A CRYPTANALYSIS OF THE TINY

ENCRYPTION ALGORITHM

by

VIKRAM REDDY ANDEM

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in the
Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2003

Submitted by Vikram Reddy Andem in partial fulfillment of the requirements for the degree of Master of Science specializing in Computer Science.

Accepted on behalf of the Faculty of the Graduate School by the thesis committee:

Kenneth G. Ricks, Ph.D.

Sibabrata Ray, Ph.D.

Randy K. Smith, Ph.D.

Phillip G. Bradford, Ph.D.
Chairperson

David W. Cordes, Ph.D.
Department Chairperson

Date

Ronald W. Rogers, Ph.D.
Dean of the Graduate School

Date

ACKNOWLEDGMENTS

It is difficult to overstate my gratitude to my advisor, Dr. Phillip Bradford. With his enthusiasm, his inspiration, and his great efforts to explain things clearly and simply, he helped to make cryptography fun for me. Throughout my thesis writing period, he provided encouragement, sound advice, good teaching, good company, and lots of good ideas. I would have been lost without him.

I am very fortunate to have Dr. Randy Smith and Dr. Sibabrata Ray on my thesis committee. Their guidance and feedback were crucial to my work. I want to thank Dr. Kenneth Ricks for serving on my thesis committee, and for providing valuable feedback.

Last and most importantly, I wish to thank my parents, Venkat Reddy Andem and Vasantha Reddy Andem. They bore me, raised me, supported me, taught me, and loved me.

CONTENTS

LIST OF FIGURES

ABSTRACT

The Tiny Encryption Algorithm (TEA) is a cryptographic algorithm designed to minimize memory footprint and maximize speed. It is a Feistel type cipher that uses operations from mixed (orthogonal) algebraic groups. This research presents the cryptanalysis of the Tiny Encryption Algorithm. In this research we inspected the most common methods in the cryptanalysis of a block cipher algorithm. TEA seems to be highly resistant to differential cryptanalysis, and achieves complete diffusion (where a one bit difference in the plaintext will cause approximately 32 bit differences in the cipher text) after only six rounds. Time performance on a modern desktop computer or workstation is very impressive.

CHAPTER 1

INTRODUCTION

1.1 Motivation

As computer systems become more pervasive and complex, security is increasingly important. Cryptographic algorithms and protocols constitute the central component of systems that protect network transmissions and store data. The security of such systems greatly depends on the methods used to manage, establish, and distribute the keys employed by the cryptographic techniques. Even if a cryptographic algorithm is ideal in both theory and implementation, the strength of the algorithm will be rendered useless if the relevant keys are poorly managed.

1.2 State of the art

Cryptography is the art and science behind the principles, means, and methods for keeping messages secure. Cryptanalysis is a study of how to compromise (defeat) cryptographic mechanism. There are two classes of key-based encryption algorithms: symmetric (or secret-key) and asymmetric (or public-key) algorithms. Symmetric algorithms use the same key for encryption and decryption, whereas asymmetric algorithms use different keys for encryption and decryption. Ideally it is infeasible to compute the decryption key from the encryption key.

Symmetric algorithms can be divided into stream ciphers and block ciphers. Stream ciphers encrypt a single bit of plain text at a time, whereas block ciphers take a number of bits (say 64 bits), and encrypt them as a single unit. Symmetric encryption is the backbone of many secure communication systems. Dozens of symmetric algorithms have been invented and implemented, both in hardware and software.

1.3 Preliminaries

The following notation is necessary for our discussion.

- Hexadecimal numbers will be subscripted with "*h*," e.g., $10_h = 16$.

Bitwise Shifts: The logical shift of x by y bits is denoted by $x << y$. The logical right shift of x by y bits is denoted by $x >> y$.

Bitwise Rotations: A left rotation of x by y bits is denoted by $x <<< y$. A right rotation of x by y bits is denoted by $x >>> y$.

Exclusive-OR: The operation of addition of n-tuples over the field F_2 (also known as exclusive-or) is denoted by $x \oplus y$.

Integer Addition: The operation of integer addition modulo 2^n is denoted by $x \boxplus y$. (where $x, y \in \mathbf{Z}_{2^n}$). The value of n should be clear from the context.

Integer Subtraction: The operation of integer subtraction modulo 2^n is denoted by $x \boxminus y$ (where $x, y \in \mathbf{Z}_{2^n}$). The value of n should be clear from the context.

Feistel ciphers (see Feistel, 1973) are a special class of iterated block ciphers where the cipher text is calculated from the plain text by repeated application of the same transformation or round function. In a Feistel cipher, the text being encrypted is split into two halves. The round function, F, is applied to one half using a sub key and the output of F is (exclusive-or-ed (XORed)) with the other half. The two halves are then swapped. Each round follows the same pattern except for the last round where there is often no swap. The focus of this thesis is the TEA Feistel Cipher.

1.4 Research Study

The Tiny Encryption Algorithm (TEA) is a cryptographic algorithm designed by Wheeler and Needham (1994). It is designed to minimize memory footprint and maximize speed. This research presents the cryptanalysis of the Tiny Encryption Algorithm based on the differential cryptanalysis proposed by Biham and Shamir (1992) and related-key cryptanalysis proposed by Kelsey, Schneier, and Wagner (1997).

1.5 Organization of the Thesis

The body of the thesis is organized as follows.

Chapter 2 provides the necessary background and foundation by reviewing the Tiny Encryption Algorithm developed by the inventors, and the weaknesses pointed out by Wagner and extensions made by the inventors.

Chapter 3 introduces the theoretical framework of cryptanalysis for the Tiny Encryption Algorithm by presenting the common attacks involved in the cryptanalysis of Block cipher. It also presents the results of this research.

Chapter 4 presents conclusions and describes possible directions for future work.

CHAPTER 2

MOTIVATION FOR RESEARCH

2.1 Background

Many symmetric block ciphers have been presented in recent years. The Tiny Encryption Algorithm (TEA) (Wheeler et al., 1994) is a compromise for safety, ease of implementation, lack of specialized tables, and reasonable performance. TEA can replace DES[1] in software, and is short enough to integrate into almost any program on any computer. Some attempts have been made to find weakness of the Tiny Encryption Algorithm. The motivation of this research is to study and implement the proposed attacks on TEA to determine whether such attempts are practically feasible.

2.2 Tiny Encryption Algorithm

The Tiny Encryption Algorithm is a Feistel type cipher (Feistel, 1973) that uses operations from mixed (orthogonal) algebraic groups. A dual shift causes all bits of the data and key to be mixed repeatedly. The key schedule algorithm is simple; the 128-bit key K is split into four 32-bit blocks K = (K[0], K[1], K[2], K[3]). TEA seems to be highly resistant to differential cryptanalysis (Biham et al., 1992) and achieves complete diffusion (where a one bit difference in the plaintext will cause approximately 32 bit differences in the cipher text). Time performance on a workstation is very impressive.

[1] DES (Data Encryption Standard), is an encryption algorithm which has been the world wide standard for 20 years. For a full description, see [6]

2.3 Technique Developed by Inventors

Wheeler et al. (1994) at the computer laboratory of Cambridge University developed the TEA encode routine. Figure 2.1 presents the TEA encode routine in C language where the key value is stored in k[0] – k[2] and data are stored in v[0] – v[1].

2.3.1 Encryption Routine

```
void code(long* v, long* k)  {
unsigned long y = v[0], z = v[1], sum = 0, /* set up */
delta = 0x9e3779b9, n = 32 ;  /* a key schedule constant */
while (n-->0) {         /* basic cycle start */
  sum += delta ;
  y +=  (z<<4)+k[0] ^ z+sum ^ (z>>5)+k[1] ;
  z +=  (y<<4)+k[2] ^ y+sum ^ (y>>5)+k[3] ;  /* end cycle */
       }
 v[0] = y ; v[1] = z ; }
```

Figure 2.1.   Encode routine.

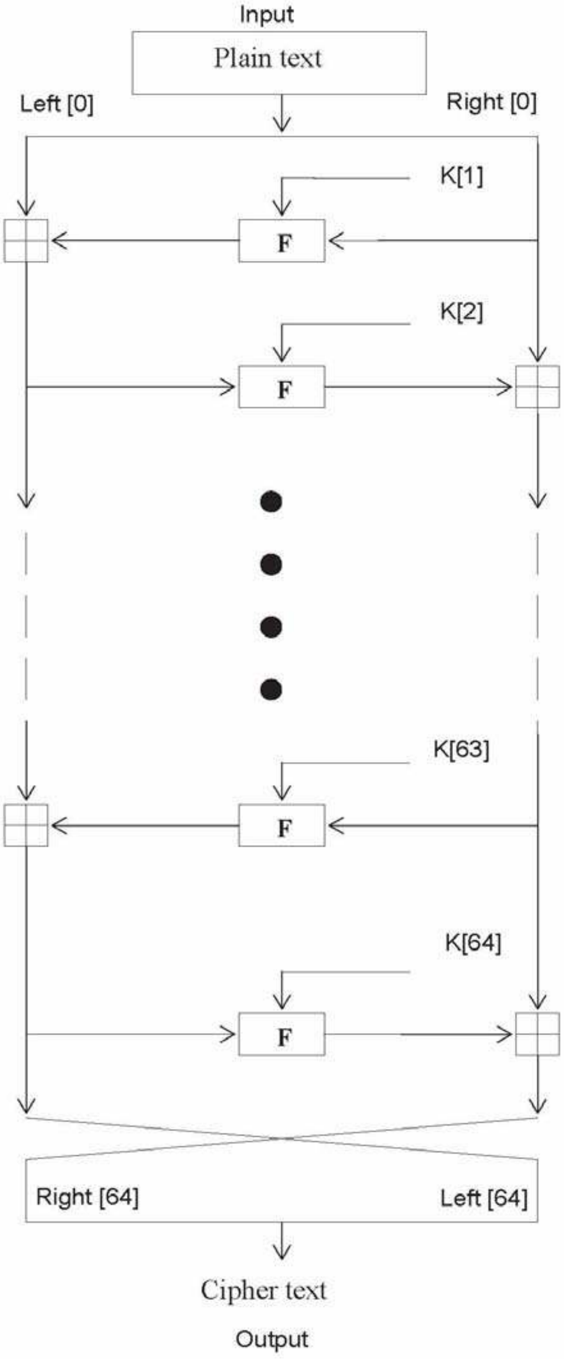


Figure 2.2. The abstract structure of TEA encryption routine.

Figure 2.2 shows the structure of the TEA encryption routine. The inputs to the encryption algorithm are a plaintext block and a key K .The plaintext is P = (Left[0], Right[0]) and the cipher text is C = (Left[64], Right[64]). The plaintext block is split into two halves, Left[0] and Right[0]. Each half is used to encrypt the other half over 64 rounds of processing and then combine to produce the cipher text block.

- Each round *i* has inputs Left[*i*-1] and Right[*i*-1], derived from the previous round, as well as a sub key K[*i*] derived from the 128 bit overall K.
- The sub keys K[*i*] are different from K and from each other.
- The constant delta $=(\sqrt{5}-\ 1)*2^{31}\ =\ 9E3779B9_{h}$, is derived from the golden number ratio to ensure that the sub keys are distinct and its precise value has no cryptographic significance.
- The round function differs slightly from a classical Fiestel cipher structure in that integer addition modulo 2^{32} is used instead of exclusive-or as the combining operator.

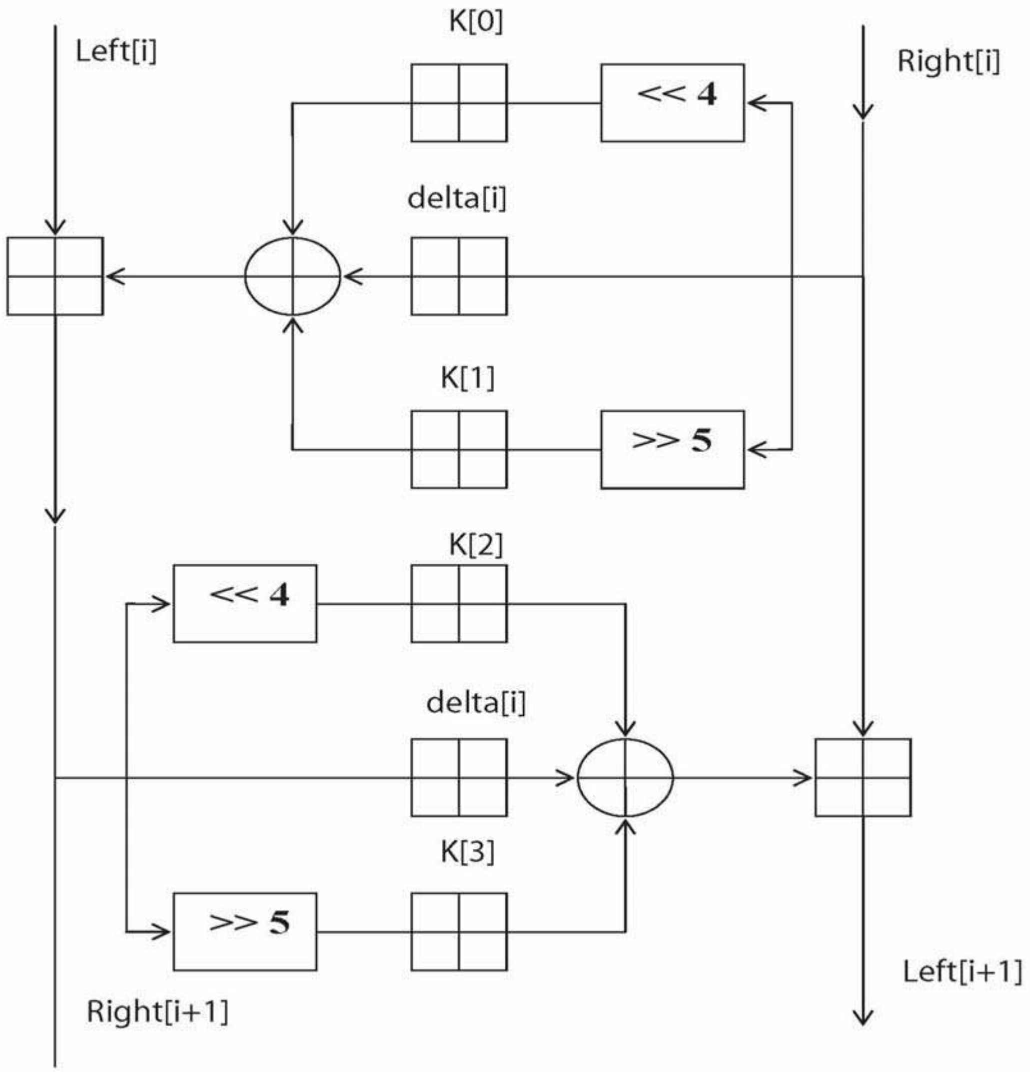


Figure 2.3. An abstraction of *i*-th cycle of TEA.

Figure 2.3 presents the internal details of the *i*th cycle of TEA. The round function, F, consists of the key addition, bitwise XOR and left and right shift operation. We can describe the output (Left[i +1] , Right[i +1]) of the *i*th cycle of TEA with the input (Left[i] ,Right[i]) as follows

$$\text{Left}\,[i{+}1] \;=\; \text{Left}[i] \boxplus \text{F}\,(\,\text{Right}[i],\, \text{K}\,[0, 1],\, \text{delta}[i]\,),$$

$$\text{Right}\,[i\,{+}1] \;=\; \text{Right}[i] \boxplus \text{F}\,(\,\text{Right}[i\,{+}1],\, \text{K}\,[2, 3],\, \text{delta}[i]\,),$$

$$\text{delta}[i] = (i\,{+}1)/2 * \text{delta},$$

The round function, F, is defined by

$$\text{F}(\text{M}, \text{K}[j,k], \text{delta}[i]\;) = ((\text{M} << 4) \boxplus \text{K}[j]) \oplus (\text{M} \boxplus \text{delta}[i]\,) \oplus ((\text{M} >> 5) \boxplus \text{K}[k]).$$

The round function has the same general structure for each round but is parameterized by the round sub key K[i]. The key schedule algorithm is simple; the 128-bit key K is split into four 32-bit blocks K = (K[0], K[1], K[2], K[3]). The keys K[0] and K[1] are used in the odd rounds and the keys K[2] and K[3] are used in even rounds.

2.3.2 Decryption Routine

```
void decode(long* v, long* k)  {
  unsigned long n = 32, sum, y = v[0], z = v[1],
  delta = 0x9e3779b9 ;
  sum = delta<<5 ;
              /* start cycle */
  while (n-->0) {
  z - = (y<<4)+k[2] ^  y+sum ^ (y>>5)+k[3] ;
  y -= (z<<4)+k[0] ^ z+sum ^ (z>>5)+k[1] ;
  sum -= delta ;  }
              /* end cycle */
     v[0] = y ; v[1] = z ;  }
```

Figure 2.4. Decode routine.

Decryption is essentially the same as the encryption process; in the decode routine the cipher text is used as input to the algorithm, but the sub keys K[*i*] are used in the reverse order.

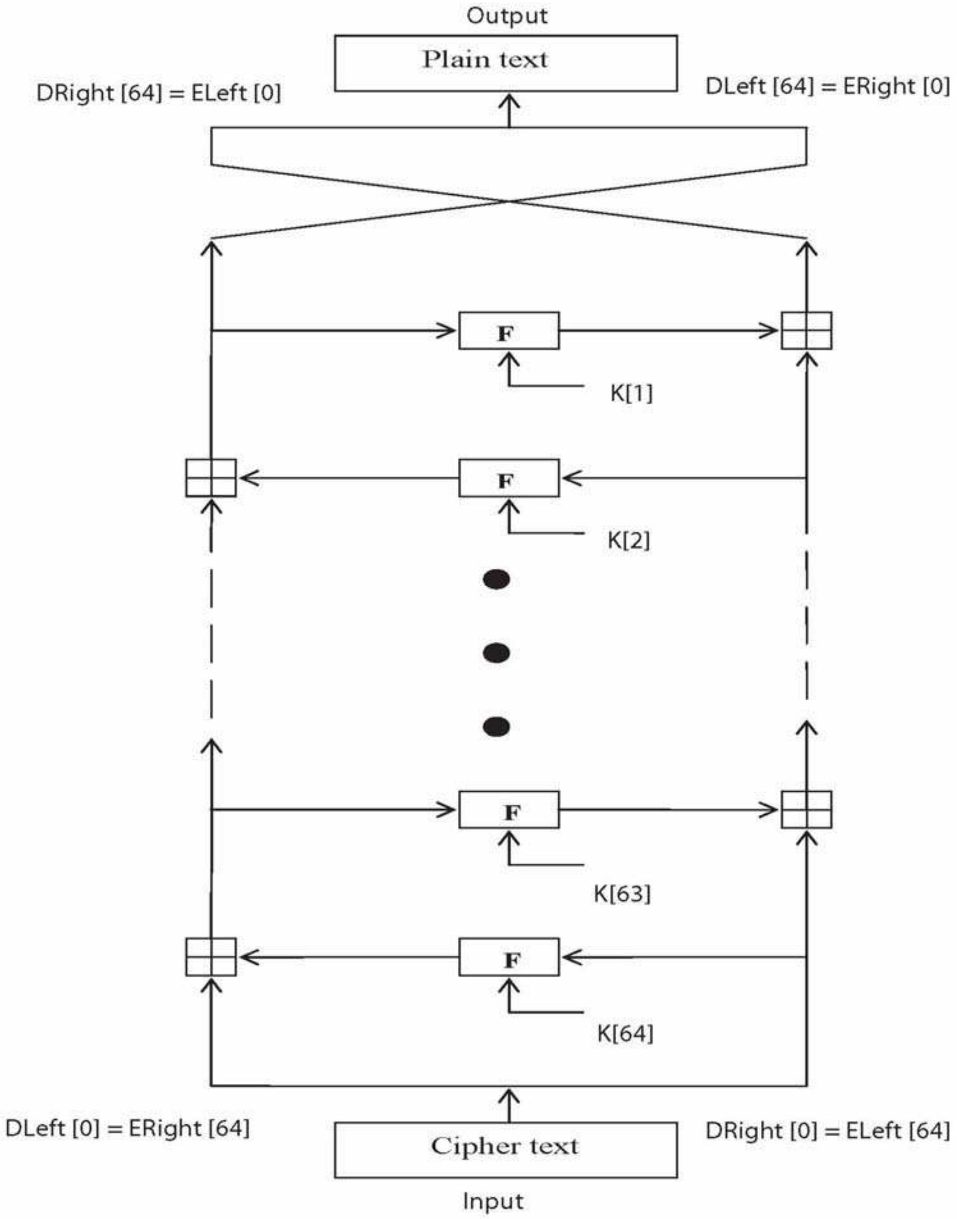


Figure 2.5. The abstract structure of TEA decryption routine.

Figure 2.5 presents the structure of the TEA decryption routine. The intermediate value of the decryption process is equal to the corresponding value of the encryption process with the two halves of the value swapped. For example, if the output of the nth encryption round is

ELeft[*i*] || ERight[*i*]   (ELeft[*i*] concatenated with ERight[*i*]).

Then the corresponding input to the (64-*i*)th decryption round is

DRight[*i*] || DLeft[*i*]   (DRight[*i*] concatenated with DLeft[*i*]).

After the last iteration of the encryption process, the two halves of the output are swapped, so that the cipher text is ERight[64] || ELeft[64], the output of that round is the final cipher text C. Now this cipher text is used as the input to the decryption algorithm. The input to the first round is ERight[64] || ELeft[64], which is equal to the 32-bit swap of the output of the 64th round of the encryption process.

2.4 Extensions of Tiny Encryption Algorithm

TEA was extended to XTEA (Extended TEA) by Wheeler et al. (1997). It was proposed to fix the two minor weaknesses pointed out by Kelsey et al. (1997). Like TEA, XTEA makes use of arithmetic and logic operations. The first enhancement is to adjust the key schedule, and the second is to introduce the key material slowly. Figure 2.6 presents the XTEA routine in C language, where the array v represents the plain text of 2 words, the array k represents the key of 4 words, and N contains the value of number of cycles.

```
XTEA( long * v, long * k, long N)
{
unsigned long y = v[0];
unsigned z =v [1];
unsigned  DELTA = 0x9e3779b9;

if (N>0)
 {
       /* coding */
    unsigned long limit = DELTA*N;
    unsigned long sum = 0 ;
    while (sum != limit)
       y += (z<<4 ^ z>>5) + z ^ sum + k[sum&3],
       sum += DELTA,
       z += (y<<4 ^ y>>5) + y ^ sum + k[sum>>11 &3] ;
              }

 else
  {

        /* decoding */
    unsigned long sum=DELTA*(-N) ;
    while (sum)
        z -=  (y<<4 ^ y>>5) + y ^ sum + k[sum>>11 &3],
        sum -= DELTA,
        y -= (z<<4 ^ z>>5) + z ^ sum + k[sum&3] ;
          }
        v[0]=y;
        v[1]=z ;
        return;
```

Figure 2.6. XTEA routine.

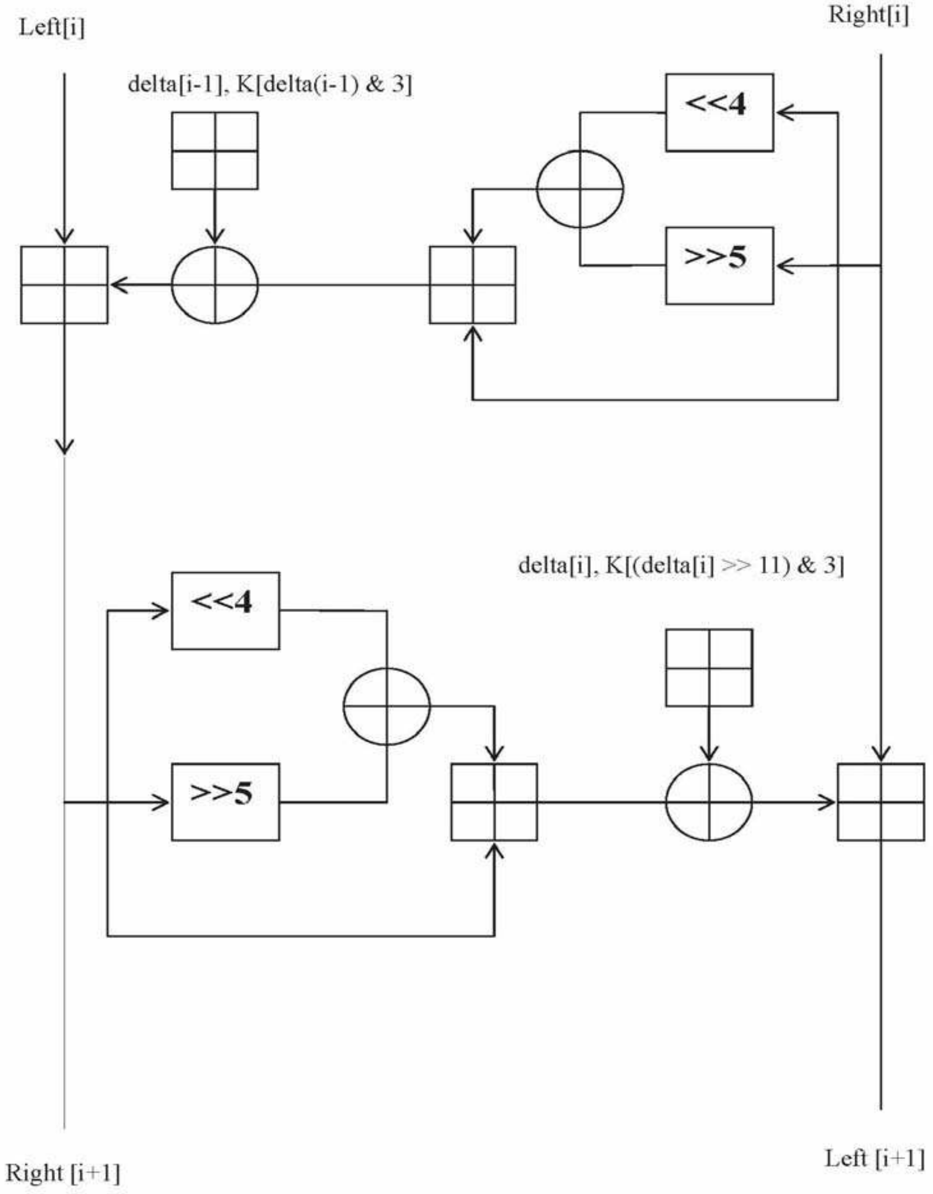

Left[i]
Right[i]
delta[i-1], K[delta(i-1) & 3]
<<4
>>5
delta[i], K[(delta[i] >> 11) & 3]
<<4
>>5
Right [i+1]
Left [i+1]

Figure 2.7. An abstraction of *i*-th cycle of XTEA.

Figure 2.7 shows the internal details of the *i*th cycle of XTEA routine. The relation between the output (Left[*i*+1], Right[*i*+1]) and the input (Left[*i*], Right[*i*]) for the *i*th cycle of XTEA is defined as follows:

$$\text{Left}[i+1] = \text{Left}[i] \boxplus \text{F}(\text{Right}[i], \text{K}[2i-1], \text{delta}[i-1]),$$

$$\text{Right}[i+1] = \text{Right}[i] \boxplus \text{F}(\text{Left}[i+1], \text{K}[2i], \text{delta}[i]),$$

$$\text{delta}[i] = (i+1)/2 * \text{delta}$$

The round function, F, is defined by

$$\text{F}(\text{M}, \text{K}[*], \text{delta}[**]) = ((\text{M}<<4) \oplus (\text{M}>>5)) \boxplus \text{M} \oplus \text{delta}[**] \boxplus \text{K}[*].$$

The round keys are generated according to the algorithm shown in Figure 2.8.

```
➢ Split the 128-bit key K into four 32-bit blocks
      K = (K[0], K[1], K[2], K[3]).
➢ for rounds r = 1,….,64
     if (r is odd) then
           key = K[delta((r-1)/2) & 3];
    else
     if ( r is even) then
           key = K[(delta((r/2)>>11) & 3]
```

Figure 2.8. Round key generation algorithm.

2.5 Block TEA

Block TEA is a block version of XTEA (Wheeler et al., 1997). Figure 2.9 presents the Block TEA in C language. It will encode or decode n words as a single block where n > 1. In Figure 2.9 v represents the n word data vector and k is the four word key.

```
long Block_TEA( long * v, long n , long * k ) {
unsigned long z = v[n-1], sum=0,e,
      DELTA = 0x9e3779b9 ;
long m, p, q ;
if ( n>1) {
       /* Coding Part */
    q = 6+52/n ;
    while (q-- > 0)     {
    sum += DELTA ;
    e = sum>>2&3 ;
    for (p=0 ; p<n ; p++ )
       z = v[p] += (z<<4 ^ z>>5) + z ^ k[p&3^e] + sum ;
                                          }
  return 0 ;

       /* Decoding Part */
  else if (n<-1) {
      n = -n ;
     q = 6+52/n ;
     sum = q*DELTA ;
     while (sum != 0) {
     e = sum>>2 & 3 ;
     for (p = n-1 ; p>0 ; p-- ) {
          z = v[p-1] ;
          v[p] -= (z<<4 ^ z>>5) + z ^ k[p&3^e] + sum ;
                                        }
      z = v[n-1] ;
         v[0] -= (z<<4 ^ z>>5) + z ^ k[p&3^e] + sum ;
      sum -= DELTA ;  }
   return 0 ;  }
   return 1 ;  } /* Signal n=0 */
```

Figure 2.9. Block TEA routine.

The basic principle of mixing data with the key is kept similar to TEA, but we run cyclically through the block. The following equation represents the mix operation

$$v[n] += mix(v[n-1], key)$$

The mix operation is performed along the words of a block and then around the whole block operation a number of times.

CHAPTER 3

CRYPTANALYSIS

Cryptanalysis is an art of analyzing and breaking ciphers; it is an attempt to take cipher text produced by an adversary, and produce the plaintext or, better yet, the key. It is also a relatively new modern science, which has become more and more popular with an advent of modern cryptography. The term attack in this context has the following implication: In intuitive terms a (passive) attack on a cryptosystem is any method of starting with some information about plaintexts and their corresponding cipher texts under some (unknown) key, and figuring out more information about the plaintexts. It is possible to state mathematically as follows:

Let the encryption system E is a collection of functions E_K, indexed by keys K, mapping some set of plain texts P to some set of cipher texts C. Similarly the decryption system D is a collection of functions D_K such that D_K(E_K(P)) = P for every plaintext P. Now fix the input functions F (for Plain texts), G (for Keys), and the output function H (for cipher texts) for n variables. Fix an encryption system E and fix a distribution of plaintexts and keys. An attack on E using G assuming F giving H with probability p is an algorithm A with a pair f, g of inputs and one output h, such that there is probability p of computing h = H(P_1,...,P_n), if we have f = F(P_1,...,P_n) and g = G(E_K(P_1),...,E_K(P_n)).

3.1 Attacks

In this research we inspected the most common methods applied in the cryptanalysis of a block cipher algorithm; the following attacks are performed for this research.

3.1.1 Cipher text only attack: An attack against (i.e., an attempt to decrypt) cipher text when only the cipher text itself is available (i.e., there is no known plaintext nor key associated with the cipher text), in mathematical notation, a cipher text only attack is one where F is constant. Given only some information G(E_K(P_1),...,E_K(P_n)) about n cipher texts, the attack has to have some chance of producing some information H(P_1,...,P_n) about the plain texts.

3.1.2 Known plaintext attack: The attacker knows or can guess the plaintext for some parts of the cipher text. The task is to decrypt the rest of the cipher text blocks using this information. This may be done by determining the key used to encrypt the data or via some shortcut. In mathematical notation, known plaintext attack has F(P_1,P_2) = P_1, G(C_1,C_2) = (C_1,C_2), and H(P_1,P_2) depending only on P_2. In other words, given two cipher texts C_1 and C_2 and one decryption P_1, the known plaintext attack should produce information about the other decryption P_2.

3.1.3 Chosen plaintext attack: The attacker is able to have any text encrypted with the unknown key. The task in the chosen plaintext attack is to determine the key used for encryption.

This research found that TEA is highly resistive to the cipher text only, known plain text, chosen Plaintext attacks, this research examined plaintexts with one bit difference to find any similarities in the cipher texts and TEA produced approximately 32 bit differences in the cipher text for one bit difference in the plaintext just after six rounds.

3.2 Differential Cryptanalysis

Biham et al. (1992) developed a method of attacking block ciphers, which they call differential cryptanalysis. This attack is the general method of attacking cryptographic algorithms. It has exposed the weakness in many algorithms. It looks specifically at cipher text pairs: pairs of cipher texts whose plain texts have particular differences and analyzes the evolution of these differences as the plain texts propagate through the rounds of the encryption algorithm when they are encrypted with the same key. The two plain texts (with a fixed difference) can be chosen at random as long as they satisfy particular differences. Then, using the differences in the resulting cipher texts, assign different probabilities to different keys. As we analyze more and more cipher texts, one key will emerge as the most probable (correct key).

3.3 Related Key Cryptanalysis

It is similar to differential cryptanalysis, but it examines the differences between keys. In this attack we need to choose a relationship between a pair of keys, but does not know the keys themselves. It relies on simple relationship between sub keys in adjacent rounds, encryption of plain texts under both the original (unknown) key K, and some derived keys $K^* = f(K)$. We need to specify how the keys are to be changed; we may be able to flip bits in the key without knowing the key. TEA admits several related-key attacks which arise from the severe simplicity of its key schedule. The following attacks are discovered by Kelsey et al. (1997).

3.3.1 Attack one: The output of the round function, F, in the even rounds will remain the same with a probability of nearly 0.5, if we simultaneously flip the next most significant bit (bit number 30) of K[2] and K[3]. This will yield a 2-round iterative differential characteristic with probability of 0.5, and thus a 60-round characteristic with probability 2^{-30}. Analysis of the discoverers indicate that a 4R differential related-key attack can break 64-round TEA with one related-key query and about 2^{34} chosen plaintexts.

3.3.2 Attack two: This attack is similar to the first attack. Under the key $K[0..3]$ we request the encryption of (Left[*i*], Right[*i*]) and under key

$$K^{*}[0..3] = K[0..3] \oplus (0, 2^{31} \oplus 2^{26}, 0, 0)$$

we request the encryption of (Left[*i*], Right[*i*] $\oplus$ 2^{31}). Examining the three terms of F(Right[*i*] , K[0,1], delta) when bit 31 of Right[*i*] is flipped along with bits 26 and 31 of K[1], we see: The high bit of (Right[*i*] + delta) is always changed, neither change has any effect on (the value of Right[*i*] after left shifting four times + K[0]) and half the time, only the high bit is changed for (the value of Right[*i*] after right shifting five times + K[1]). If we choose one key difference it gives a one-cycle iterative differential characteristic with probability of 0.5.

3.4 Impossible Differential Cryptanalysis

Cryptanalysis with impossible differentials was introduced by Biham et al. (1992) and is powerful. The normal differential cryptanalysis finds a key using the differential characteristic with high probability. But, this attack uses the differential cryptanalysis with probability zero which is called impossible differentials. Initially we need to find an impossible differential characteristic, we then choose any plain text pairs with the input difference of the impossible differential characteristic, and obtain the corresponding cipher text pairs. Using the special property derived from the impossible characteristic we eliminate the cipher text pairs with the input difference of the impossible differential characteristic. If we can find no impossible differentials for a cipher, the cipher cannot be attacked by cryptanalysis with impossible differentials. In the last one or two rounds for every key value in the key space, we decrypt the cipher text pairs with that key value and we eliminate the key value from the key space if the differences of the decrypted cipher text satisfy the output difference of the impossible differential characteristic. We repeat the above process until the only one key value remains with very high probability. Generally speaking, the search for impossible differentials is difficult because much complexity is required to guarantee completeness. Some researchers (e.g., Moon, Hwang, Lee, Lee, & Lim, 2002) used this method and found the 12-round impossible differential cryptanalysis of the XTEA and the 10-round impossible differential cryptanalysis of TEA. They were able to successfully attack 14-round XTEA using the 12-round impossible differential characteristic with $2^{62.5}$ chosen plain texts and 2^{85} encryptions , they were also successful in attacking 11-round TEA using the 10-round impossible differential with $2^{52.5}$ chosen plain texts and 2^{84} encryptions. This research investigated the characteristics discovered by Moon et al. (2002) and found it to be true.

3.5 Equivalent Keys

In a cipher system if the encryption of Plain text with keys K and K^* produces the same cipher text then the two keys are said to be equivalent keys. This can be represented as an equivalence relation $E_K(P) \;\; = E_{K^*}(P)$. We can define equivalence classes and say that a pair of keys K and K^* belong to the same equivalence class if they satisfy the above relation. In the design of a good cipher every key which belongs to the equivalence class should have a distinct mapping of plain text to cipher text and there should not be any equivalent keys so we expect 2^k equivalence classes. As the TEA uses 128-bit key we expect the TEA cipher to have 2^{128} equivalence classes, but research shows different as TEA has 2^{126} equivalence classes (there are 2^{126} distinct mappings of plain texts to cipher text) and not 2^{128} equivalence classes. This weakness is because of the round function (the core of any feistel cipher) of the TEA, the following explanation gives a clear picture of why the key space is only 126 and not 128.

for all values of a, b $\in \mathbf{Z}_{2^{32}}$

$$2^{31} \boxplus 2^{31} = 0,$$

$$a \boxplus 2^{31} = a \boxplus 80000000_h$$

So,

$$a \boxplus b = (a \oplus 80000000_h) \boxplus (b \oplus 80000000_h)$$

in the same way the round function can be manipulated to prove

F(M, K[j,k], delta[i]) = F(M, (K[j] $\oplus$ 80000000_h, K[k] $\oplus$ 80000000_h, delta[i]))

This implies that every 128-bit key K = (K[0], K[1], K[2], K[3]) has three equivalent keys of the form:

(K[0], K[1], K[2] $\oplus$ 80000000_h, K[3] $\oplus$ 80000000_h)

(K[0] $\oplus$ 80000000_h, K[1] $\oplus$ 80000000_h, K[2], K[3])

$(K[0] \oplus 80000000_h, K[1] \oplus 80000000_h, K[2] \oplus 80000000_h, K[3] \oplus 80000000_h)$

this means that even though TEA uses a 128-bit key it produces the same security as a 126-bit key.

The equivalent keys for TEA are as follows

$00000000\ \ 80000000\ \ 00000000\ \ 00000000_h$

$80000000\ \ 00000000\ \ 00000000\ \ 00000000_h$

$80000000\ \ 00000000\ \ 80000000\ \ 80000000_h$

$00000000\ \ 80000000\ \ 80000000\ \ 80000000_h$

for example, encrypting the plain text $(00000000\ 00000000_h)$ with any of the above four keys will produce the same cipher text $(9327C497\ 31B08BBE_h)$. The presence of equivalent keys makes TEA unsuitable for use in a hash function which is based on block ciphers.

3.6 First Correction Made by the Designers

XTEA was presented to cater the weaknesses and attacks mentioned above by the designers of TEA, In XTEA the designers have made the following adjustments: first adjustment was to adjust the key schedule, and the second is to introduce the key material slowly. Designers have corrected the round function

$F(M, K[j,k], \text{delta}[i]) = ((M \ll 4) \boxplus K[j]) \oplus (M \boxplus \text{delta}[i]) \oplus ((M \gg 5) \boxplus K[k])$.

to $F(M, K[*], \text{delta}[**]) = ((M \ll 4) \oplus (M \gg 5)) \boxplus M \oplus \text{delta}[**] \boxplus K[*]$.

This correction brings the following advantages:

- It corrects the mixing proportion of TEA

- Corrects the related-key attacks and equivalence classes for TEA
- Shift of 11 causes the sequence to be irregular, and all 4 keywords are used in the first two cycles.

3.7 Comparison of TEA and XTEA with the Equivalent Keys

Plain Text	Key	Cipher with TEA	Cipher with XTEA
0000000000000000	0000000080000000 0000000000000000	9327c497 31b08bbe	4f190ccf c8deabfc
0000000000000000	8000000000000000 0000000000000000	9327c497 31b08bbe	57e8c05 50151937
0000000000000000	8000000000000000 8000000080000000	9327c497 31b08bbe	31c4e2c6 347b2de
0000000000000000	0000000080000000 8000000080000000	9327c497 31b08bbe	ed69b785 66781ef3

Figure 3.1. Comparison of TEA and XTEA with the equivalent keys.

3.8 Second Correction Made by the Designers

Considering the lack of back propagation in Block TEA the designers have corrected the corrected the round function (see Wheeler et al., 1998) to make the decoding difference propagation about as fast as the forward coding mode.

The mix operation along the words off the block

v[n] + = mix (v[n-1], key)

is corrected to v[m] + =f(v[m-1], v[m+1]).

The corrected round function is given below:

```
#define  MX  (z>>5 ^ y<<2) + (y>>3 ^ z<<4) ^ (sum ^ y) + (k[p&3 ^ e] ^ z);
```

3.9 Intermediate Cipher Text Values for Small Key Pair Differences

In this research we generated and analyzed the cipher texts of TEA and XTEA for the first 32 rounds. Figure 3.2 presents the sample output of Encoded data (cipher texts) obtained by linearly incrementing a single bit in the keys k[0], k[1], k[2], and k[3] with the initial plaint text as “00000000 00000000” (stored in variable v).

```
For 1 round: Incrementing k[0] by  1        For 1 round: Incrementing k[1] by  1
Input Data:  v = 0x0  0                      Input Data:  v = 0x0  0
Key = 0x0 0 0 0                              Key = 0x0 0 0 0
Encoded data = 0x9e3779b9 dbe8d32f           Encoded data = 0x9e3779b9 dbe8d32f
Key = 0x1 0 0 0                              Key = 0x0 1 0 0
Encoded data = 0x60a9b239 d3681922           Encoded data = 0x60a9b23b d3681904
Key = 0x2 0 0 0                              Key = 0x0 2 0 0
Encoded data = 0xa22ef269 3b005643           Encoded data = 0xa22ef472 3b0077ac
Key = 0x3 0 0 0                              Key = 0x0 3 0 0
Encoded data = 0xb199be6 5390bd63            Encoded data = 0xb177cd7 5361f7b2
Key = 0x4 0 0 0                              Key = 0x0 4 0 0
Encoded data = 0xd57900a9 761c765a           Encoded data = 0xd0348261 bf0475cb
Key = 0x5 0 0 0                              Key = 0x0 5 0 0
Encoded data = 0x4b9d76ad c87be75d           Encoded data = 0x78b912e8 5da8e081
```

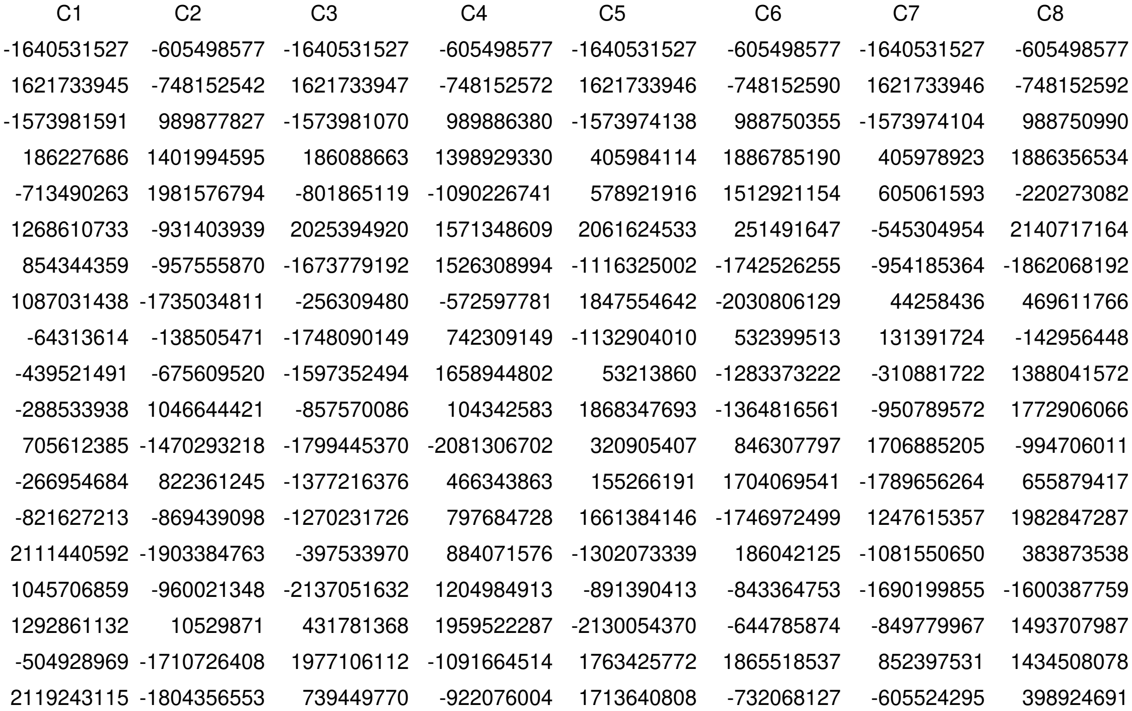
```
For 1 round: Incrementing k[2] by  1        For 1 round: Incrementing k[3] by  1
Input Data:  v = 0x0  0                      Input Data:  v = 0x0  0
Key = 0x0 0 0 0                              Key = 0x0 0 0 0
Encoded data = 0x9e3779b9 dbe8d32f           Encoded data = 0x9e3779b9 dbe8d32f
Key = 0x0 0 1 0                              Key = 0x0 0 0 1
Encoded data = 0x60a9b23a d36818f2           Encoded data = 0x60a9b23a d36818f0
Key = 0x0 0 2 0                              Key = 0x0 0 0 2
Encoded data = 0xa22f0f86 3aef2213           Encoded data = 0xa22f0fa8 3aef248e
Key = 0x0 0 3 0                              Key = 0x0 0 0 3
Encoded data = 0x1832d372 70760ea6           Encoded data = 0x1832bf2b 706f8436
Key = 0x0 0 4 0                              Key = 0x0 0 0 4
Encoded data = 0x2281a5bc 5a2d5842           Encoded data = 0x241081d9 f2dee646
Key = 0x0 0 5 0                              Key = 0x0 0 0 5
Encoded data = 0x7ae1e4d5 efd753f            Encoded data = 0xdf7f4e86 7f98c06c
```

Figure 3.2. Sample output of data obtained by linearly incrementing a single bit in the keys k[0], k[1], k[2] and k[3] on an Input data (for v = 00000000 00000000)

3.10 Statistical Analysis

In the process of obtaining the statistical analysis of cipher texts, this research generated the decimal equivalent values of the cipher texts, Figure 3.3 shows the sample decimal values for the data presented in Figure 3.2.

The sample cipher text “0x9e3779b9 dbe8d32f” is converted into decimal equivalent “-1640531527 -605498577.” The first half’s (right half’s) of these cipher texts are presented in C1 (column one) and the second half’s in the C2 (column two) of Figure 3.3.



C1	C2	C3	C4	C5	C6	C7	C8
-1640531527	-605498577	-1640531527	-605498577	-1640531527	-605498577	-1640531527	-605498577
1621733945	-748152542	1621733947	-748152572	1621733946	-748152590	1621733946	-748152592
-1573981591	989877827	-1573981070	989886380	-1573974138	988750355	-1573974104	988750990
186227686	1401994595	186088663	1398929330	405984114	1886785190	405978923	1886356534
-713490263	1981576794	-801865119	-1090226741	578921916	1512921154	605061593	-220273082
1268610733	-931403939	2025394920	1571348609	2061624533	251491647	-545304954	2140717164
854344359	-957555870	-1673779192	1526308994	-1116325002	-1742526255	-954185364	-1862068192
1087031438	-1735034811	-256309480	-572597781	1847554642	-2030806129	44258436	469611766
-64313614	-138505471	-1748090149	742309149	-1132904010	532399513	131391724	-142956448
-439521491	-675609520	-1597352494	1658944802	53213860	-1283373222	-310881722	1388041572
-288533938	1046644421	-857570086	104342583	1868347693	-1364816561	-950789572	1772906066
705612385	-1470293218	-1799445370	-2081306702	320905407	846307797	1706885205	-994706011
-266954684	822361245	-1377216376	466343863	155266191	1704069541	-1789656264	655879417
-821627213	-869439098	-1270231726	797684728	1661384146	-1746972499	1247615357	1982847287
2111440592	-1903384763	-397533970	884071576	-1302073339	186042125	-1081550650	383873538
1045706859	-960021348	-2137051632	1204984913	-891390413	-843364753	-1690199855	-1600387759
1292861132	10529871	431781368	1959522287	-2130054370	-644785874	-849779967	1493707987
-504928969	-1710726408	1977106112	-1091664514	1763425772	1865518537	852397531	1434508078
2119243115	-1804356553	739449770	-922076004	1713640808	-732068127	-605524295	398924691

Figure 3.3. Decimal equivalent values of the encoded cipher texts.

This research discovered the following properties of generated cipher texts:

- The cipher text values in any column do not start or end at any particular sequence nor exhibit any common property even though there generation was due to a single bit key difference. This is presented in the Figure 3.4.

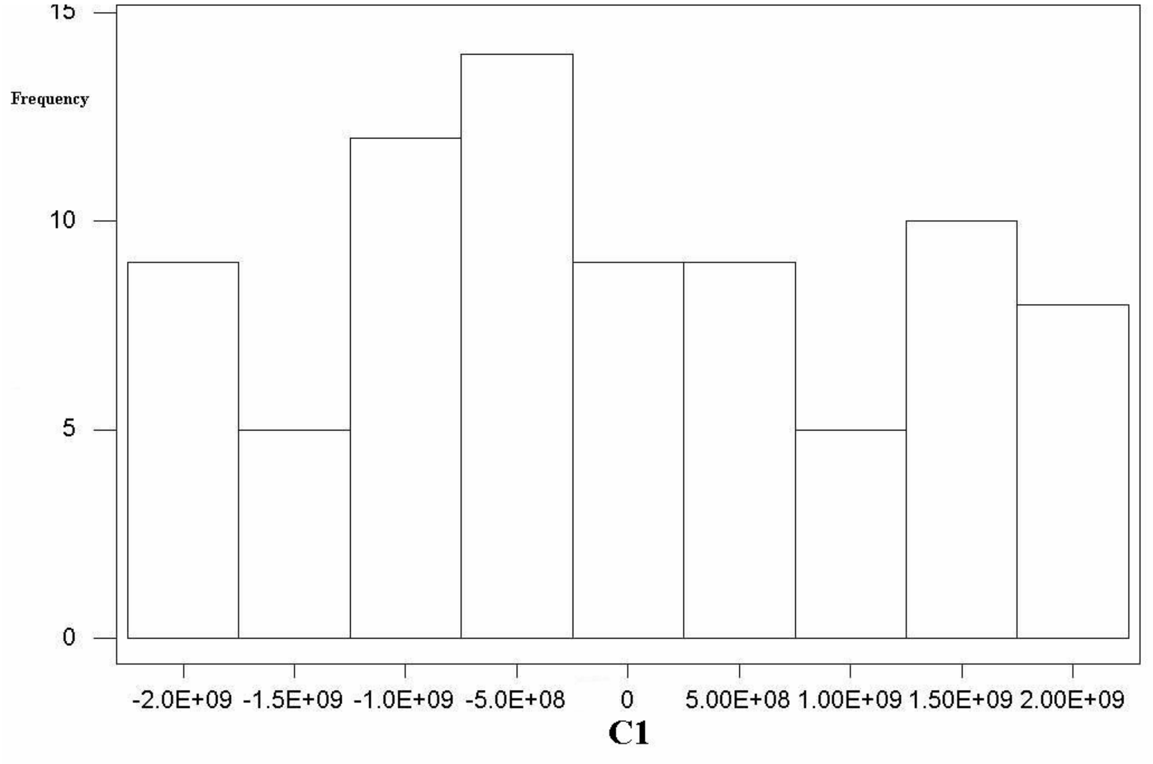


Figure 3.4. Histogram of the cipher text values in column one (C1).

- There are no common properties among the values of the cipher texts generated by linearly incrementing only a single bit on k[0] and [1], This is presented in Figure 3.5.

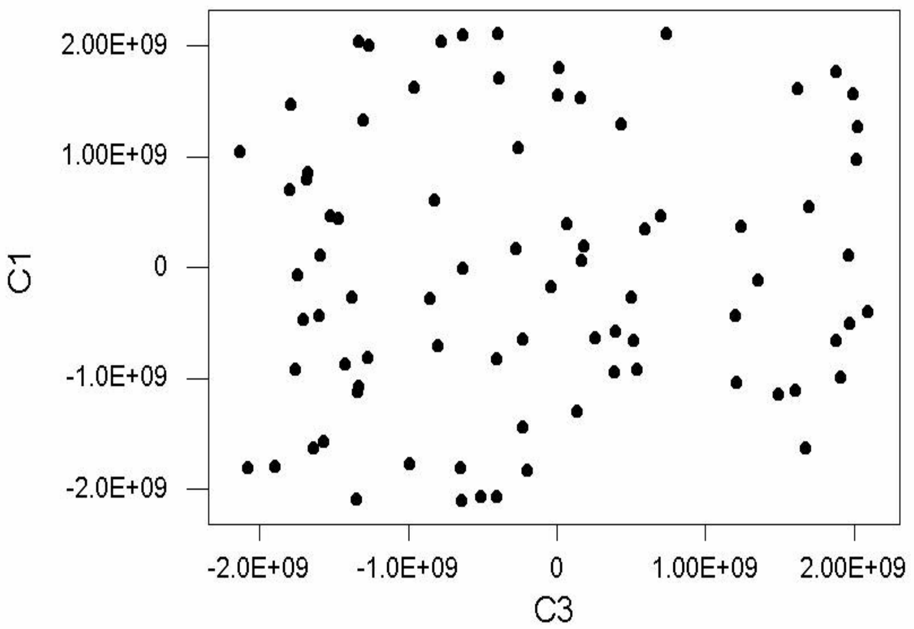


Figure 3.5. Comparison of column one (C1) with column three (C3).

- Comparison of cipher text values obtained by incrementing k[0] with k[1] and k[2] gave no common attributes, This is presented in Figure 3.6.

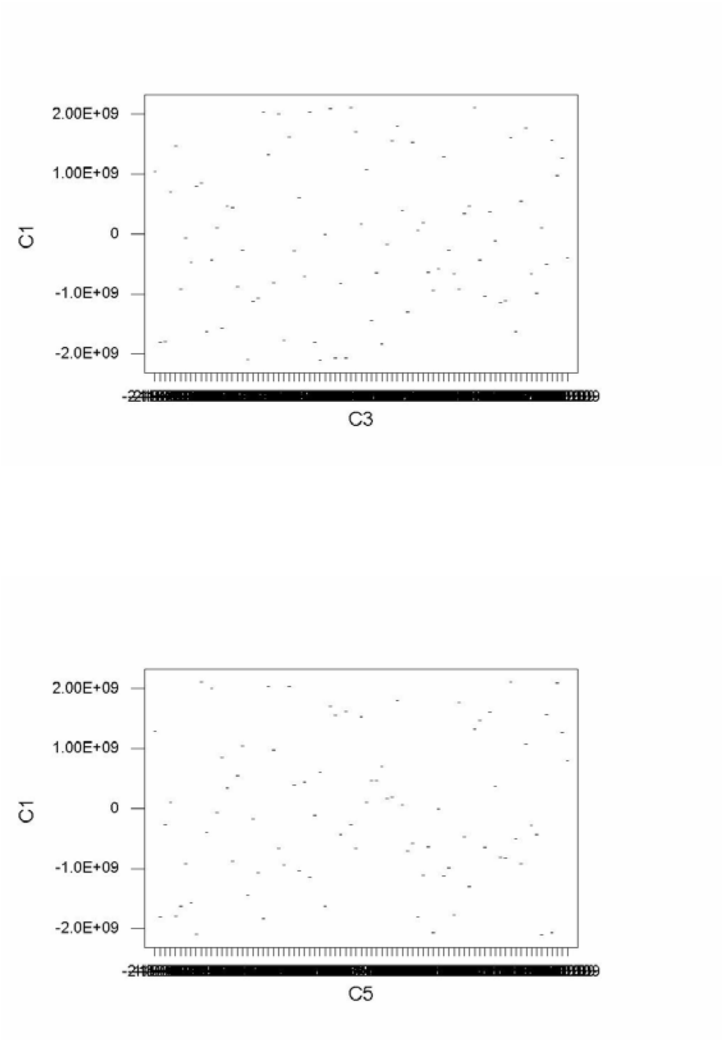


Figure 3.6. Comparison of columns one-three with column one-five.

- There are no attributes which can be derived by making a comparison with values any column to values in any other column, This is presented in Figure 3.7

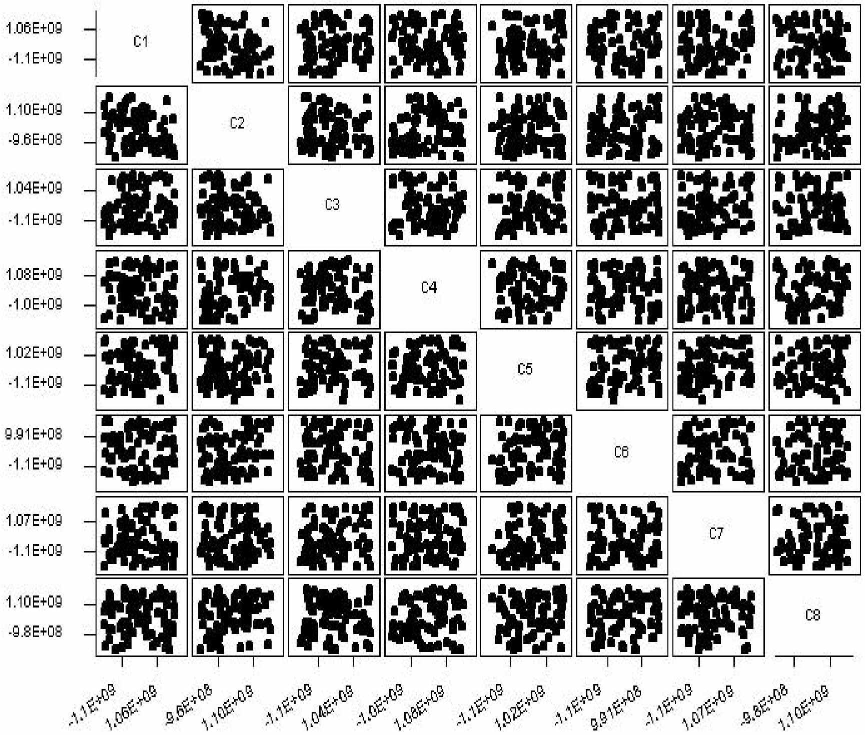


Figure 3.7. Comparison of cipher text values in each column with the cipher text values in other columns.

In order to derive any Meta properties (if present), this research has examined the spatial distribution of cipher text values for the Left half and the Right half separately. For the Left half, columns C1, C3, and C5 are compared. Figures 3.8, 3.9, and 3.10 present this comparison.

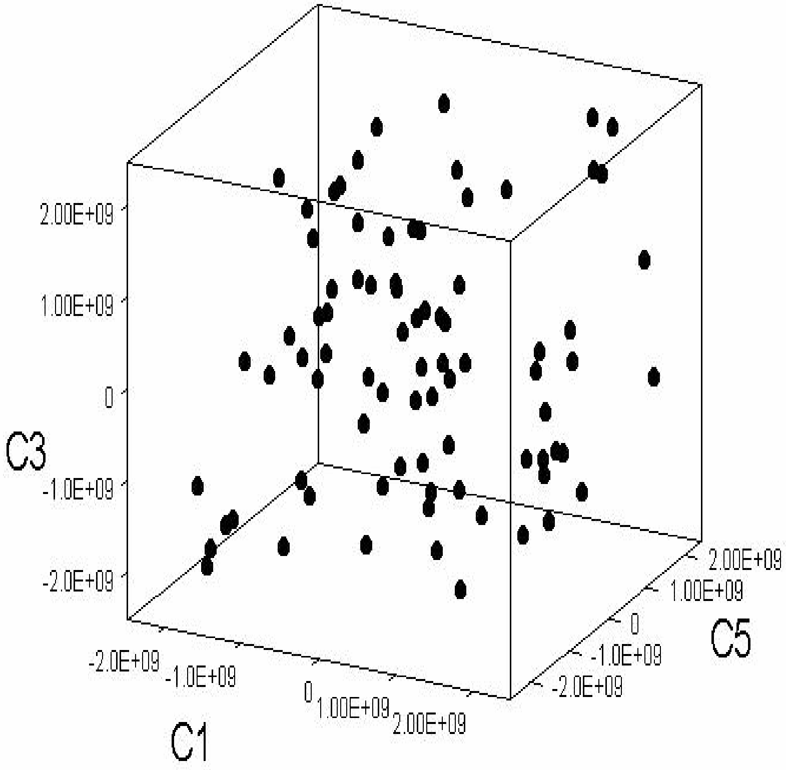


Figure 3.8. 3D plot of cipher text values in columns C1, C3, and C5.

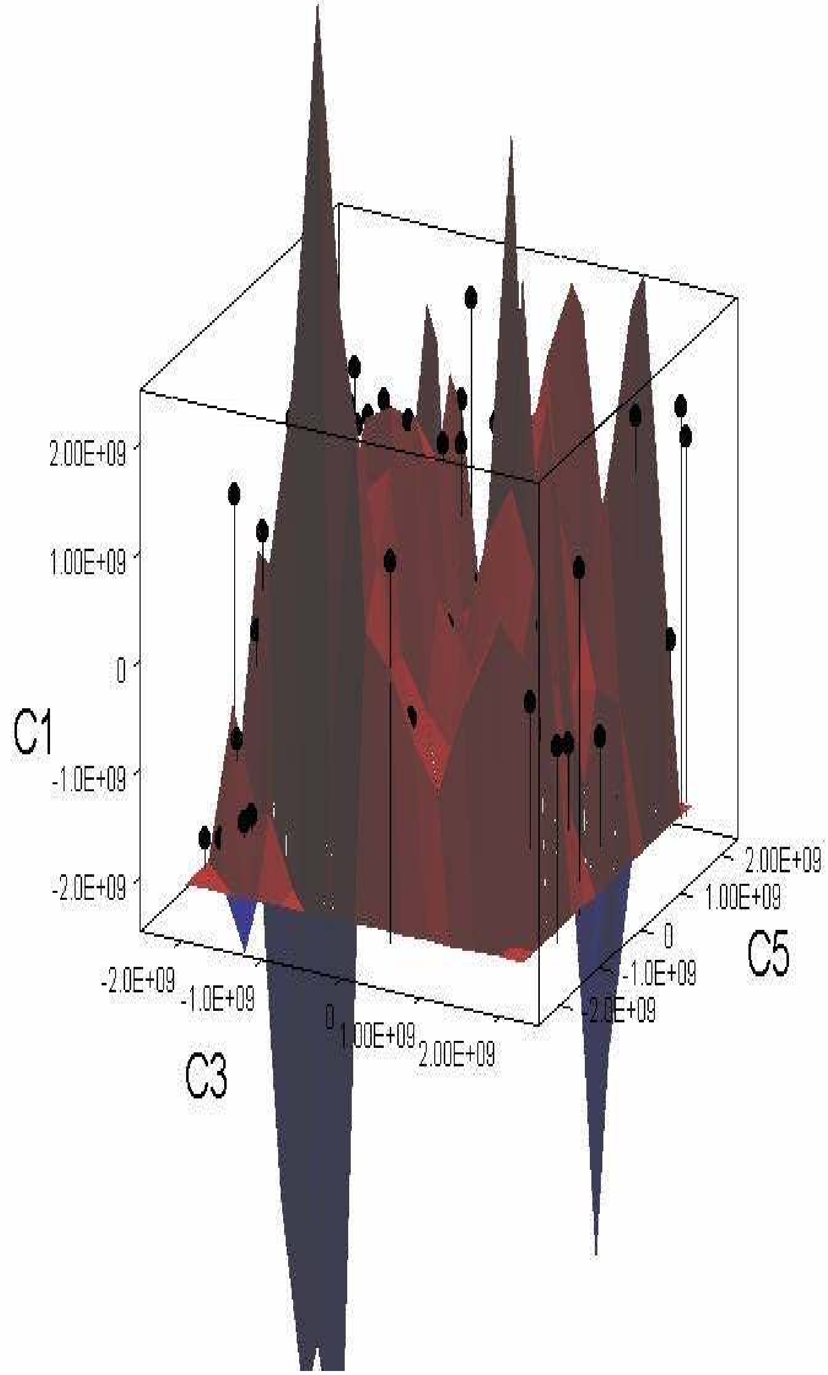


Figure 3.9. 3D surface plot of cipher text values in columns C1, C3, and C5.

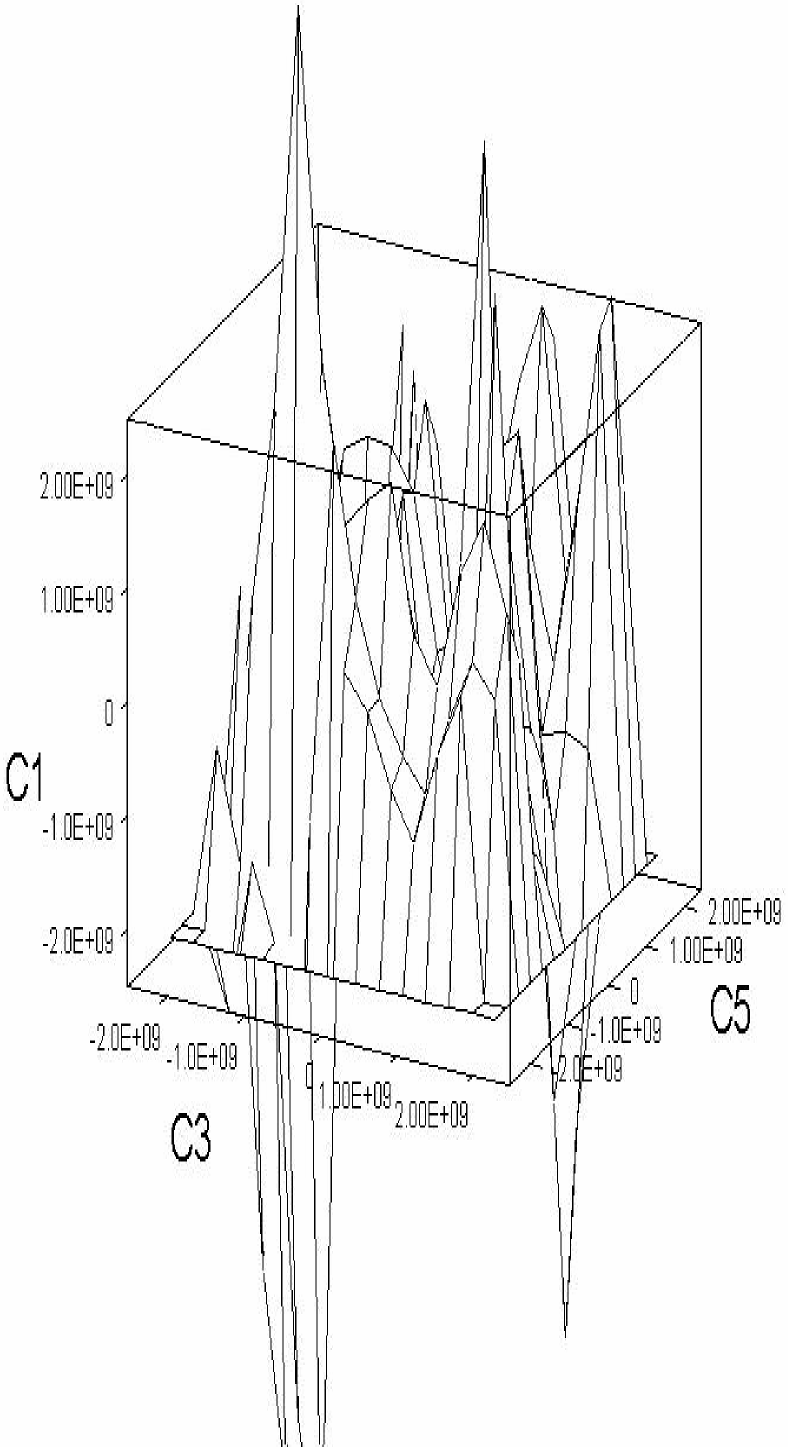


Figure 3.10. 3D wire frame plot of cipher text values in columns C1, C3, and C5.

For the Right half, columns C2, C4, and C6 are compared. Figures 3.11, 3.12, and 3.13 present this comparison.

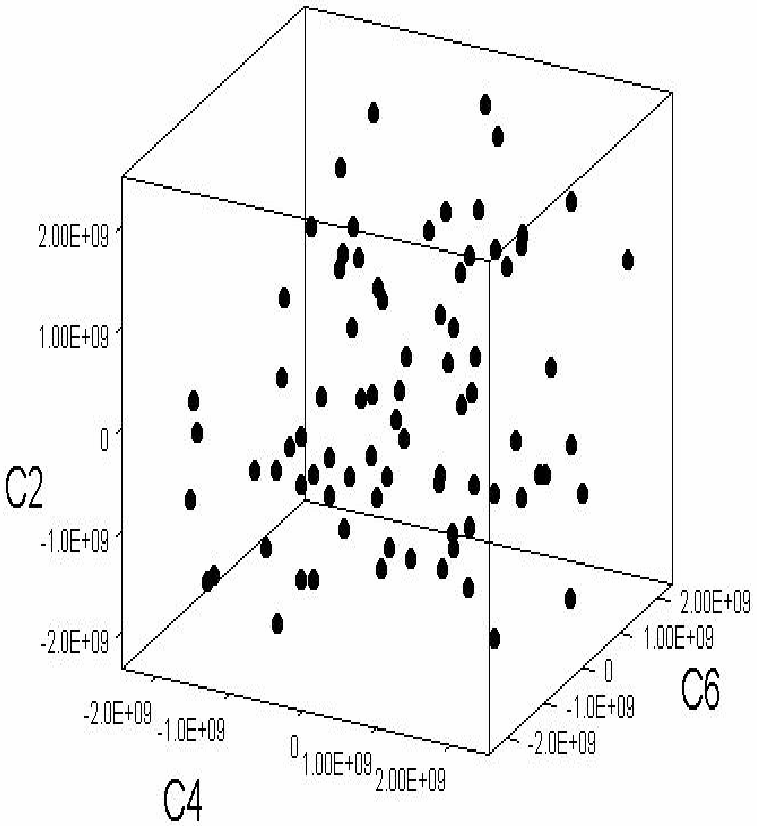


Figure 3.11. 3D plot of cipher text values in columns C2, C4, and C6.

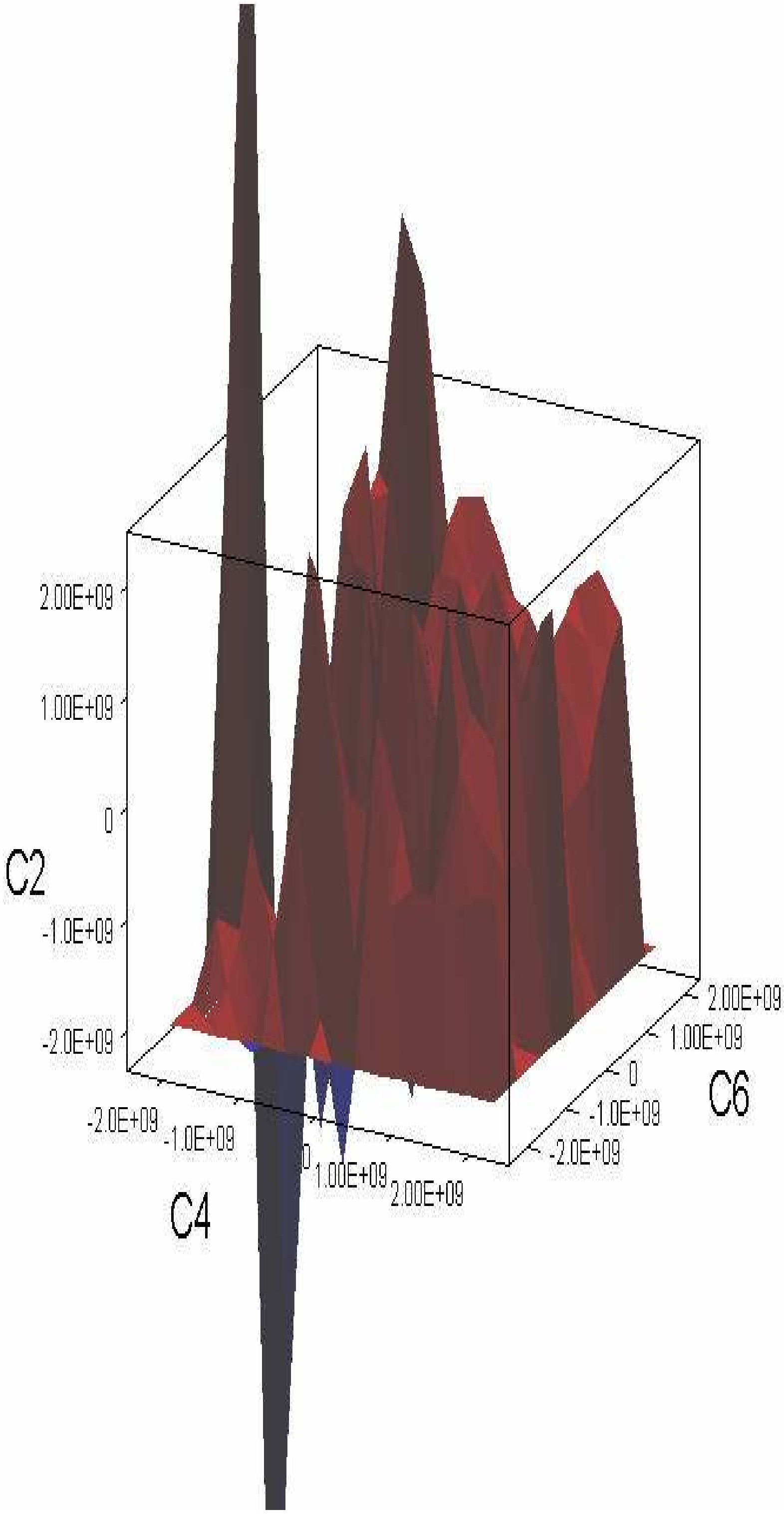


Figure 3.12. 3D surface plot of cipher text values in columns C2, C4, and C6.

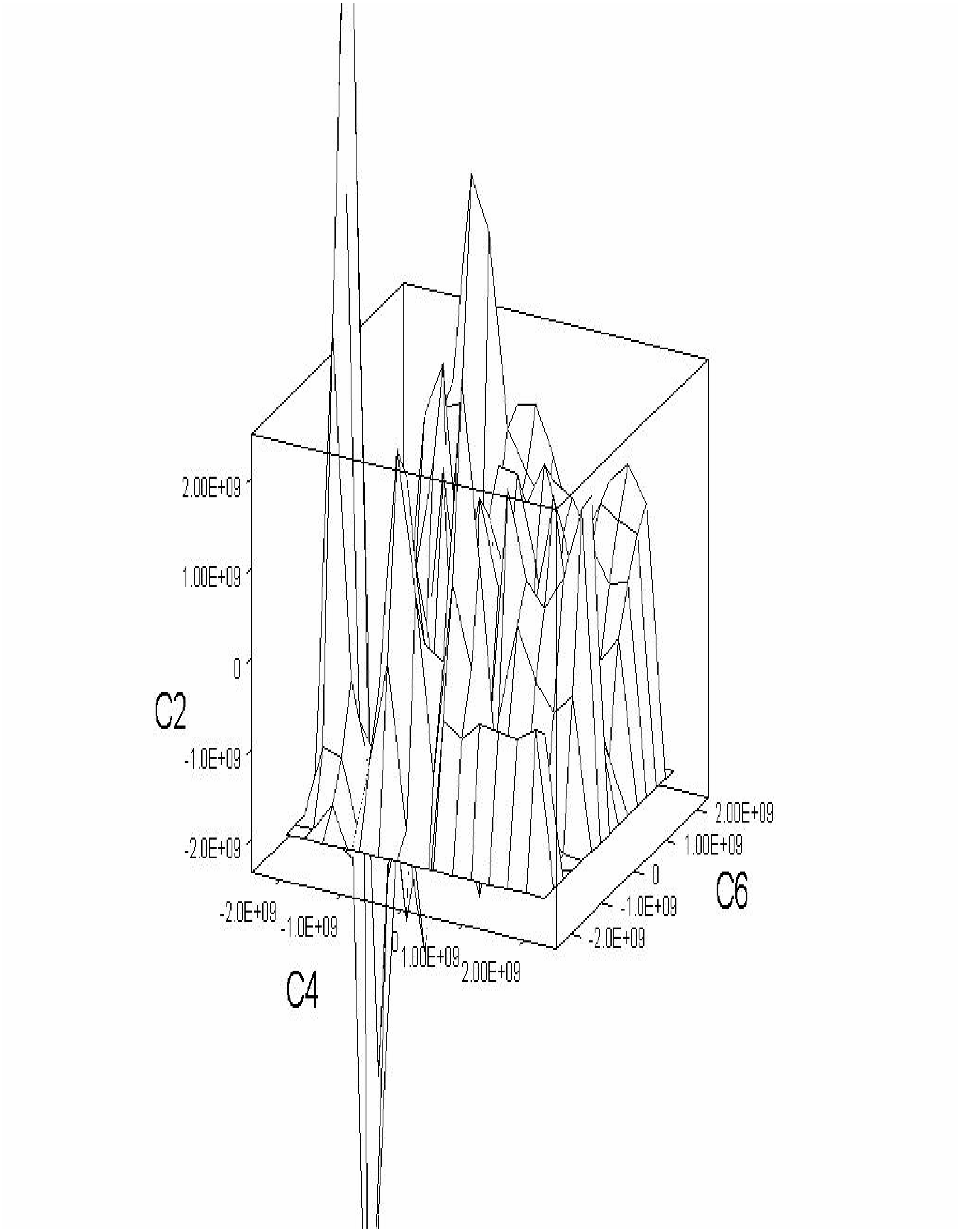


Figure 3.13. 3D wire frame plot of cipher text values in columns C2, C4, and C6.

This research generated the Normal probability plot for the occurrences of cipher text values in each column and this is presented in the graph 3.14 for the Left half and 3.15 for the right half.

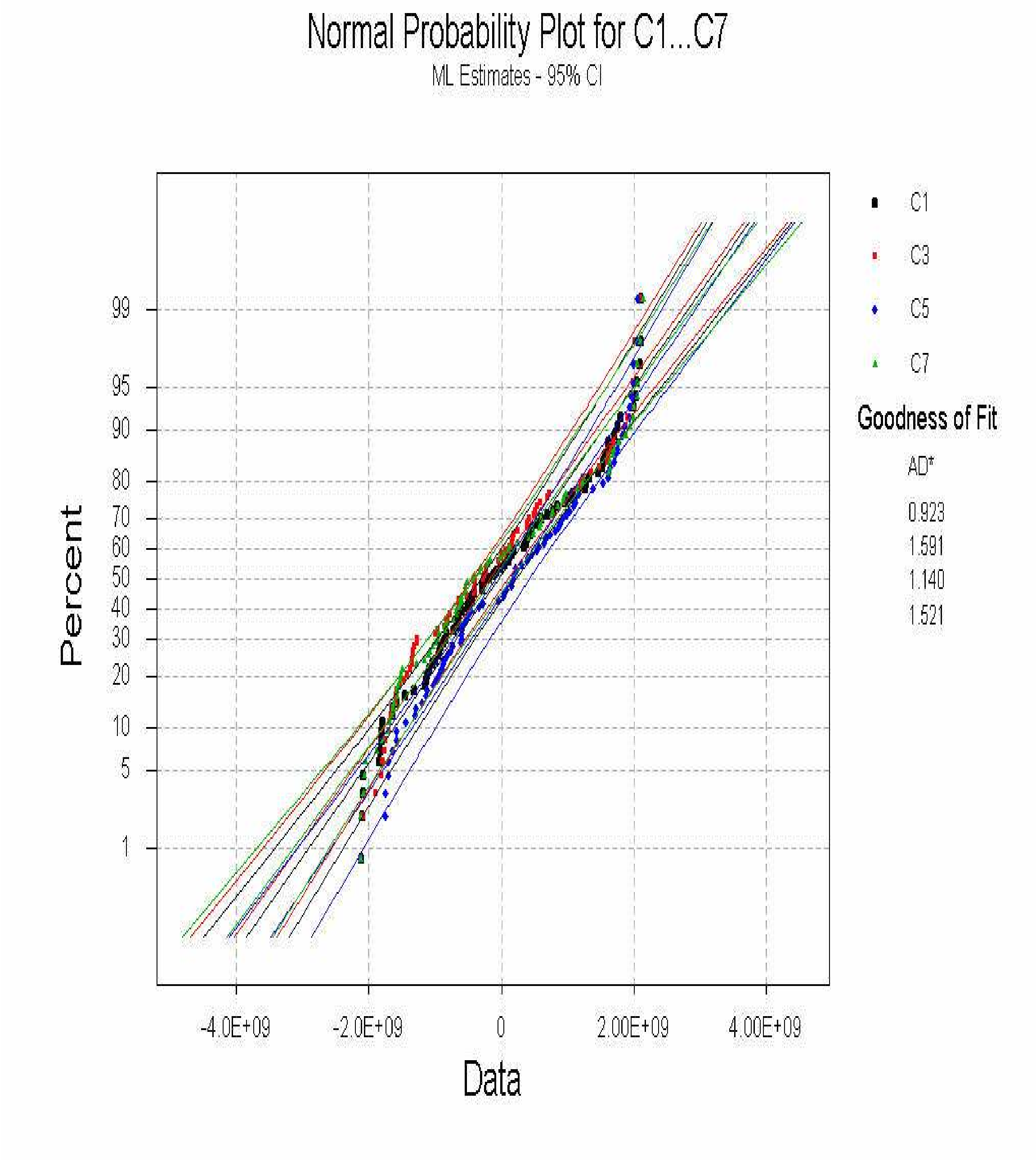


Figure 3.14. Normal probability plot for columns C1, C3, C5, C7.

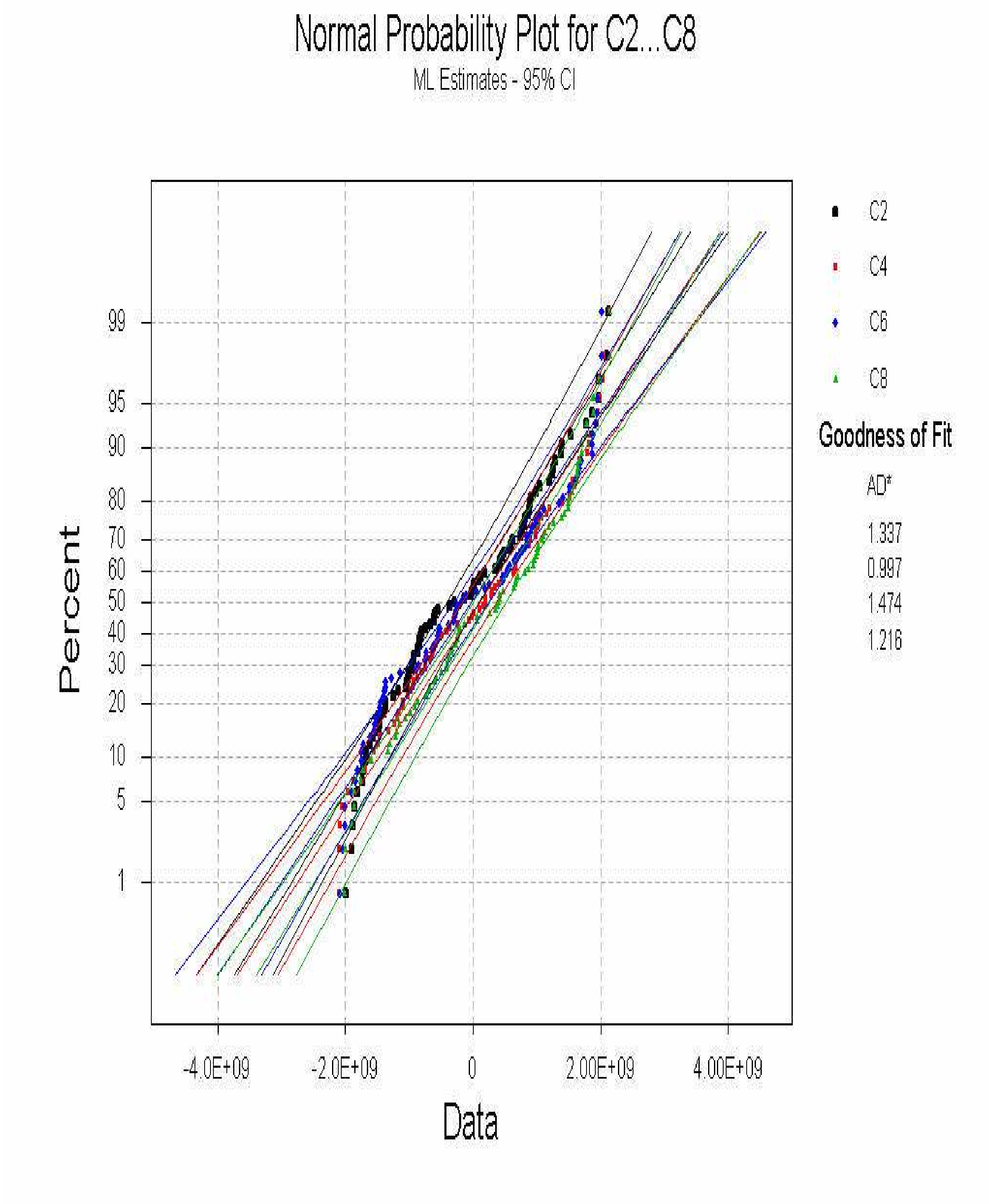


Figure 3.15. Normal probability plot for columns C2, C4, C6, C8.

3.11 Observations

This research found the encryption of cipher texts with very few rounds (less than six) to be weak. Encryption of cipher texts with more than six rounds produced a very good mixture of intermediate values and showed high resistance to cryptanalytic attacks. With the few exceptions mentioned in section 3.8, this research concludes TEA as a best fit cryptographic algorithm for small devices where memory and power are primary concern.

CHAPTER 4

SUMMARY AND CONCLUSIONS

4.1 Contributions of the Thesis:

- This research presented a comprehensive in-depth perceptive of TEA, XTEA, and Block TEA.
- This research implemented the TEA, XTEA, and Block TEA.
- This research analyzed the published attacks, weakness of TEA, XTEA, and Block TEA.
- This research showed that Block TEA has 2^{127} equivalence classes and not 2^{128} equivalence classes.

4.2 Future Work

We feel that cryptanalysis of TEA like feistel ciphers is far from complete task. In this section we present several directions for future research.

- Finding more equivalent keys (if exists) for TEA.
- Extending the results in this thesis to study other feistel ciphers
- Investigation of key space for other fiestel ciphers.

REFERENCES

Biham, E., & Shamir, A. (1992). *Differential cryptanalysis of the data encryption Standard*. New York: Springer-Verlag.

Feistel, H. (1973, May). “Cryptography and computer privacy.” *Scientific American, 228.*

Kelsey, J., Schneier, B., & Wagner, D. (1997). Related key cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA. *In Information and Communications Security—Proceedings of ICICS,* 1334.

Mirza, F. (1998). *Block ciphers and cryptanalysis.* England: Royal Holloway University of London, Department of Mathematics.

Moon, D., Hwang, K., Lee, W., Lee, S., & Lim, J. (2002). Impossible differential cryptanalysis of reduced round XTEA and TEA. *In Fast Software Encryption – Proceedings of the 9th International Workshop.*

Schneier,B. (1996). *Applied cryptography* (2nd ed.). New York: John Wiley & Sons.

Wheeler, D.J., & Needham, R.J. (1994). TEA, a tiny encryption algorithm. *In Fast Software Encryption – Proceedings of the 2nd International Workshop*,1008.

Wheeler, D.J., & Needham, R.J. (1997). *Tea extensions.* Unpublished manuscript, Computer Laboratory, Cambridge University, England.

Wheeler, D.J., & Needham, R.J. (1998). *Correction of xtea.* Unpublished manuscript, Computer Laboratory, Cambridge University, England.

APPENDIX

IMPLEMENTATION

The implementation of TEA used for this research is shown below. Tea.c takes the 64-bit plain text input in array v and 128-bit key in array k and give the cipher text as output on execution.

```
Tea.c

#include <stdio.h>
void main (int artgc, char **argv, char **env)
{
unsigned long v[2]; /* Plaintext */
unsigned long k[4]; /* Key       */
unsigned long w[2]; /* cipher text */

/* Input to Plain Text */
v[0] = 0x12345678; v[1] = 0x33333333;
printf("\n\n\nInput Data: ");
printf (" v = 0x%x  %x\n\n", v[0], v[1]);

/* Key */
k[0] = 0x00000000; k[1] = 0x80000000;
k[2] = 0x80000000; k[3] = 0x80000000;
printf ("Key = 0x%x %x %x %x\n\n", k[0], k[1], k[2], k[3]);

/* Now call Encode Routine */
   tea_code (v, k);
   printf ("\nEncoded data = 0x%x %x\n\n", v[0], v[1]);

/* Now call Decode Routine */
   tea_decode (v, k);
```

Tea.c

```
  printf ("\nDecoded data = 0x%x %x\n", v[0], v[1]);
  }

tea_code(long* v, long* k)
 {
 /* long is 4 bytes. */
  unsigned long v0=v[0], v1=v[1];
  unsigned long  k0=k[0], k1=k[1], k2=k[2], k3=k[3];
  unsigned long sum=0;
  unsigned long delta = 0x9e3779b9, n=32 ;
  while (n-- > 0) {
    sum += delta ;
    v0 += (v1<<4)+k0 ^ v1+sum ^ (v1>>5)+k1 ;
    v1 += (v0<<4)+k2 ^ v0+sum ^ (v0>>5)+k3 ;
  }
  v[0]=v0 ;
  v[1]=v1 ;
}


tea_decode(long* v, long* k)
 {
  unsigned long v0=v[0], v1=v[1];
  unsigned long k0=k[0], k1=k[1], k2=k[2], k3=k[3];
  unsigned long n=32, sum, delta=0x9e3779b9 ;
  sum=delta<<5 ;

  while (n-- > 0) {
    v1 -= (v0<<4)+k2 ^ v0+sum ^ (v0>>5)+k3 ;
    v0 -= (v1<<4)+k0 ^ v1+sum ^ (v1>>5)+k1 ;
    sum -= delta ;
  }
  v[0]=v0 ;
  v[1]=v1 ;
}
```

The implementation of XTEA used for this research is shown below. Xtea.c takes the 64-bit plain text input in array v and 128-bit key in array k and give the cipher text as output on execution.

```
Xtea.c

 #include <stdio.h>
 void main (int artgc, char **argv, char **env)
 {
 unsigned long v[2]; /* Plaintext */
 unsigned long k[4]; /* Key       */
 unsigned long w[2]; /* cipher text */

 /* Input to Plain Text */
 v[0] = 0x12345678; v[1] = 0x33333333;
 printf("\n\n\nInput Data: ");
 printf (" v = 0x%x  %x\n\n", v[0], v[1]);

 /* Key */
 k[0] = 0x00000000; k[1] = 0x80000000;
 k[2] = 0x00000000; k[3] = 0x00000000;
 printf ("Key = 0x%x %x %x %x\n\n", k[0], k[1], k[2], k[3]);

 tea_code (v, k);
 printf ("\nEncoded data = 0x%x %x  \n\n", v[0], v[1]);

 /* Now call Decode Routine */
 tea_decode (v, k);
 printf ("\nDecoded data = 0x%x %x \n", v[0], v[1]);}

 tea_code(long* v, long* k)
 {
   /* long is 4 bytes. */
 unsigned long v0=v[0], v1=v[1];
 unsigned long k0=k[0], k1=k[1], k2=k[2], k3=k[3];
 unsigned long sum=0, delta=0x9e3779b9;
 int n =32;
 unsigned long limit=delta*n;
   while (sum!=limit)
```

Xtea.c

```
  {
  v0   +=  (v1<<4 ^ v1>>5) + v1 ^ sum + k[sum&3];
  sum  +=  delta ;
  v1   +=  (v0<<4 ^ v0>>5) + v0 ^ sum + k[sum>>11 &3];
    }
   v[0]=v0 ;
   v[1]=v1 ;
   }

tea_decode(long* v, long* k)
 {
 unsigned long v0=v[0], v1=v[1];
 unsigned long k0=k[0], k1=k[1], k2=k[2], k3=k[3];
 unsigned long sum = 0xc6ef3720;
 unsigned long delta=0x9e3779b9 ;
 int count =32;
  while (count) {
  v1   -=  (v0<<4 ^ v0>>5) + v0 ^ sum + k[sum>>11 &3];
  sum  -=  delta ;
  v0   -=  (v1<<4 ^ v1>>5) + v1 ^ sum + k[sum&3];
    count = count-1;
    }
   v[0]=v0 ;
   v[1]=v1;
 }
```

In the process of extracting similarities between the intermediate values of the cipher texts during the 2nd ,4th , 8th ,16th rounds of the encryption process of Xtea, the program Xtea.java was developed in Java language using the oracle 9*i* database to store the intermediate results.

The Program Xtea.java is presented below.

```
Xtea.java

  import java.lang.*;
  import java.math.*;
  import java.io.*;
  import java.util.*;
  import javax.servlet.*;
  import java.sql.*;
  import javax.servlet.http.*;

  public class wsrg
  extends GenericServlet {

   public void service(ServletRequest request, ServletResponse response)
    throws ServletException, IOException {

     // Get print writer
     PrintWriter pw = response.getWriter();

     // Get enumeration of parameter names
     Enumeration e = request.getParameterNames();

     // Display parameter names and values
       String pname1 = (String)e.nextElement();
       pw.print(pname1 + " = ");
       String pvalue1 = request.getParameter(pname1);
       pw.println(pvalue1+"\n");
       String pname2 = (String)e.nextElement();
       pw.print(pname2 + " = ");
       String pvalue2 = request.getParameter(pname2);
       pw.println(pvalue2);
       String pname3 = (String)e.nextElement();
       pw.print(pname3 + " = ");
       String pvalue3 = request.getParameter(pname3);
       pw.println(pvalue3);
       String pname4 = (String)e.nextElement();
       pw.print(pname4 + " = ");
       String pvalue4 = request.getParameter(pname4);
       pw.println(pvalue4);
       String pname5 = (String)e.nextElement();
       pw.print(pname5 + " = ");
       String pvalue5 = request.getParameter(pname5);
       pw.println(pvalue5);
```

```
      String pname6 = (String)e.nextElement();
      pw.print(pname6 + " = ");
      String pvalue6 = request.getParameter(pname6);
      pw.println(pvalue6);
      pw.println("display");

     /* String input,seed;

     input=pvalue3;
     if (pvalue3=="") input=pvalue6;
     else input=pvalue3;
     if (pvalue4=="") seed=pvalue5;
     else seed=pvalue4;
     input=input + seed; */
     Xtea a=new Xtea( );
    String stri = a.nmain(pvalue3,pvalue5);
     pw.println(stri);
     pw.close();
  }
}

class Xtea
{
        private int _key[];
        private byte _keyBytes[];
        private int _padding;

        int w[] = new int[2];
        static String src1;
        static String keyString1;
        protected static final char hex[] =
       { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };


        public static String nmain(String args,String seed)
        {

        String keyString = "80000000000000008000000080000000";
        keyString1= keyString;
        byte key[] = new BigInteger(keyString, 16).toByteArray();
        Xtea t = new Xtea(key);
        String src = args;
        src1=src;
        src = t.padPlaintext(src);
        byte plainSource[] = src.getBytes();
                int enc[] = t.encode(plainSource, plainSource.length);
                String hexStr = t.binToHex(enc);
                return(hexStr);

        }


        public Xtea(){}

        public Xtea(byte key[])
        {
                int klen = key.length;
                _key = new int[4];
                if (klen != 16) throw new
ArrayIndexOutOfBoundsException(this.getClass().getName() +
": Key is not 16 bytes");

                int j, i;
                for (i = 0, j = 0; j < klen; j += 4, i++)
                key[i] = (key[j] << 24 ) | (((key[j+1])&0xff) << 16) |
(((key[j+2])&0xff) << 8) | ((key[j+3])&0xff);
                _keyBytes = key;
        }
        public Xtea(int key[])
        {
                _key = key;
        }
        public String toString()
        {
                String tea = this.getClass().getName();
                tea +=  ": Extended Tiny Encryption Algorithm (XTEA)
   key: " + getHex(_keyBytes);
                return tea;
        }

        public int [] encipher(int v[])
        {
```

```
                        int y= v[0];
                        int z= v[1];
                        int sum=0;
                        int delta=0x9E3779B9;
                        String sttr[]=new String[33];
                        int n=32;
    try
      {
     Connection con = null;
      Statement stmt = null;
      ResultSet rs = null;
        Class.forName ("oracle.jdbc.driver.OracleDriver");
         con = DriverManager.getConnection
("jdbc:oracle:thin:@mdb.cs.ua.edu:1521:wdb", "reddy001", "Stanf0rd96");

                        while(n-->0)
                        {
                                y += (z << 4 ^ z >>> 5) + z ^ sum + _key[(int)
    (sum&3)];
                                sum += delta;
                                z += (y << 4 ^ y >>> 5) + y ^ sum + _key[(int)
   (sum >>>11) & 3];

                                int acd[]= new int[2];
                                acd[0] = (int)y;
                                acd[1] = (int)z;
                                int m= 32-n;
                                sttr[m]=binToHex(acd);

                        }

                        w[0] = (int)y;
                        w[1] = (int)z;
                        stmt = con.createStatement();

              rs = stmt.executeQuery("insert into finaldata values ('" + src1
   +"','"+keyString1+"','"+sttr[1]+"','"+sttr[3]+"','"+sttr[7]+"','"+sttr[15]
   +"','"+sttr[31]+"')");
              con.close();
               }
```

```
 catch(ClassNotFoundException e)
            {
                 System.out.println("Couldn't load database driver: " +
e.getMessage());
            }
    catch(SQLException e)
            {
                  System.out.println("SQLException caught: " + e.getMessage());
            }
                        return w;
            }


            public int [] decipher(int v[])
            {
                        int y=v[0];
                        int z=v[1];
                        int sum=0xC6EF3720;
                        int delta=0x9E3779B9;
                        int n=32;
                while (n-->0)
                        {
                        z -= (y << 4 ^ y >>> 5) + y ^ sum + _key[(sum>>>11) & 3];
                        sum -= delta;
                        y -= (z << 4 ^ z >>> 5) + z ^ sum + _key[sum &3];
                        }
                        int w[] = new int[2];
                        w[0] = (int)y;
                        w[1] = (int)z;
                        return w;
            }

            public int [] encode(byte b[], int count)
            {
                        int j ,i;
                        int bLen = count;
                        byte bp[] = b;
                        _padding = bLen % 8;
                        if (_padding != 0)
                        {
```

```
                        _padding = 8 - (bLen % 8);
                        bp = new byte[bLen + _padding];
                        System.arraycopy(b, 0, bp, 0, bLen);
                        bLen = bp.length;
                }

                int intCount = bLen / 4;
                int r[] = new int[2];
                int out[] = new int[intCount];
                for (i = 0, j = 0; j < bLen; j += 8, i += 2)
                {
                r[0] = (bp[j] << 24 ) | (((bp[j+1])&0xff) << 16) |
 (((bp[j+2])&0xff) << 8) | ((bp[j+3])&0xff);
                r[1] = (bp[j+4] << 24 ) | (((bp[j+5])&0xff) << 16) |
(((bp[j+6])&0xff) << 8) | ((bp[j+7])&0xff);
                        r = encipher(r);
                        out[i] = r[0];
                        out[i+1] = r[1];
                        break;
                }
                return out;
        }

     public int padding()
        {
                return _padding;
        }

    public byte [] decode(byte b[], int count)
        {
                int i, j;
                int intCount = count / 4;
                int ini[] = new int[intCount];
                for (i = 0, j = 0; i < intCount; i += 2, j += 8)
                {
                ini[i] = (b[j] << 24 ) | (((b[j+1])&0xff) << 16) |
 (((b[j+2])&0xff) << 8) | ((b[j+3])&0xff);
                ini[i+1] = (b[j+4] << 24 ) |
 (((b[j+5])&0xff) << 16) | (((b[j+6])&0xff) << 8) | ((b[j+7])&0xff);
                }
        return decode(ini);
        }

        public byte [] decode(int b[])
        {
                int intCount = b.length;
                byte outb[] = new byte[intCount * 4];
                int tmp[] = new int[2];
                int i, j;
                for (j = 0, i = 0; i < intCount; i += 2, j += 8)
                {
                        tmp[0] = b[i];
                        tmp[1] = b[i+1];
                        tmp = decipher(tmp);
                        outb[j]   = (byte)(tmp[0] >>> 24);
                        outb[j+1] = (byte)(tmp[0] >>> 16);
                        outb[j+2] = (byte)(tmp[0] >>> 8);
                        outb[j+3] = (byte)(tmp[0]);
                        outb[j+4] = (byte)(tmp[1] >>> 24);
                        outb[j+5] = (byte)(tmp[1] >>> 16);
                        outb[j+6] = (byte)(tmp[1] >>> 8);
                        outb[j+7] = (byte)(tmp[1]);
                }
                return outb;
        }

        public int [] hexToBin(String hexStr) throws
ArrayIndexOutOfBoundsException
        {
                int hexStrLen = hexStr.length();
                if ((hexStrLen % 8) != 0)
                throw new ArrayIndexOutOfBoundsException
("Hex string has incorrect length, required to be divisible by eight:
 " + hexStrLen);

                int outLen = hexStrLen / 8;
                int out[] = new int[outLen];
                byte nibble[] = new byte[2];
            byte b[] = new byte[4];
                int posn = 0;
                for (int i = 0; i <outLen; i++)
                {
                        for (int j = 0; j < 4; j++)
                        {
                                for (int k = 0; k < 2; k++)
                                {
                                switch (hexStr.charAt(posn++))
                                {
                                case '0': nibble[k] = (byte)0; break;
                                case '1': nibble[k] = (byte)1; break;
                                case '2': nibble[k] = (byte)2; break;
                                case '3': nibble[k] = (byte)3; break;
                                case '4': nibble[k] = (byte)4; break;
                                case '5': nibble[k] = (byte)5; break;
                                case '6': nibble[k] = (byte)6; break;
                                case '7': nibble[k] = (byte)7; break;
                                case '8': nibble[k] = (byte)8; break;
                                case '9': nibble[k] = (byte)9; break;
                                case 'A': nibble[k] = (byte)0xA; break;
                                case 'B': nibble[k] = (byte)0xB; break;
                                case 'C': nibble[k] = (byte)0xC; break;
                                case 'D': nibble[k] = (byte)0xD; break;
                                case 'E': nibble[k] = (byte)0xE; break;
                                case 'F': nibble[k] = (byte)0xF; break;
                                case 'a': nibble[k] = (byte)0xA; break;
                                case 'b': nibble[k] = (byte)0xB; break;
                                case 'c': nibble[k] = (byte)0xC; break;
                                case 'd': nibble[k] = (byte)0xD; break;
                                case 'e': nibble[k] = (byte)0xE; break;
                                case 'f': nibble[k] = (byte)0xF; break;
                                }
                                }
                                b[j] = (byte)(nibble[0] << 4 | nibble[1]);
                        }

out[i] = (b[0] << 24 ) | (((b[1])&0xff) << 16) | (((b[2])&0xff) << 8) |
((b[3])&0xff);
                }
```

```
    return out;
    }

    public String binToHex(int enc[]) throws
ArrayIndexOutOfBoundsException
    {
        if ((enc.length % 2)    == 1)
        throw new ArrayIndexOutOfBoundsException
("Odd number of ints found: " + enc.length);

        StringBuffer sb = new StringBuffer();
        byte outb[] = new byte[8];
        int tmp[] = new int[2];
        int counter = enc.length / 2;
        for (int i = 0; i < enc.length; i += 2)
        {
            outb[0]   = (byte)(enc[i] >>> 24);
            outb[1] = (byte)(enc[i] >>> 16);
            outb[2] = (byte)(enc[i] >>> 8);
            outb[3] = (byte)(enc[i]);
            outb[4] = (byte)(enc[i+1] >>> 24);
            outb[5] = (byte)(enc[i+1] >>> 16);
            outb[6] = (byte)(enc[i+1] >>> 8);
            outb[7] = (byte)(enc[i+1]);

            sb.append(getHex(outb));
        }
        return sb.toString();
    }

    public String getHex(byte b[])
    {
        StringBuffer r = new StringBuffer();
        for (int i = 0; i < b.length; i++)
        {
            int c = ((b[i]) >>> 4) & 0xf;
            r.append(hex[c]);
            c = ((int)b[i] & 0xf);
            r.append(hex[c]);
        }
```

```
return r.toString();
        }

        public String padPlaintext(String str, char pc)
        {
                StringBuffer sb = new StringBuffer(str);
                int padding = sb.length() % 8;
                for (int i = 0; i < padding; i++)
                sb.append(pc);
                return sb.toString();
        }

        public String padPlaintext(String str)
        {
                return padPlaintext(str, ' ');
        }

}
```

Btea.html is a Java Script implementation of Block TEA. In this research it is implemented using the corrected round function.

```
Btea.html

  <html>
    <head>
      <title>TEA</title>
      <meta http-equiv="content-type" content="text/html;
           charset=iso-8859-1">
      <style type="text/css"> <!-- p, td { font-family: Arial, sans-serif; }
           --> </style>
      <script type="text/javascript">

  function callencode(val, key)
  {
      var v = strToLongs(escape(val).replace(/%20/g,' '));
      var k = keyToLongs(key);
      var n = v.length;
      if (n == 0) return("");
      if (n == 1) v[n++]=0;
      var z = v[n-1], y = v[0], delta = 0x9E3779B9;
      var mx, e, q = Math.floor(6 + 52/n), sum = 0;
      while (q-- > 0) {
          sum += delta;
          e = sum>>>2 & 3;
          for (var p = 0; p < n-1; p++) {
              y = v[p+1];
        mx = (z>>>5 ^ y<<2) + (y>>>3 ^ z<<4) ^ (sum^y) + (k[p&3 ^ e] ^ z)
              z = v[p] += mx;
          }
          y = v[0];
          mx = (z>>>5 ^ y<<2) + (y>>>3 ^ z<<4) ^ (sum^y) + (k[p&3 ^ e] ^ z)
          z = v[n-1] += mx;
      }
      return(longsToStr(v));
  }
```

Btea.html

```
function calldecode(val, key)
{
   var v = strToLongs(val);
   var k = keyToLongs(key);
   var n = v.length;
   if (n == 0) return("");
   var z = v[n-1], y = v[0], delta = 0x9E3779B9;
   var mx, e, q = Math.floor(6 + 52/n), sum = q*delta;
   while (sum != 0) {
      e = sum>>>2 & 3;
      for (var p = n-1; p > 0; p--) {
         z = v[p-1];
         mx = (z>>>5 ^ y<<2) + (y>>>3 ^ z<<4) ^ (sum^y) + (k[p&3 ^ e] ^ z)
         y = v[p] -= mx;
      }
      z = v[n-1];
      mx = (z>>>5 ^ y<<2) + (y>>>3 ^ z<<4) ^ (sum^y) + (k[p&3 ^ e] ^ z)
      y = v[0] -= mx;
      sum -= delta;
   }

   var s = longsToStr(v);
   if (s.indexOf("\x00") != -1) {
    s = s.substr(0, s.indexOf("\x00"));
   }
   return(unescape(s));
}

function strToLongs(s) {
   var l = new Array(Math.ceil(s.length/4))
   for (var i=0; i<l.length; i++) {
      l[i] = s.charCodeAt(i*4) + (s.charCodeAt(i*4+1)<<8) +
           (s.charCodeAt(i*4+2)<<16) + (s.charCodeAt(i*4+3)<<24);
   }
   return(l);
}


function keyToLongs(k) {
   var l = new Array(4)
```

```
    for (var i=0; i<4; i++) {
       l[i] = k.charCodeAt(i*4) + (k.charCodeAt(i*4+1)<<8) +
        (k.charCodeAt(i*4+2)<<16) + (k.charCodeAt(i*4+3)<<24);
    }
    return(l);
}

function longsToStr(l) {
    var s = "";
    for (var i=0; i<l.length; i++) {
       s += String.fromCharCode(l[i] & 0xFF, l[i]>>>8 & 0xFF,
                         l[i]>>>16 & 0xFF, l[i]>>>24 & 0xFF);
    }
    return(s);
}


    </script>
  </head>
  <body bgcolor="#00FFFF">
    <form name="f" action="none!">
    <body bgcolor="#00FFFF">
    <script type="text/javascript">
     document.write("<h1>Block Tiny Encryption Algorithm</h1>")
     var d = new Date()
     x=d.getMinutes()
     y=d.getSeconds()
     h=d.getHours()
     no=Math.random()*1000000000000*x*y*h
     document.write("You can use this sample generated random number as
 a key: ");
     document.write(Math.floor(no))
     </script>
<p> &nbsp;</p>
   <p>
   </p>
     <p>Possible Equivalent key pairs to prove the key space is 127 instead of
     128 (for Block TEA) <br>
     <br>
     Pair 1:<br>
     00000000800000000000000000000000 <br>
     00000000800000008000000080000000 <br>
```

```
<br>
    Pair 2:<br>
    80000000000000000000000000000000 <br>
    80000000000000008000000080000000 <br>
_____________________________________________________________________
______________________</p>

   </body>
     <table width="701" style="border-collapse: collapse" bordercolor=
"#111111" cellpadding="0" cellspacing="0" height="200">    <tr>
        <td width="697" height="35"><b><font size="4"
face="Batang">&nbsp;</font></b></td>
        <td width="4" height="35">&nbsp;</td>  </tr>   <tr>
        <td width="697" height="41"><b><font size="4" face="Batang">
Please type the Plain Text</font></b><input type="text" name="val"
size="100" ><p>&nbsp;</p> <p><b><font size="4" face="Batang">
Please type the Key </font></b><input type="text" name="key" size="32" >
</p><p>&nbsp;</td> <td width="4" height="41">&nbsp;</td>
/tr> <tr> <td width="697" height="54"><script type="text/javascript">var
ciferText;</script>   <!-- use 'ciferText' because <input> field will truncate at
null chars --> <input type="button" value="The Cipher text is"
           onClick='f.encrypted.value = ciferText = callencode(f.val.value,
f.key.value)'>&nbsp; <input type="text" name="encrypted" size="100"></td>
       </tr><tr><td width="697" height="22">&nbsp;<p>&nbsp;</td>
       </tr> <tr><td width="697" height="26">
        <input type="button" value="Decrypted Value of the cipher text is"
                onClick='f.decrypted.value = calldecode(ciferText,
f.key.value)'>&nbsp;       <input type="text" name="decrypted" size="100">
</td>   </tr> <tr> <td width="697" height="22">&nbsp;</td>
       </tr> </table></form> </body><p>
_____________________________________________________________________
<p><br>  <br>&nbsp;</p>
</html>
```