned char msg[3];
        unsigned char plain[3];
```

```
        unsigned char t0;
        long int countcodebreaks=0;
        int key_changes;
        for(key_changes=1;key_changes<=50000;key_changes++)
        {       assignkey(1);
                t0=tbox(0);
                for(i=0;i<no_of_octets;i++)
                        msg[i]=plain[i]=(1-t0)&0xff;
                /*msg[0]=plain[0]=msg[2]=plain[2]=1-tbox(2);
                msg[1]=plain[1]=1-tbox(0);*/
                CMEA2(msg);
                if(msg[0]==(unsigned char)(-t0))
                {       countcodebreaks++;
                        printf("\nCode broken !! C[1-t0] = msg[0] = %d !!\n\n",msg[0]);
                }

        }

        printf("\n\nAmbiguity  = %d",countcodebreaks);
        printf("\n\nFraction of keys satisfying the equality =
%f",(float)countcodebreaks/key_changes);
}
////////////////////////////////EO checkT0/////////////////////////////////////////


void Keyunchanged(unsigned char *msg)
{       int i,j;
        unsigned long int suma,sumb;
        unsigned char msg2[3],plain[3];
        int temp,temp2;
        unsigned long int count1,count2,count3,counta,countb;
        printf("\n\nKEY UNCHANGED\n\n");
        printf("\tPlaintext\tMine\tCMEA\n");
        suma=sumb=0;
        for(count1=0;count1<=255;count1++)
                for(count2=0;count2<=255;count2++)
                        for(count3=0;count3<=255;count3++){
                                msg[0]=plain[0]=msg2[0]=(unsigned char)count3;
                                msg[1]=plain[1]=msg2[1]=(unsigned char)count2;
                                msg[2]=plain[2]=msg2[2]=(unsigned char)count1;
                                CMEA(msg);
                                CMEA2(msg2);
                            //  printf("\n");
                                for(i=0;i<no_of_octets;i++)
                                {       //printf("\n%d octet :
%0x\t%0x\t%0x",i,plain[i],msg[i],msg2[i]);
```

```
                                        temp = plain[i]^msg[i];
                                        temp2 = plain[i]^msg2[i];
                                        counta=countb=0;
                                        for(j=0;j<8;j++)
                                        {       if(temp&0x01)
                                                        counta++;
                                                if(temp2&0x01)
                                                        countb++;
                                                temp=temp>>1;
                                                temp2=temp2>>1;
                                        }
                                        suma+=counta;
                                        sumb+=countb;
                                        //printf("\t%ld\t%ld\n",counta,countb);
                                }

                        }
                printf("\n\nTotal and average bit changes in CAVE = %ld,
%lf",suma,(float)suma/(count1*count2*count3));
                printf("\nTotal and average bit changes in NEW =  %ld,
%lf",sumb,(float)sumb/(count1*count2*count3));
                printf("\nPercentage fall = %lf",(float)(suma-sumb)*100/suma);
}
//////////////////////EO Keyunchanged/////////////////////

  void Plainunchanged(unsigned char *msg)
{       unsigned long int i0,i1,i2,i3,i4,i5,i6,i7;
        int i,j,temp,temp2;
        long unsigned int suma,sumb,counta,countb;
        unsigned char msg2[3],plain[3];
        suma=sumb=0;
        for(i=0;i<3;i++)
                plain[i]=msg2[i]=msg[i];
        for(i0=0;i0<=1;i0++)
        for(i1=0;i1<=1;i1++)
        for(i2=0;i2<=1;i2++)
        for(i3=0;i3<=1;i3++)
        for(i4=0;i4<=1;i4++)
        for(i5=0;i5<=1;i5++)
        for(i6=0;i6<=255;i6++)
        for(i7=0;i7<=255;i7++)
        {       cmeakey[0]=i0;
                cmeakey[1]=i1;
                cmeakey[2]=i2;
                cmeakey[3]=i3;
                cmeakey[4]=i4;
```

```
                cmeakey[5]=i5;
                cmeakey[6]=i6;
                cmeakey[7]=i7;
                CMEA(msg);
                CMEA2(msg2);
            //  printf("\n");
                for(i=0;i<no_of_octets;i++)
                {       //printf("%d octet : %0x\t%0x\t%0x",i,plain[i],msg[i],msg2[i]);
                        temp = plain[i]^msg[i];
                        temp2 = plain[i]^msg2[i];
                        counta=countb=0;
                        for(j=0;j<8;j++)
                        {       if(temp&0x01)
                                counta++;
                                if(temp2&0x01)
                                countb++;
                                temp=temp>>1;
                                temp2=temp2>>1;
                        }
                        suma+=counta;
                        sumb+=countb;
                        //printf("\t%ld\t%ld\n",counta,countb);
                }
        }
        printf("\n\nTotal and average bit changes in CAVE = %ld,
%lf",suma,(float)suma/(i0*i1*i2*i3*i4*i5*i6*i7));
        printf("\nTotal and average bit changes in NEW =  %ld,
%lf",sumb,(float)sumb/(i0*i1*i2*i3*i4*i5*i6*i7));
        if(sumb>suma)
                printf("\nPercentage rise = %lf%%",((float)(sumb-suma)*100/suma));
        else
                printf("\nPercentage fall = %lf%%",((float)(suma-sumb)*100/suma));
}
////////////////////////////////EO keyunchanged/////////////



int main()
{
 long unsigned int i,j;
 unsigned int count1,count2,count3;
 unsigned char msg[no_of_octets];
 unsigned char plain[no_of_octets];
 int nochange,countchanges;
```

```
/*
  if(!(checker(msg,plain)))
        printf("\n\nSuccess!! \n");

  else
        printf("\n\nDecrypted value not matching with plain text!!\n");
  checkT0();
*/
Plainunchanged(msg);
// Keyunchanged(msg);
  }

/*

int switchmsbkey(unsigned char *msg,unsigned char *plain)
{       unsigned char msg2[no_of_octets];
        long int i,mismatch;
        for(i=0;i<no_of_octets;i++)
             msg2[i]=msg[i];
        assignkey(1);
        CMEA(msg);
        assignkey(2);
        CMEA(msg2);
        mismatch=0;
        for(i=0;i<no_of_octets;i++)
                if(msg[i]!=msg2[i])
                        mismatch++;
        if(mismatch>2)
        {       //printf("\n\nValue changed by changing MSBs!!");
                return 1;
        }
        else
        {       //printf("\n\nNo value change...try again!! \n");
                return 0;
        }
}
//////////////////////EO switchmsbkey//////////////////////



void Nonlinearcheck()
{       const long int no_of_terms=24;
        unsigned char msg1[3],msg2[3],temp,cryptedxor[3];
        int i,j,delta_p,delta_c,distribution[no_of_terms][no_of_terms];
        unsigned long int
i1,j1,count1,count2,count3,countcipher,countplain;//countdetection=0;
```

```
	for(j1=0;j1<no_of_terms;j1++)
		for(i1=0;i1<no_of_terms;i1++)
			distribution[i1][j1]=0;
	msg1[0]=0xb1;msg1[1]=0x1c;msg1[2]=0x15;msg2[0]=0x00;msg2[1]=0x0;msg2[
2]=0x0;
	for(count1=0;count1<6;count1++)
		for(count2=0;count2<6;count2++)
			for(count3=0;count3<256;count3++)
			{	msg1[0]=count1;msg1[1]=count2;msg1[2]=count3;
				countplain=countcipher=0;
				for(i=0;i<no_of_octets;i++)
				{	temp=msg1[i]^msg2[i];
					//printf("%d octet :
%0x\t%0x\t%0x",i,plain[i],msg[i],msg2[i]);

					for(j=0;j<8;j++)
					{	if(temp&0x01)
						countplain++;
						temp=temp>>1;
					}
				}
				CMEA2(msg1);
				CMEA2(msg2);
				//CMEA2(msg1xormsg2);
				//printf("\t");
				for(i=0;i<no_of_octets;i++)
				{	temp=msg1[i]^msg2[i];
					for(j=0;j<8;j++)
					{	if(temp&0x01)
							countcipher++;
						temp=temp>>1;
					}
				}
				distribution[countplain][countcipher]++;
				//printf("\n");
			}
	printf("\n  ");
	for(i=0;i<no_of_terms;i++)
		if(i<=10)
			printf(" %d  ",i);
		else printf("%d  ",i);
	for(i1=0;i1<no_of_terms;i1++)
	{	printf("\n%d",i1);
		for(j1=0;j1<no_of_terms;j1++)
			printf("  %d ",distribution[i1][j1]);
```

```
	}
		//printf("\n\n%d",countdetection);
}

//////////////////////////////EO Nonlinearcheck/////////////
void Nonlinearcheck2()
{	unsigned char msg1[3],msg2[3],msg1xormsg2[3],cryptedxor[3],temp[3];
	int i,j;
	unsigned long int count1,count2,count3,countnonlinear=0;

	msg1[0]=0xb1;msg1[1]=0x1c;msg1[2]=0x15;msg2[0]=0xa0;msg2[1]=0xab;msg2
[2]=0xca;
	for(count1=0;count1<256;count1++)
		for(count2=0;count2<256;count2++)
			for(count3=0;count3<256;count3++)
			{	msg1[0]=count1;msg1[1]=count2;msg1[2]=count3;
	//for(i=0;i<3;i++)
	//	printf("\n%0x\n\n",msg1xormsg2[i]=msg1[i]^msg2[i]);
				for(i=0;i<3;i++)
					msg1xormsg2[i]=msg1[i]^msg2[i];
				CMEA(msg1);
				CMEA(msg2);
				CMEA(msg1xormsg2);
				//printf("\t");
				for(i=0;i<3;i++)

				{	cryptedxor[i]=msg1[i]^msg2[i];
		//printf("\nMsg1= %0x \tMsg2= %0x\tXORed msges=%0x\t
CryptedXORs=%0x",msg1[i],msg2[i],cryptedxor[i],msg1xormsg2[i]);
					if(!(cryptedxor[i]^msg1xormsg2[i]))
					{	//printf(" Linearity Alert %d!!!! ",i);
						countnonlinear++;
					}
				}
	//printf("\n");
			}
	printf("\n\nNo of non linearities = %ld out of
%ld",countnonlinear,count1*count2*count3);

}
//////////////////////////////EO Nonlinearcheck2/////////////
```

PROGRAM 4

//This program is used to demonstrate the Key_Equivalence_classes of the CMEA algorithm

```
//It can be shown that by using the methods described we can overcome the weakness by
making
// a very small change in the Tbox function.

#include<stdio.h>
#define PLAIN1 0x02
#define PLAIN2 0xd2
#define PLAIN3 0xf3
#define CMEAK0 0xa1
#define CMEAK1 0x74
#define CMEAK2 0x26
#define CMEAK3 0xdd
#define CMEAK4 0x02
#define CMEAK5 0x75
#define CMEAK6 0x69
#define CMEAK7 0x14
#define no_of_octets 3
unsigned char cmeakey[8];
void lastbits(unsigned char *,unsigned char *);
void assignkey(int);
unsigned char CaveTable[256] =
{0xd9, 0x23, 0x5f, 0xe6, 0xca, 0x68, 0x97, 0xb0,
0x7b, 0xf2, 0x0c, 0x34, 0x11, 0xa5, 0x8d, 0x4e,
0x0a, 0x46, 0x77, 0x8d, 0x10, 0x9f, 0x5e, 0x62,
0xf1, 0x34, 0xec, 0xa5, 0xc9, 0xb3, 0xd8, 0x2b,
0x59, 0x47, 0xe3, 0xd2, 0xff, 0xae, 0x64, 0xca,
0x15, 0x8b, 0x7d, 0x38, 0x21, 0xbc, 0x96, 0x00,
0x49, 0x56, 0x23, 0x15, 0x97, 0xe4, 0xcb, 0x6f,
0xf2, 0x70, 0x3c, 0x88, 0xba, 0xd1, 0x0d, 0xae,
0xe2, 0x38, 0xba, 0x44, 0x9f, 0x83, 0x5d, 0x1c,
0xde, 0xab, 0xc7, 0x65, 0xf1, 0x76, 0x09, 0x20,
0x86, 0xbd, 0x0a, 0xf1, 0x3c, 0xa7, 0x29, 0x93,
0xcb, 0x45, 0x5f, 0xe8, 0x10, 0x74, 0x62, 0xde,
0xb8, 0x77, 0x80, 0xd1, 0x12, 0x26, 0xac, 0x6d,
0xe9, 0xcf, 0xf3, 0x54, 0x3a, 0x0b, 0x95, 0x4e,
0xb1, 0x30, 0xa4, 0x96, 0xf8, 0x57, 0x49, 0x8e,
0x05, 0x1f, 0x62, 0x7c, 0xc3, 0x2b, 0xda, 0xed,
0xbb, 0x86, 0x0d, 0x7a, 0x97, 0x13, 0x6c, 0x4e,
0x51, 0x30, 0xe5, 0xf2, 0x2f, 0xd8, 0xc4, 0xa9,
0x91, 0x76, 0xf0, 0x17, 0x43, 0x38, 0x29, 0x84,
0xa2, 0xdb, 0xef, 0x65, 0x5e, 0xca, 0x0d, 0xbc,
0xe7, 0xfa, 0xd8, 0x81, 0x6f, 0x00, 0x14, 0x42,
0x25, 0x7c, 0x5d, 0xc9, 0x9e, 0xb6, 0x33, 0xab,
0x5a, 0x6f, 0x9b, 0xd9, 0xfe, 0x71, 0x44, 0xc5,
0x37, 0xa2, 0x88, 0x2d, 0x00, 0xb6, 0x13, 0xec,
0x4e, 0x96, 0xa8, 0x5a, 0xb5, 0xd7, 0xc3, 0x8d,
```

```
0x3f, 0xf2, 0xec, 0x04, 0x60, 0x71, 0x1b, 0x29,
0x04, 0x79, 0xe3, 0xc7, 0x1b, 0x66, 0x81, 0x4a,
0x25, 0x9d, 0xdc, 0x5f, 0x3e, 0xb0, 0xf8, 0xa2,
0x91, 0x34, 0xf6, 0x5c, 0x67, 0x89, 0x73, 0x05,
0x22, 0xaa, 0xcb, 0xee, 0xbf, 0x18, 0xd0, 0x4d,
0xf5, 0x36, 0xae, 0x01, 0x2f, 0x94, 0xc3, 0x49,
0x8b, 0xbd, 0x58, 0x12, 0xe0, 0x77, 0x6c, 0xda };

//////////////////Beginning of tbox//////////////////////
unsigned char tbox(unsigned char z){
        int k_index,i;
        unsigned char result,result1,result2;
        //printf(" %d size %d",cmeakey[1],sizeof(cmeakey));
        k_index = 0;
        result = z;

        for (i = 0; i < 4; i++)
        {
            // result =result^cmeakey[k_index];
            // result =result + cmeakey[k_index+1];
            result1 =result^cmeakey[k_index];
            result2 =result + cmeakey[k_index+1];
               if((result2<cmeakey[k_index+1])||(result2<result1))
                        result2^=0x01;
               result = z + CaveTable[result2];
            // result = z + CaveTable[result];
            k_index += 2;
        }
        return(result);
}

//////////////////////End of tbox//////////////////////
void CMEA(unsigned char *msg_buf)
{
        const int octet_count=no_of_octets;
        int msg_index,half;
        unsigned char k,z;
        /* first manipulation (inverse of third) */
        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
            //  k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
                msg_buf[msg_index] += k;
                z += msg_buf[msg_index];
        }
```

```
/* second manipulation (self-inverse) */

        half = octet_count/2;
        for (msg_index = 0; msg_index < half; msg_index++)
        {
                msg_buf[msg_index] ^=
                //msg_buf[octet_count - 1 - msg_index] | 0x00;
                msg_buf[octet_count - 1 - msg_index] | 0x01;
        }

/* third manipulation (inverse of first) */

        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
             k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
             //k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
             z += msg_buf[msg_index];
             msg_buf[msg_index] -= k;
        }
}
//////////////////End of CMEA/////////////////////
void CMEA2(unsigned char *msg_buf)
{
        const int octet_count=no_of_octets;
        int msg_index,half;
        unsigned char k,z;
/* first manipulation (inverse of third) */
        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
                //k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
                msg_buf[msg_index] += k;
                z += msg_buf[msg_index];
        }
/* second manipulation (self-inverse) */

        half = octet_count/2;
        for (msg_index = 0; msg_index < half; msg_index++)
        {
                msg_buf[msg_index] ^=
                msg_buf[octet_count - 1 - msg_index] | 0x00;
                //msg_buf[octet_count - 1 - msg_index] | 0x01;
        }
```

```
/* third manipulation (inverse of first) */

        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
             k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
            // k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                z += msg_buf[msg_index];
                msg_buf[msg_index] -= k;
        }
}
//////////////////End of CMEA2//////////////////////

unsigned long int checker(unsigned char *msg,unsigned char *plain){
unsigned long int  countrite=0;
int i;
unsigned int count1,count2,count3;
  for(count1=0;count1<=2;count1++){
        for(count2=0;count2<=255;count2++){
                for(count3=0;count3<=255;count3++){
                        msg[0]=plain[0]=(unsigned char)count3;
                        msg[1]=plain[1]=(unsigned char)count2;
                        msg[2]=plain[2]=(unsigned char)count1;
                        CMEA(msg);
                        //lastbits(msg,plain);
                        CMEA(msg);
                        for(i=0;i<3;i++)
                                if(plain[i]==msg[i])
                                        countrite++;
                               // printf("\n\n");

                }
         }
    }
          return count1*count2*count3-countrite/3;
}
//////////////////////EO Check data////////////////////////////////


void assignkey(int swch)
{
        int i;
        if(swch==0)
        {       cmeakey[0]=CMEAK0;
                cmeakey[1]=CMEAK1;
                cmeakey[2]=CMEAK2;
```

```
                cmeakey[3]=CMEAK3;
                cmeakey[4]=CMEAK4;
                cmeakey[5]=CMEAK5;
                cmeakey[6]=CMEAK6;
                cmeakey[7]=CMEAK7;
        }
        else if(swch==1)
        {       //printf("\t\tKeychanged!!\t\t ");
                for(i=0;i<8;i++)
                {       cmeakey[i]=rand()%256;

                }
        }
        else if(swch==2)
        {     for(i=0;i<7;i+=2)
                {       //printf("\n%dth cmeakey previous =%0x",i,cmeakey[i]);
                        cmeakey[i]=cmeakey[i]^0x80;
                        cmeakey[i+1]=cmeakey[i+1]^0x80;
                        //printf("\t\t%dth cmeakey changed =%0x",i,cmeakey[i]);
                }
        }

}
////////////////////EO assignkeys//////////////////////////
int switchmsbkey(unsigned char *msg,unsigned char *plain)
{     unsigned char msg2[no_of_octets];
        long int i,mismatch;
        for(i=0;i<no_of_octets;i++)
             msg2[i]=msg[i];
        assignkey(1);
        CMEA(msg);
        assignkey(2);
        CMEA(msg2);
        mismatch=0;
        for(i=0;i<no_of_octets;i++)
                if(msg[i]!=msg2[i])
                        mismatch++;
        return mismatch;
}

////////////////////EO switchmsbkey/////////////////////


int main()
{
  long unsigned int i,j;
```

```
 unsigned int count1,count2,count3;
 unsigned char msg[no_of_octets];
 unsigned char plain[no_of_octets];
 long unsigned int countchanges;

 if(!(checker(msg,plain)))
 {       countchanges=0;
         printf("\n\nSuccess!! \n");

      for(i=0;i<50000;i++)
      {  for(j=0;j<3;j++)
                         msg[j]=plain[j]=rand()%256;
                 countchanges+=switchmsbkey(msg,plain);
         }
         printf("\n\nTotal number of key changes = %ld",i);
         printf("\n\n No of changes on switching MSB's = %ld",countchanges);
         printf("\n\n No of same Ctexts on switching MSB's = %ld\n\n",i*3-
countchanges);
         printf("\n\n Average output change per key msb switching = %f
",(float)countchanges/(i*3));
 }
 else
         printf("\n\nDecrypted value not matching with plain text!!\n");

}
//////////////////////EO main()////////////////////////////////////////////
```

PROGRAM 5

```
#include<stdio.h>
#define PLAIN1 0x02
#define PLAIN2 0xd2
#define PLAIN3 0xf3
#define CMEAK0 0xa1
#define CMEAK1 0x74
#define CMEAK2 0x26
#define CMEAK3 0xdd
#define CMEAK4 0x02
#define CMEAK5 0x75
#define CMEAK6 0x69
#define CMEAK7 0x14
#define no_of_octets 3
unsigned char cmeakey[8];
void lastbits(unsigned char *,unsigned char *);
```

```
void assignkey(int);
unsigned char CaveTable[256] =
{0xd9, 0x23, 0x5f, 0xe6, 0xca, 0x68, 0x97, 0xb0,
0x7b, 0xf2, 0x0c, 0x34, 0x11, 0xa5, 0x8d, 0x4e,
0x0a, 0x46, 0x77, 0x8d, 0x10, 0x9f, 0x5e, 0x62,
0xf1, 0x34, 0xec, 0xa5, 0xc9, 0xb3, 0xd8, 0x2b,
0x59, 0x47, 0xe3, 0xd2, 0xff, 0xae, 0x64, 0xca,
0x15, 0x8b, 0x7d, 0x38, 0x21, 0xbc, 0x96, 0x00,
0x49, 0x56, 0x23, 0x15, 0x97, 0xe4, 0xcb, 0x6f,
0xf2, 0x70, 0x3c, 0x88, 0xba, 0xd1, 0x0d, 0xae,
0xe2, 0x38, 0xba, 0x44, 0x9f, 0x83, 0x5d, 0x1c,
0xde, 0xab, 0xc7, 0x65, 0xf1, 0x76, 0x09, 0x20,
0x86, 0xbd, 0x0a, 0xf1, 0x3c, 0xa7, 0x29, 0x93,
0xcb, 0x45, 0x5f, 0xe8, 0x10, 0x74, 0x62, 0xde,
0xb8, 0x77, 0x80, 0xd1, 0x12, 0x26, 0xac, 0x6d,
0xe9, 0xcf, 0xf3, 0x54, 0x3a, 0x0b, 0x95, 0x4e,
0xb1, 0x30, 0xa4, 0x96, 0xf8, 0x57, 0x49, 0x8e,
0x05, 0x1f, 0x62, 0x7c, 0xc3, 0x2b, 0xda, 0xed,
0xbb, 0x86, 0x0d, 0x7a, 0x97, 0x13, 0x6c, 0x4e,
0x51, 0x30, 0xe5, 0xf2, 0x2f, 0xd8, 0xc4, 0xa9,
0x91, 0x76, 0xf0, 0x17, 0x43, 0x38, 0x29, 0x84,
0xa2, 0xdb, 0xef, 0x65, 0x5e, 0xca, 0x0d, 0xbc,
0xe7, 0xfa, 0xd8, 0x81, 0x6f, 0x00, 0x14, 0x42,
0x25, 0x7c, 0x5d, 0xc9, 0x9e, 0xb6, 0x33, 0xab,
0x5a, 0x6f, 0x9b, 0xd9, 0xfe, 0x71, 0x44, 0xc5,
0x37, 0xa2, 0x88, 0x2d, 0x00, 0xb6, 0x13, 0xec,
0x4e, 0x96, 0xa8, 0x5a, 0xb5, 0xd7, 0xc3, 0x8d,
0x3f, 0xf2, 0xec, 0x04, 0x60, 0x71, 0x1b, 0x29,
0x04, 0x79, 0xe3, 0xc7, 0x1b, 0x66, 0x81, 0x4a,
0x25, 0x9d, 0xdc, 0x5f, 0x3e, 0xb0, 0xf8, 0xa2,
0x91, 0x34, 0xf6, 0x5c, 0x67, 0x89, 0x73, 0x05,
0x22, 0xaa, 0xcb, 0xee, 0xbf, 0x18, 0xd0, 0x4d,
0xf5, 0x36, 0xae, 0x01, 0x2f, 0x94, 0xc3, 0x49,
0x8b, 0xbd, 0x58, 0x12, 0xe0, 0x77, 0x6c, 0xda };

/*unsigned char CaveTable2[256]=
{217, 35, 95,230,202,104,151,176,123,242, 12, 52, 17,165,141,
  78, 10, 70,119,141, 16,159, 94, 98,241, 52,236,165,201,179,
 216, 43, 89, 71,227,210,255,174,100,202, 21,139,125, 56, 33,
 188,150,  0, 73, 86, 35, 21,151,228,203,111,242,112, 60,136,
 186,209, 13,174,226, 56,186, 68,159,131, 93, 28,222,171,199,
 101,241,118,  9, 32,134,189, 10,241, 60,167, 41,147,203, 69,
  95,232, 16,116, 98,222,184,119,128,209, 18, 38,172,109,233,
 207,243, 84, 58, 11,149, 78,177, 48,164,150,248, 87, 73,142,
   5, 31, 98,124,195, 43,218,237,187,134, 13,122,151, 19,108,
  78, 81, 48,229,242, 47,216,196,169,145,118,240, 23, 67, 56,
```

```
 41,132,162,219,239,101, 94,202, 13,188,231,250,216,129,111,
  0, 20, 66, 37,124, 93,201,158,182, 51,171, 90,111,155,217,
254,113, 68,197, 55,162,136, 45,  0,182, 19,236, 78,150,168,
 90,181,215,195,141, 63,242,236,  4, 96,113, 27, 41,  4,121,
227,199, 27,102,129, 74, 37,157,220, 95, 62,176,248,162,145,
 52,246, 92,103,137,115,  5, 34,170,203,238,191, 24,208, 77,
 245, 54,174,  1, 47,148,195, 73,139,189, 88, 18,224,119,108,
218 };*/

//////////////////Beginning of tbox//////////////////////
unsigned char tbox(unsigned char z){
        int k_index,i;
        unsigned char result,result1,result2;
        //printf(" %d size %d",cmeakey[1],sizeof(cmeakey));
        k_index = 0;
        result = z;

        for (i = 0; i < 4; i++)
        {
             result =result^cmeakey[k_index];
             result =result + cmeakey[k_index+1];
            /* result1 =result^cmeakey[k_index];
             result2 =result + cmeakey[k_index+1];
                 if((result2<cmeakey[k_index+1])||(result2<result1))
                          result2^=0x01;
                 result = z + CaveTable[result2];*/
                 result = z + CaveTable[result];
                 k_index += 2;
        }
        return(result);
}

//////////////////////End of tbox//////////////////////
void CMEA(unsigned char *msg_buf)
{
        const int octet_count=no_of_octets;
        int msg_index,half;
        unsigned char k,z;
        /* first manipulation (inverse of third) */
        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
            //   k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                 k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
                 msg_buf[msg_index] += k;
                 z += msg_buf[msg_index];
```

```
        }
/* second manipulation (self-inverse) */

        half = octet_count/2;
        for (msg_index = 0; msg_index < half; msg_index++)
        {
                msg_buf[msg_index] ^=
                msg_buf[octet_count - 1 - msg_index] | 0x00;
                //msg_buf[octet_count - 1 - msg_index] | 0x01;
        }

/* third manipulation (inverse of first) */

        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
             k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
             //k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
             z += msg_buf[msg_index];
             msg_buf[msg_index] -= k;
        }
}
//////////////////End of CMEA//////////////////////
void CMEA2(unsigned char *msg_buf)
{
        const int octet_count=no_of_octets;
        int msg_index,half;
        unsigned char k,z;
        /* first manipulation (inverse of third) */
        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
                //k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
                msg_buf[msg_index] += k;
                z += msg_buf[msg_index];
        }
/* second manipulation (self-inverse) */

        half = octet_count/2;
        for (msg_index = 0; msg_index < half; msg_index++)
        {
                msg_buf[msg_index] ^=
                msg_buf[octet_count - 1 - msg_index] | 0x00;
                //msg_buf[octet_count - 1 - msg_index] | 0x01;
        }
```

```
/* third manipulation (inverse of first) */

        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
             k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
            // k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                z += msg_buf[msg_index];
                msg_buf[msg_index] -= k;
        }
}
//////////////////End of CMEA2/////////////////////

unsigned long int checker(unsigned char *msg,unsigned char *plain){
unsigned long int  countrite=0;
int i;
unsigned int count1,count2,count3;
  for(count1=0;count1<=255;count1++){
        for(count2=0;count2<=255;count2++){
                for(count3=0;count3<=255;count3++){
                        msg[0]=plain[0]=(unsigned char)count3;
                        msg[1]=plain[1]=(unsigned char)count2;
                        msg[2]=plain[2]=(unsigned char)count1;
                        CMEA(msg);
                        lastbits(msg,plain);
                        CMEA(msg);
                        for(i=0;i<3;i++)
                                if(plain[i]==msg[i])
                                        countrite++;
                                // printf("\n\n");

                }
         }
    }
          return count1*count2*count3-countrite/3;
}
//////////////////////EO Check data////////////////////////////////
 void lastbits(unsigned char *msg,unsigned char *plain)
 {     int i=0;
        //for(i=0;i<no_of_octets;i++)
        {
        printf("\nLSB of  Ptext %d=%d\t ",i,(plain[i]>>1)%2);
        //printf("\nPlaintext %d = %0x",i,plain[i]);
        printf("LSB of Ctext %d = %d",i,(msg[i]>>1)%2);
        //printf("\tCiphertext %d = %0x",i,msg[i]);
```

```
        }
        if((plain[0]>>1)%2==(msg[0]>>1)%2)
                printf("\n\nLSBs of texts not matching!!!\n\n");
  }
//////////////////////EO last bits//////////////////////////

void checkT0()
{       int i,j;
        unsigned char msg[3];
        unsigned char plain[3];
        unsigned char t0;
        long int countcodebreaks=0;
        int key_changes;
        for(key_changes=1;key_changes<=500;key_changes++)
        {       assignkey(1);
                for(j=0;j<1;j++)
                {       t0=tbox(j);
                        //printf(" T%d = %d",j,t0);
                        for(i=0;i<no_of_octets;i++)
                                msg[i]=plain[i]=(1-t0)&0xff;

                        CMEA(msg);
                        if(msg[0]==(unsigned char)(-t0))
                        {       countcodebreaks++;
                                printf("\nCode broken !! C[1-t0] = msg[0] = %d for j=%d
!!\n\n",msg[0],j);
                        }
                }
        }

        printf("\n\nAmbiguity  = %d",countcodebreaks);
        assignkey(0);
}
//////////////////////EO checkT0////////////////////////////

void assignkey(int swch)
{
        int i;
        if(swch==0)
        {       cmeakey[0]=CMEAK0;
                cmeakey[1]=CMEAK1;
                cmeakey[2]=CMEAK2;
                cmeakey[3]=CMEAK3;
                cmeakey[4]=CMEAK4;
                cmeakey[5]=CMEAK5;
                cmeakey[6]=CMEAK6;
```

```
                cmeakey[7]=CMEAK7;
        }
        else if(swch==1)
        {       printf("\t\tKeychanged!!\t\t ");
                for(i=0;i<8;i++)
                {       cmeakey[i]=rand()%256;

                }
        }
        else if(swch==2)
        {     for(i=0;i<2;i++)
                {       //printf("\n%dth cmeakey previous =%0x",i,cmeakey[i]);
                        cmeakey[i]=cmeakey[i]^0x80;
                        //printf("\t\t%dth cmeakey changed =%0x",i,cmeakey[i]);
                }
        }

}
////////////////////EO assignkeys//////////////////////////
int switchmsbkey(unsigned char *msg,unsigned char *plain)
{     unsigned char msg2[no_of_octets];
        long int i,mismatch;
        for(i=0;i<no_of_octets;i++)
             msg2[i]=msg[i];
        assignkey(1);
        CMEA(msg);
        assignkey(2);
        CMEA(msg2);
        mismatch=0;
        for(i=0;i<no_of_octets;i++)
                if(msg[i]!=msg2[i])
                        mismatch++;
        if(mismatch>2)
        {       //printf("\n\nValue changed by changing MSBs!!");
                return 1;
        }
        else
        {       //printf("\n\nNo value change...try again!! \n");
                return 0;
        }
}

////////////////////EO switchmsbkey//////////////////////
void Keyunchanged(unsigned char *msg)
{     int i,j;
        unsigned long int suma,sumb;
```

```
        unsigned char msg2[3],plain[3];
        int temp,temp2;
        unsigned long int count1,count2,count3,counta,countb;
        printf("\n\nKEY UNCHANGED\n\n");
        printf("\tPlaintext\tMine\tCMEA\n");
        suma=sumb=0;
        for(count1=0;count1<=255;count1++)
                for(count2=0;count2<=255;count2++)
                        for(count3=0;count3<=255;count3++){
                                msg[0]=plain[0]=msg2[0]=(unsigned char)count3;
                                msg[1]=plain[1]=msg2[1]=(unsigned char)count2;
                                msg[2]=plain[2]=msg2[2]=(unsigned char)count1;
                                CMEA(msg);
                                CMEA2(msg2);
                            //  printf("\n");
                                for(i=0;i<no_of_octets;i++)
                                {     //  printf("%d octet :
%0x\t%0x\t%0x",i,plain[i],msg[i],msg2[i]);
                                        temp = plain[i]^msg[i];
                                        temp2 = plain[i]^msg2[i];
                                        counta=countb=0;
                                        for(j=0;j<8;j++)
                                        {       if(temp&0x01)
                                                        counta++;
                                                if(temp2&0x01)
                                                        countb++;
                                                temp=temp>>1;
                                                temp2=temp2>>1;
                                        }
                                        suma+=counta;
                                        sumb+=countb;
                                        //printf("\t%ld\t%ld\n",counta,countb);
                                }

                        }
                printf("\n\nTotal and average bit changes in CAVE = %ld,
%lf",suma,(float)suma/(count1*count2*count3));
                printf("\nTotal and average bit changes in NEW =  %ld,
%lf",sumb,(float)sumb/(count1*count2*count3));
                printf("\nPercentage fall = %lf",(float)(suma-sumb)*100/suma);
}
//////////////////////EO Keyunchanged//////////////////////
void Plainunchanged(unsigned char *msg)
{       unsigned long int i0,i1,i2,i3,i4,i5,i6,i7;
        int i,j,temp,temp2;
        long unsigned int suma,sumb,counta,countb;
```

```
        unsigned char msg2[3],plain[3];
        for(i=0;i<3;i++)
                plain[i]=msg2[i]=msg[i];
        for(i0=0;i0<=255;i0++)
        for(i1=0;i1<=255;i1++)
        for(i2=0;i2<=255;i2++)
        for(i3=0;i3<=255 ;i3++)
        for(i4=0;i4<=255;i4++)
        for(i5=0;i5<=255;i5++)
        for(i6=0;i6<=255;i6++)
        for(i7=0;i7<=255;i7++)
        {       cmeakey[0]=i0;
                cmeakey[1]=i1;
                cmeakey[2]=i2;
                cmeakey[3]=i3;
                cmeakey[4]=i4;
                cmeakey[5]=i5;
                cmeakey[6]=i6;
                cmeakey[7]=i7;
                CMEA(msg);
                CMEA2(msg2);
            //  printf("\n");
                for(i=0;i<no_of_octets;i++)
                {       //printf("%d octet : %0x\t%0x\t%0x",i,plain[i],msg[i],msg2[i]);
                        temp = plain[i]^msg[i];
                        temp2 = plain[i]^msg2[i];
                        counta=countb=0;
                        for(j=0;j<8;j++)
                        {       if(temp&0x01)
                                counta++;
                                if(temp2&0x01)
                                countb++;
                                temp=temp>>1;
                                temp2=temp2>>1;
                        }
                        suma+=counta;
                        sumb+=countb;
                        //printf("\t%ld\t%ld\n",counta,countb);
                }
        }
        printf("\n\nTotal and average bit changes in CAVE = %ld,
%lf",suma,(float)suma/(i0*i1*i2*i3*i4*i5*i6*i7));
        printf("\nTotal and average bit changes in NEW =  %ld,
%lf",sumb,(float)sumb/(i0*i1*i2*i3*i4*i5*i6*i7));
        if(sumb>suma)
                printf("\nPercentage rise = %lf%%",((float)(sumb-suma)*100/suma));
```

```
	else
		printf("\nPercentage fall = %lf%%",((float)(suma-sumb)*100/suma));
}
////////////////////////////////EO keyunchanged//////////////////////////

void Nonlinearcheck()
{	const long int no_of_terms=24;
	unsigned char msg1[3],msg2[3],temp,cryptedxor[3];
	int i,j,delta_p,delta_c,distribution[no_of_terms][no_of_terms];
	unsigned long int
i1,j1,count1,count2,count3,countcipher,countplain;//countdetection=0;
	for(j1=0;j1<no_of_terms;j1++)
		for(i1=0;i1<no_of_terms;i1++)
			distribution[i1][j1]=0;
	msg1[0]=0xb1;msg1[1]=0x1c;msg1[2]=0x15;msg2[0]=0x00;msg2[1]=0x0;msg2[
2]=0x0;
	for(count1=0;count1<6;count1++)
		for(count2=0;count2<6;count2++)
			for(count3=0;count3<256;count3++)
			{	msg1[0]=count1;msg1[1]=count2;msg1[2]=count3;
				countplain=countcipher=0;
				for(i=0;i<no_of_octets;i++)
				{	temp=msg1[i]^msg2[i];
					//printf("%d octet :
%0x\t%0x\t%0x",i,plain[i],msg[i],msg2[i]);

					for(j=0;j<8;j++)
					{	if(temp&0x01)
						countplain++;
						temp=temp>>1;
					}
				}
				CMEA2(msg1);
				CMEA2(msg2);
				//CMEA2(msg1xormsg2);
				//printf("\t");
				for(i=0;i<no_of_octets;i++)
				{	temp=msg1[i]^msg2[i];
					for(j=0;j<8;j++)
					{	if(temp&0x01)
							countcipher++;
						temp=temp>>1;
					}
				}
				distribution[countplain][countcipher]++;
				//printf("\n");
```

```
                        }
        printf("\n  ");
        for(i=0;i<no_of_terms;i++)
                if(i<=10)
                        printf(" %d  ",i);
                else printf("%d  ",i);
        for(i1=0;i1<no_of_terms;i1++)
        {       printf("\n%d",i1);
                for(j1=0;j1<no_of_terms;j1++)
                        printf("  %d ",distribution[i1][j1]);

        }
                //printf("\n\n%d",countdetection);
}

//////////////////////////////EO Nonlinearcheck/////////////
void Nonlinearcheck2()
{       unsigned char msg1[3],msg2[3],msg1xormsg2[3],cryptedxor[3],temp[3];
        int i,j;
        unsigned long int count1,count2,count3,countnonlinear=0;

        msg1[0]=0xb1;msg1[1]=0x1c;msg1[2]=0x15;msg2[0]=0xa0;msg2[1]=0xab;msg2
[2]=0xca;
        for(count1=0;count1<256;count1++)
                for(count2=0;count2<256;count2++)
                        for(count3=0;count3<256;count3++)
                        {       msg1[0]=count1;msg1[1]=count2;msg1[2]=count3;
        //for(i=0;i<3;i++)
        //      printf("\n%0x\n\n",msg1xormsg2[i]=msg1[i]^msg2[i]);
                                for(i=0;i<3;i++)
                                        msg1xormsg2[i]=msg1[i]^msg2[i];
                                CMEA(msg1);
                                CMEA(msg2);
                                CMEA(msg1xormsg2);
                                //printf("\t");
                                for(i=0;i<3;i++)

                                {       cryptedxor[i]=msg1[i]^msg2[i];
                //printf("\nMsg1= %0x \tMsg2= %0x\tXORed msges=%0x\t
CryptedXORs=%0x",msg1[i],msg2[i],cryptedxor[i],msg1xormsg2[i]);
                                        if(!(cryptedxor[i]^msg1xormsg2[i]))
                                        {       //printf(" Linearity Alert %d!!!! ",i);
                                                countnonlinear++;
                                        }
                                }
        //printf("\n");
```

```
                        }
        printf("\n\nNo of non linearities = %ld out of
%ld",countnonlinear,count1*count2*count3);

}

//////////////////////////////EO Nonlinearcheck2//////////////////////////////
int main()
{
 long unsigned int i,j,count,countrite;
 unsigned int count1,count2,count3;
// char msg[3]={PLAIN1,PLAIN2,PLAIN3};
// char plain[3]={PLAIN1,PLAIN2,PLAIN3};
 unsigned char msg[no_of_octets]={0xb6,0x2d,0xa2};//,0x44,0xfe,0x9b};
 unsigned char plain[no_of_octets]={0xb6,0x2d,0xa2};//,0x44,0xfe,0x9b};
 unsigned char t0;
 int nochange,countchanges;
 //Nonlinearcheck();
// assignkey(0);
 //clrscr();
 //Keyunchanged(msg);
 //Plainunchanged(msg);
if(checker(msg,plain)==0)
        printf("\n\nSuccess!!!!!!!!\n");
  /*{   countchanges=nochange=0;
        printf("\n\nSuccess!! \n");
   }
         for(i=0;i<5000;i++)
         {      // for(j=0;j<3;j++)
                        //msg[j]=plain[j]=rand()%256;
                if(switchmsbkey(msg,plain))
                        countchanges++;
                else
                        nochange++;
         }
         printf("\n\nTotal number of key chabges = %d",i);
         printf("\n\n No of changes on switching MSB's = %d",countchanges);
         printf("\n\n No of same Ctexts on switching MSB's = %d\n\n",nochange);

    }
 else
        printf("\n\nDecrypted value not matching with plain text!!\n");
     */
// printf("Errors = %ld",checker(msg,plain));
 //checkT0();
/*
```

```
 for(i=0;i<3;i++){
  msg[i]=((1-t0)&0x00ff);
  plain[i]=((1-t0)&0x00ff);
  }
  printf("T0= %d, sizeof int = %d",t0,sizeof(int));
 */
}
```

PROGRAM 6

```
#include<stdio.h>
#define PLAIN1 0xA0
#define PLAIN2 0x7B
#define PLAIN3 0x1C
#define CMEAK0 0xA0
#define CMEAK1 0x7b
#define CMEAK2 0x1c
#define CMEAK3 0xd1
#define CMEAK4 0x02
#define CMEAK5 0x75
#define CMEAK6 0x69
#define CMEAK7 0x14
#define no_of_octets 6
unsigned char cmeakey[8];
void lastbits(unsigned char *,unsigned char *);
void assignkey(int);
unsigned char CaveTable[256] =
{0xd9, 0x23, 0x5f, 0xe6, 0xca, 0x68, 0x97, 0xb0,
0x7b, 0xf2, 0x0c, 0x34, 0x11, 0xa5, 0x8d, 0x4e,
0x0a, 0x46, 0x77, 0x8d, 0x10, 0x9f, 0x5e, 0x62,
0xf1, 0x34, 0xec, 0xa5, 0xc9, 0xb3, 0xd8, 0x2b,
0x59, 0x47, 0xe3, 0xd2, 0xff, 0xae, 0x64, 0xca,
0x15, 0x8b, 0x7d, 0x38, 0x21, 0xbc, 0x96, 0x00,
0x49, 0x56, 0x23, 0x15, 0x97, 0xe4, 0xcb, 0x6f,
0xf2, 0x70, 0x3c, 0x88, 0xba, 0xd1, 0x0d, 0xae,
0xe2, 0x38, 0xba, 0x44, 0x9f, 0x83, 0x5d, 0x1c,
0xde, 0xab, 0xc7, 0x65, 0xf1, 0x76, 0x09, 0x20,
0x86, 0xbd, 0x0a, 0xf1, 0x3c, 0xa7, 0x29, 0x93,
0xcb, 0x45, 0x5f, 0xe8, 0x10, 0x74, 0x62, 0xde,
0xb8, 0x77, 0x80, 0xd1, 0x12, 0x26, 0xac, 0x6d,
0xe9, 0xcf, 0xf3, 0x54, 0x3a, 0x0b, 0x95, 0x4e,
0xb1, 0x30, 0xa4, 0x96, 0xf8, 0x57, 0x49, 0x8e,
0x05, 0x1f, 0x62, 0x7c, 0xc3, 0x2b, 0xda, 0xed,
0xbb, 0x86, 0x0d, 0x7a, 0x97, 0x13, 0x6c, 0x4e,
0x51, 0x30, 0xe5, 0xf2, 0x2f, 0xd8, 0xc4, 0xa9,
```

```
0x91, 0x76, 0xf0, 0x17, 0x43, 0x38, 0x29, 0x84,
0xa2, 0xdb, 0xef, 0x65, 0x5e, 0xca, 0x0d, 0xbc,
0xe7, 0xfa, 0xd8, 0x81, 0x6f, 0x00, 0x14, 0x42,
0x25, 0x7c, 0x5d, 0xc9, 0x9e, 0xb6, 0x33, 0xab,
0x5a, 0x6f, 0x9b, 0xd9, 0xfe, 0x71, 0x44, 0xc5,
0x37, 0xa2, 0x88, 0x2d, 0x00, 0xb6, 0x13, 0xec,
0x4e, 0x96, 0xa8, 0x5a, 0xb5, 0xd7, 0xc3, 0x8d,
0x3f, 0xf2, 0xec, 0x04, 0x60, 0x71, 0x1b, 0x29,
0x04, 0x79, 0xe3, 0xc7, 0x1b, 0x66, 0x81, 0x4a,
0x25, 0x9d, 0xdc, 0x5f, 0x3e, 0xb0, 0xf8, 0xa2,
0x91, 0x34, 0xf6, 0x5c, 0x67, 0x89, 0x73, 0x05,
0x22, 0xaa, 0xcb, 0xee, 0xbf, 0x18, 0xd0, 0x4d,
0xf5, 0x36, 0xae, 0x01, 0x2f, 0x94, 0xc3, 0x49,
0x8b, 0xbd, 0x58, 0x12, 0xe0, 0x77, 0x6c, 0xda };

/*unsigned char CaveTable2[256]=
{217, 35, 95,230,202,104,151,176,123,242, 12, 52, 17,165,141,
  78, 10, 70,119,141, 16,159, 94, 98,241, 52,236,165,201,179,
 216, 43, 89, 71,227,210,255,174,100,202, 21,139,125, 56, 33,
 188,150,  0, 73, 86, 35, 21,151,228,203,111,242,112, 60,136,
 186,209, 13,174,226, 56,186, 68,159,131, 93, 28,222,171,199,
 101,241,118,  9, 32,134,189, 10,241, 60,167, 41,147,203, 69,
  95,232, 16,116, 98,222,184,119,128,209, 18, 38,172,109,233,
 207,243, 84, 58, 11,149, 78,177, 48,164,150,248, 87, 73,142,
   5, 31, 98,124,195, 43,218,237,187,134, 13,122,151, 19,108,
  78, 81, 48,229,242, 47,216,196,169,145,118,240, 23, 67, 56,
  41,132,162,219,239,101, 94,202, 13,188,231,250,216,129,111,
   0, 20, 66, 37,124, 93,201,158,182, 51,171, 90,111,155,217,
 254,113, 68,197, 55,162,136, 45,  0,182, 19,236, 78,150,168,
  90,181,215,195,141, 63,242,236,  4, 96,113, 27, 41,  4,121,
 227,199, 27,102,129, 74, 37,157,220, 95, 62,176,248,162,145,
  52,246, 92,103,137,115,  5, 34,170,203,238,191, 24,208, 77,
 245, 54,174,  1, 47,148,195, 73,139,189, 88, 18,224,119,108,
218 };*/

//////////////////Beginning of tbox//////////////////////
unsigned char tbox(unsigned char z){
        int k_index,i;
        unsigned char result,result1,result2;
        //printf(" %d size %d",cmeakey[1],sizeof(cmeakey));
        k_index = 0;
        result = z;

        for (i = 0; i < 4; i++)
        {
             result =result^cmeakey[k_index];
```

```
                result =result + cmeakey[k_index+1];
               /* result1 =result^cmeakey[k_index];
                result2 =result + cmeakey[k_index+1];
                    if((result2<cmeakey[k_index+1])||(result2<result1))
                            result2^=0x01;
                    result = z + CaveTable[result2];*/
                    result = z + CaveTable[result];
                    k_index += 2;
            }
            return(result);
}

//////////////////////End of tbox//////////////////////
void CMEA(unsigned char *msg_buf)
{
            const int octet_count=no_of_octets;
            int msg_index,half;
            unsigned char k,z;
            /* first manipulation (inverse of third) */
            z = 0;
            for (msg_index = 0; msg_index < octet_count; msg_index++)
            {
                //  k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                    k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
                    msg_buf[msg_index] += k;
                    z += msg_buf[msg_index];
            }
/* second manipulation (self-inverse) */

            half = octet_count/2;
            for (msg_index = 0; msg_index < half; msg_index++)
            {
                    msg_buf[msg_index] ^=
                    //msg_buf[octet_count - 1 - msg_index] | 0x00;
                    msg_buf[octet_count - 1 - msg_index] | 0x01;
            }

/* third manipulation (inverse of first) */

            z = 0;
            for (msg_index = 0; msg_index < octet_count; msg_index++)
            {
                k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
                //k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                z += msg_buf[msg_index];
                msg_buf[msg_index] -= k;
```

```
        }
}
//////////////////End of CMEA/////////////////////
void CMEA2(unsigned char *msg_buf)
{
        const int octet_count=no_of_octets;
        int msg_index,half;
        unsigned char k,z;
        /* first manipulation (inverse of third) */
        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
                //k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
                msg_buf[msg_index] += k;
                z += msg_buf[msg_index];
        }
/* second manipulation (self-inverse) */

        half = octet_count/2;
        for (msg_index = 0; msg_index < half; msg_index++)
        {
                msg_buf[msg_index] ^=
                msg_buf[octet_count - 1 - msg_index] | 0x00;
                //msg_buf[octet_count - 1 - msg_index] | 0x01;
        }

/* third manipulation (inverse of first) */

        z = 0;
        for (msg_index = 0; msg_index < octet_count; msg_index++)
        {
             k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
            // k = tbox((unsigned char)((z+1) ^ (msg_index & 0xff)));
                z += msg_buf[msg_index];
                msg_buf[msg_index] -= k;
        }
}
//////////////////End of CMEA2/////////////////////

unsigned long int checker(unsigned char *msg,unsigned char *plain){
unsigned long int  countrite=0;
int i;
unsigned int count1,count2,count3;
  for(count1=0;count1<=255;count1++){
        for(count2=0;count2<=255;count2++){
```

```
                for(count3=0;count3<=255;count3++){
                        msg[0]=plain[0]=(unsigned char)count3;
                        msg[1]=plain[1]=(unsigned char)count2;
                        msg[2]=plain[2]=(unsigned char)count1;
                        CMEA(msg);
                        lastbits(msg,plain);
                        CMEA(msg);
                        for(i=0;i<3;i++)
                                if(plain[i]==msg[i])
                                        countrite++;
                                // printf("\n\n");

                }
            }
        }
            return count1*count2*count3-countrite/3;
}
//////////////////////EO Check data////////////////////////////////
 void lastbits(unsigned char *msg,unsigned char *plain)
 {    int i=0;
        //for(i=0;i<no_of_octets;i++)
        {
        printf("\nLSB of  Ptext %d=%d\t ",i,(plain[i]>>1)%2);
        //printf("\nPlaintext %d = %0x",i,plain[i]);
        printf("LSB of Ctext %d = %d",i,(msg[i]>>1)%2);
        //printf("\tCiphertext %d = %0x",i,msg[i]);
        }
        if((plain[0]>>1)%2==(msg[0]>>1)%2)
                printf("\n\nLSBs of texts not matching!!!\n\n");
  }
//////////////////////EO last bits//////////////////////////

void checkT0()
{       int i,j;
        unsigned char msg[3];
        unsigned char plain[3];
        unsigned char t0;
        long int countcodebreaks=0;
        int key_changes;
        for(key_changes=1;key_changes<=500;key_changes++)
        {       assignkey(1);
                for(j=0;j<1;j++)
                {       t0=tbox(j);
                        //printf(" T%d = %d",j,t0);
                        for(i=0;i<no_of_octets;i++)
                                msg[i]=plain[i]=(1-t0)&0xff;
```

```
                        CMEA(msg);
                        if(msg[0]==(unsigned char)(-t0))
                        {       countcodebreaks++;
                                printf("\nCode broken !! C[1-t0] = msg[0] = %d for j=%d
!!\n\n",msg[0],j);
                        }
                }
        }

        printf("\n\nAmbiguity  = %d",countcodebreaks);
        assignkey(0);
}
//////////////////////EO checkT0////////////////////////////

void assignkey(int swch)
{
        int i;
        if(swch==0)
        {       cmeakey[0]=CMEAK0;
                cmeakey[1]=CMEAK1;
                cmeakey[2]=CMEAK2;
                cmeakey[3]=CMEAK3;
                cmeakey[4]=CMEAK4;
                cmeakey[5]=CMEAK5;
                cmeakey[6]=CMEAK6;
                cmeakey[7]=CMEAK7;
        }
        else if(swch==1)
        {       printf("\t\tKeychanged!!\t\t ");
                for(i=0;i<8;i++)
                {       cmeakey[i]=rand()%256;

                }
        }
        else if(swch==2)
        {     for(i=0;i<2;i++)
                {       //printf("\n%dth cmeakey previous =%0x",i,cmeakey[i]);
                        cmeakey[i]=cmeakey[i]^0x80;
                        //printf("\t\t%dth cmeakey changed =%0x",i,cmeakey[i]);
                }
        }

}
//////////////////////EO assignkeys////////////////////////////
int switchmsbkey(unsigned char *msg,unsigned char *plain)
```

```
{       unsigned char msg2[no_of_octets];
        long int i,mismatch;
        for(i=0;i<no_of_octets;i++)
             msg2[i]=msg[i];
        assignkey(1);
        CMEA(msg);
        assignkey(2);
        CMEA(msg2);
        mismatch=0;
        for(i=0;i<no_of_octets;i++)
                if(msg[i]!=msg2[i])
                        mismatch++;
        if(mismatch>2)
        {       //printf("\n\nValue changed by changing MSBs!!");
                return 1;
        }
        else
        {       //printf("\n\nNo value change...try again!! \n");
                return 0;
        }
}

//////////////////////EO switchmsbkey//////////////////////
void Keyunchanged(unsigned char *msg)
{       int i,j;
        unsigned long int suma,sumb;
        unsigned char msg2[3],plain[3];
        int temp,temp2;
        unsigned long int count1,count2,count3,counta,countb;
        printf("\n\nKEY UNCHANGED\n\n");
        printf("\tPlaintext\tMine\tCMEA\n");
        suma=sumb=0;
        for(count1=0;count1<=255;count1++)
                for(count2=0;count2<=255;count2++)
                        for(count3=0;count3<=255;count3++){
                                msg[0]=plain[0]=msg2[0]=(unsigned char)count3;
                                msg[1]=plain[1]=msg2[1]=(unsigned char)count2;
                                msg[2]=plain[2]=msg2[2]=(unsigned char)count1;
                                CMEA(msg);
                                CMEA2(msg2);
                            //  printf("\n");
                                for(i=0;i<no_of_octets;i++)
                                {       //  printf("%d octet :
%0x\t%0x\t%0x",i,plain[i],msg[i],msg2[i]);
                                        temp = plain[i]^msg[i];
                                        temp2 = plain[i]^msg2[i];
```

```
                                        counta=countb=0;
                                        for(j=0;j<8;j++)
                                        {       if(temp&0x01)
                                                        counta++;
                                                if(temp2&0x01)
                                                        countb++;
                                                temp=temp>>1;
                                                temp2=temp2>>1;
                                        }
                                        suma+=counta;
                                        sumb+=countb;
                                        //printf("\t%ld\t%ld\n",counta,countb);
                                }

                        }
                printf("\n\nTotal and average bit changes in CAVE = %ld,
%lf",suma,(float)suma/(count1*count2*count3));
                printf("\nTotal and average bit changes in NEW =  %ld,
%lf",sumb,(float)sumb/(count1*count2*count3));
                printf("\nPercentage fall = %lf",(float)(suma-sumb)*100/suma);
}
//////////////////////EO Keyunchanged/////////////////////
void Plainunchanged(unsigned char *msg)
{       unsigned long int i0,i1,i2,i3,i4,i5,i6,i7;
        int i,j,temp,temp2;
        long unsigned int suma,sumb,counta,countb;
        unsigned char msg2[3],plain[3];
        for(i=0;i<3;i++)
                plain[i]=msg2[i]=msg[i];
        for(i0=0;i0<=255;i0++)
        for(i1=0;i1<=255;i1++)
        for(i2=0;i2<=255;i2++)
        for(i3=0;i3<=255 ;i3++)
        for(i4=0;i4<=255;i4++)
        for(i5=0;i5<=255;i5++)
        for(i6=0;i6<=255;i6++)
        for(i7=0;i7<=255;i7++)
        {       cmeakey[0]=i0;
                cmeakey[1]=i1;
                cmeakey[2]=i2;
                cmeakey[3]=i3;
                cmeakey[4]=i4;
                cmeakey[5]=i5;
                cmeakey[6]=i6;
                cmeakey[7]=i7;
                CMEA(msg);
```

```
                CMEA2(msg2);
            //  printf("\n");
                for(i=0;i<no_of_octets;i++)
                {       //printf("%d octet : %0x\t%0x\t%0x",i,plain[i],msg[i],msg2[i]);
                        temp = plain[i]^msg[i];
                        temp2 = plain[i]^msg2[i];
                        counta=countb=0;
                        for(j=0;j<8;j++)
                        {       if(temp&0x01)
                                counta++;
                                if(temp2&0x01)
                                countb++;
                                temp=temp>>1;
                                temp2=temp2>>1;
                        }
                        suma+=counta;
                        sumb+=countb;
                        //printf("\t%ld\t%ld\n",counta,countb);
                }
        }
        printf("\n\nTotal and average bit changes in CAVE = %ld,
%lf",suma,(float)suma/(i0*i1*i2*i3*i4*i5*i6*i7));
        printf("\nTotal and average bit changes in NEW =  %ld,
%lf",sumb,(float)sumb/(i0*i1*i2*i3*i4*i5*i6*i7));
        if(sumb>suma)
                printf("\nPercentage rise = %lf%%",((float)(sumb-suma)*100/suma));
        else
                printf("\nPercentage fall = %lf%%",((float)(suma-sumb)*100/suma));
}
//////////////////////////////EO keyunchanged/////////////

void Nonlinearcheck()
{       const long int no_of_terms=24;
        unsigned char msg1[3],msg2[3],temp,cryptedxor[3];
        int i,j,delta_p,delta_c,distribution[no_of_terms][no_of_terms];
        unsigned long int
i1,j1,count1,count2,count3,countcipher,countplain;//countdetection=0;
        for(j1=0;j1<no_of_terms;j1++)
                for(i1=0;i1<no_of_terms;i1++)
                        distribution[i1][j1]=0;
        msg1[0]=0xb1;msg1[1]=0x1c;msg1[2]=0x15;msg2[0]=0x00;msg2[1]=0x0;msg2[
2]=0x0;
        for(count1=0;count1<6;count1++)
                for(count2=0;count2<6;count2++)
                        for(count3=0;count3<256;count3++)
                        {       msg1[0]=count1;msg1[1]=count2;msg1[2]=count3;
```

```
                                countplain=countcipher=0;
                                for(i=0;i<no_of_octets;i++)
                                {       temp=msg1[i]^msg2[i];
                                        //printf("%d octet :
%0x\t%0x\t%0x",i,plain[i],msg[i],msg2[i]);

                                        for(j=0;j<8;j++)
                                        {       if(temp&0x01)
                                                countplain++;
                                                temp=temp>>1;
                                        }
                                }
                                CMEA2(msg1);
                                CMEA2(msg2);
                                //CMEA2(msg1xormsg2);
                                //printf("\t");
                                for(i=0;i<no_of_octets;i++)
                                {       temp=msg1[i]^msg2[i];
                                        for(j=0;j<8;j++)
                                        {       if(temp&0x01)
                                                        countcipher++;
                                                temp=temp>>1;
                                        }
                                }
                                distribution[countplain][countcipher]++;
                                //printf("\n");
                        }
        printf("\n  ");
        for(i=0;i<no_of_terms;i++)
                if(i<=10)
                        printf(" %d  ",i);
                else printf("%d  ",i);
        for(i1=0;i1<no_of_terms;i1++)
        {       printf("\n%d",i1);
                for(j1=0;j1<no_of_terms;j1++)
                        printf("  %d ",distribution[i1][j1]);

        }
                //printf("\n\n%d",countdetection);
}

//////////////////////////////EO Nonlinearcheck//////////////
void Nonlinearcheck2()
{       unsigned char msg1[3],msg2[3],msg1xormsg2[3],cryptedxor[3],temp[3];
        int i,j;
        unsigned long int count1,count2,count3,countnonlinear=0;
```

```
	msg1[0]=0xb1;msg1[1]=0x1c;msg1[2]=0x15;msg2[0]=0xa0;msg2[1]=0xab;msg2
[2]=0xca;
	for(count1=0;count1<256;count1++)
		for(count2=0;count2<256;count2++)
			for(count3=0;count3<256;count3++)
			{	msg1[0]=count1;msg1[1]=count2;msg1[2]=count3;
	//for(i=0;i<3;i++)
	//	printf("\n%0x\n\n",msg1xormsg2[i]=msg1[i]^msg2[i]);
					for(i=0;i<3;i++)
						msg1xormsg2[i]=msg1[i]^msg2[i];
					CMEA(msg1);
					CMEA(msg2);
					CMEA(msg1xormsg2);
					//printf("\t");
					for(i=0;i<3;i++)

					{	cryptedxor[i]=msg1[i]^msg2[i];
		//printf("\nMsg1= %0x \tMsg2= %0x\tXORed msges=%0x\t
CryptedXORs=%0x",msg1[i],msg2[i],cryptedxor[i],msg1xormsg2[i]);
						if(!(cryptedxor[i]^msg1xormsg2[i]))
						{	//printf(" Linearity Alert %d!!!! ",i);
							countnonlinear++;
						}
					}
	//printf("\n");
			}
	printf("\n\nNo of non linearities = %ld out of
%ld",countnonlinear,count1*count2*count3);

}

////////////////////////////////EO Nonlinearcheck2//////////////
int main()
{
 long unsigned int i;
// char msg[3]={PLAIN1,PLAIN2,PLAIN3};
// char plain[3]={PLAIN1,PLAIN2,PLAIN3};
 unsigned char msg[no_of_octets]={0xb6,0x2d,0xa2,0x44,0xfe,0x9B};
 cmeakey[0]=CMEAK0;
 cmeakey[1]=CMEAK1;
 cmeakey[2]=CMEAK2;
 cmeakey[3]=CMEAK3;
 cmeakey[4]=CMEAK4;
 cmeakey[5]=CMEAK5;
 cmeakey[6]=CMEAK6;
```

```
cmeakey[7]=CMEAK7;
CMEA(msg);
printf("\n\n");
for(i=0;i<6;i++)
        printf("\t%0x",msg[i]);

}
```