AN INVESTIGATION OF DATA-DEPENDENT STRUCTURES IN CRYPTOGRAPHIC CIPHERS

BRIAN J. KIDNEY

AN INVESTIGATION OF DATA DEPENDENT STRUCTURES IN CRYPTOGRAPHIC CIPHERS

BY

© BRIAN J. KIDNEY

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Master of Engineering

FACULTY OF ENGINEERING AND APPLIED SCIENCE

MEMORIAL UNIVERSITY OF NEWFOUNDLAND

March 2010

St. John's Newfoundland

Abstract

This thesis studies the use of data dependent structures in cryptography. Since the introduction of RC5 by Rivest in 1994, which relied heavily on data dependent rotations for its security [1], these structures have gained interest in cryptography. During the Advanced Encryption Standard selection process two candidate ciphers, RC6 and MARS, relied on data dependent structures.

The thesis focuses on CIKS-1, a cipher introduced in the Journal of Cryptography in 2002 [2], that relies mainly on data dependent permutations for its security. Due to its reliance on these permutations, this cipher is chosen as a basis for the study of data–dependent structures in cryptographic algorithms.

The first attack on CIKS-1 presented is a chosen plaintext attack which exploits the lack of change in the Hamming weight of the data as it is enciphered. The research shows that there is a class of weak keys with low weight that can be detected when the input weight is constrained. An attack on a 6 round reduced version of the cipher is outlined that can reduce the search space of the first round subkey to within a weight of two from the weight of the actual key. This attack is experimentally shown to work when the subkey weights are around six or less with a total time complexity for the attack of 2^{52} encryption operations.

The second attack presented is a variant of classical differential cryptanalysis. Instead of focusing on the exact bit difference of the two inputs that make up the differential, the attack instead focuses on the difference in their weights. An experimental attack on a three–round reduced version of the cipher is presented using this

technique which can retrieve the last round subkey of CIKS–1 with a data complexity of approximately 2^{35} plaintext/ciphertext pairs and time complexity of approximately 2^{35} encryption operations plus 2^{68} partial decryption operations. It is also shown that this can theoretically be extended to the whole cipher with a total data complexity of 2^{51} plaintext/ciphertext pairs and time complexity of approximately 2^{52} encryption operations plus 2^{84} partial decryption operations.

Despite the weaknesses discovered in CIKS–1, there is potentially some merit in using data dependent permutations in ciphers. Therefore, the implementation of CIKS–1 in software is investigated. The cipher was originally designed to be fast in hardware and contains many operations that work at the bit level, which are inefficient to implement in software. A software version of the cipher is presented which uses bitslicing, effectively parallelizing the cipher on a single processor. This version experimentally shows a speed up of approximately 175 times over a more straight forward implementation using arrays of elements to hold individual bits.

Acknowledgments

First and foremost, I would like to thank my wife, Lori Hogan. Without her support and encouragement, this thesis would not exist. She has been my sounding board, my editor, my motivational speaker and my rock. She is the only person with whom I would ever have undertaken this journey.

I would like to thank my supervisors, Dr. Howard Heys and Dr. Theodore Norvell. Their funding allow me to start my research. Their support, guidance, teaching and immense patience have allowed me to complete my research. Along the way, they have taught me many things.

I would also like to thank my parents, Noel and Roma Kidney. They have always believed in me and pushed me to work hard, set my goals higher and never let me quit.

Finally, I would like to thank my old office mate, Andrew House. He was always there to answer a question, give an opinion or provide a needed distraction. Even after he finished his degree, he was always there to push me toward completing my own.

Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

AES Advanced Encryption Standard

CBC Cipher Block Chaining

CFB Cipher Feedback

CP Controlled Permutation

CV Control Vector

DES Data Encryption Standard

DDP Data Dependent Permutation

DDR Data Dependent Rotation

ECB Electonic Codebook

IV Initialization Vector

LHS Left Hand Side

LSB Least Significant Bit

MSB Most Significant Bit

NIST The United States National Institute of Standards and Technology

NSA United States National Security Agency

OFB Output Feedback

RHS Right Hand Side

SAC Strict Avalanche Criteria

SPN Substitution Permutation Network

STL Standard Template Library

SWIFT Society for Worldwide Interbank Financial Telecommunications

WEP Wired Equivalent Privacy

Chapter 1

Introduction

> "There is a ton of evidence both in computing and outside of it which shows that poor security can be very much worse than no security at all. In particular, stuff which makes users think they are secure but is worthless is very dangerous indeed." Alan Cox, Linux Kernel Developer

For hundreds of years, governments throughout the world have used cryptography to guard their secrets from their enemies. Julius Caesar used simple substitutions and rotations to convey messages to his troops at war. The simplest of these ciphers substituted Greek letters for Roman, making it impossible for his enemies to read it. Another, now known as the Caesar Cipher, rotated each letter three places down the alphabet (i.e. $A \rightarrow D$), thus making the message appear to be gibberish if intercepted [5].

Over time, simple ciphers such as these were realized to be easily defeated using statistical information known about the language in use, and thus to be weak. They were replaced by newer ciphers using techniques such as the wholesale replacement of alpha numeric characters with symbols or using published works as keys. One example of this was the set of encrypted messages left by Thomas Beale, an American prospector, outlining details on a large cache of gold, silver and gems he had buried. One of the messages was found to have used the United States Declaration

of Independence as a key. After numbering the words in the document, the message is revealed by replacing each number in the encoded message with the first letter in the corresponding word from the Declaration [5].

By the Second World War these types of ciphers were also being replaced, mandated by the need for more secure communication to relay military plans. The German government had started to employ the use of mechanical cipher devices such as the German Enigma machine, which created codes increasingly difficult to break by hand. This in turn led to the use of computers being employed by the code breakers, which again led to the development of more secure ciphers using computing.

With the proliferation of computers and computer networks after the war, the use of cryptography moved from primarily a government and military domain to other sectors such as banking. For example, in 2006 the Society for Worldwide Interbank Financial Telecommunications (SWIFT) handled an average of 11.4 million secure transactions per day on their network [6]. Without encryption, these transactions would be vulnerable to attacks which could induce chaos in world economies.

Today, encryption is ubiquitous in modern life. In 2005, 58% of Canadian internet users went online to do their banking electronically and 55% used it to pay bills, all secured by forms of encryption [7]. Voice over IP applications such as Skype™ now employ encryption to keep conversations private. Even operating systems such as Microsoft Windows™ and Apple OSX™ include functionality to encrypt personal files.

One byproduct of the public exposure of cryptography is the move to standardize it in public. For many years, ciphers used for standards were chosen by governments in conjunction with industry. This led to belief that these ciphers were intentionally weakened to allow the government to decipher them easily. Such was suspected of the Data Encryption Standard (DES) [8]. Although it has never been found or even proven to exist, many have suspected a trap door in this cipher that was developed

by IBM in conjunction with the United States National Security Agency (NSA), an agency of the government of the United States [8]. Today, commercial security standards such as this are no longer developed secretly. Standards such as the Advanced Encryption Standard (AES) (the replacement for DES) are now selected through public processes involving governments, industries and academics. In fact, many of the world's cryptographic ciphers are now published and undergo scrutiny from scholars worldwide to determine their strengths and weaknesses. One such cipher is CIKS 1, which was published in January 2002 [2].

The CIKS 1 cipher was proposed as a fast and secure method of encryption designed for hardware implementation. The main primitives used in the algorithm are Data Dependent Permutations (DDPs), a large set of functions that use part of the data involved in the encryption (either plaintext or key) to permute other portions of the data. These structures have appeared in other ciphers as well, including RC5 and RC6, but normally in the less general form, Data Dependent Rotations (DDRs)[1][9]. DDRs have shown to be resistant to popular cryptanalysis techniques such as linear and differential attacks [10] and the CIKS–1 authors state the same for DDPs [2].

1.1 Motivation for Research

The DDP is proposed as a component in new cryptographic algorithms. It can be implemented in hardware to achieve fast speeds [2] and a subset of these permutations has been shown to be resistant to linear and differential cryptanalysis [10]. Two popular algorithms proposed by Ron Rivest, RC5 and RC6, use a subset of the DDP functions, in the way of DDRs, as main parts of cipher [1][9].

In 2002, CIKS 1 was proposed as a new cipher that was fast in hardware and resistant to attack. The cipher uses a more general form of the DDP as its main primitive with only four other functions to create its security. Since this algorithm

relies so heavily on DDPs, it makes a good candidate for a study on the security properties of the functions. Therefore, we use this cipher as our basis for cryptographic attacks to determine the qualities of DDPs.

1.2 Scope of Work

The purpose of this thesis is to investigate the properties of DDPs as a cryptographic primitive. First, an introduction to cryptography and concepts required in later chapters is provided. Included is a look at general guidelines for secure algorithms and common cryptanalysis techniques. There is also an overview of selected ciphers with DDPs with a more in–depth look at CIKS 1.

In later chapters the CIKS 1 cipher is used as a testbed for the DDP. The properties of the DDP are studied under the use of low weight inputs, exposing the need for a well–defined key schedule. A differential attack is proposed for the cipher which exposes limitations on the CIKS 1 use of DDPs. Finally, there is a study of techniques optimizing speed when implementing DDPs in software.

1.3 Thesis Outline

The thesis progresses in the following manner:

- Chapter One: An introduction to the research conducted.
- Chapter Two: An introduction to cryptography and cryptanalysis.
- Chapter Three: An introduction to ciphers with data dependent structures, with particular focus given to the CIKS 1 cipher.
- Chapter Four: A weight based attack on the CIKS 1 cipher is proposed.
- Chapter Five: A differential attack on the CIKS–1 cipher is proposed.

- Chapter Six: A bitsliced implementation of CIKS–1 is presented as an efficient software implementation.

- Chapter Seven: A summary of results and conclusions.

1.4 Summary of Results

In Chapter 4 of this thesis, a weight based attack on the CIKS 1 cipher is proposed. The attack focuses on the limited effect of the weight of the key on the weight of the data being processed by the cipher. A class of weak keys with low Hamming weight that can be exploited to constrict the search area for actual key is presented. An attack is demonstrated on a six- round reduced version of the cipher with subkey Hamming weights limited to six or less. The attack has a total time complexity of 2^{52} encryption operations and reduces the search space for the actual key to a value with Hamming weights within two of the actual weight.

In Chapter 5 another attack is proposed which takes advantage of weight propagation in CIKS–1. This attack is a non–traditional differential attack where the differentials are differences in the Hamming weight of the plaintext inputs. An experimental verification of the attack is completed on a three–round reduced version of the cipher with a data complexity of 2^{35} plaintext/ciphertext pairs and a time complexity of 2^{68} partial decryption operations. Differentials for the attack on the full 8 round versions of the cipher are also presented which results in an attack with a data complexity of 2^{52} plaintext/ciphertext pairs and a time complexity of 2^{84} partial decryption operations.

Finally, in Chapter 6, efficient implementation for CIKS–1 in software is investigated. Since the cipher was designed for hardware, many of the primitives perform operations at the level of bits and as such do not effectively utilize word based instructions in general purpose processors. An implementation of CIKS 1 using the

bitslicing technique is presented which fully utilizes the instruction set of modern processors. The 32 bit and 64 bit versions of this implementation are compared to implementations using arrays and bitset (from the C++ STL). It is shown that the bitslice technique provides a speed up of 234 times for the 32 bit implementation and 425 times for the a 64–bit implementation over the fastest of the other versions presented.

Chapter 2

Overview of Cryptography

Cryptography is the study and process of hiding information. Generally, it is used to keep third parties from viewing sensitive data. This includes concealing stored data such as encrypting data on a computer hard drive, but more often refers to the transfer of information over an insecure communications channel. For illustration, a transfer of information model will be used in this discussion, employing the following characters: Alice, the data transmitter; Bob, the data receiver; and Oscar, the intruder.

To transfer the data securely, a cryptosystem is used. A cryptosystem is defined as a set of functions and data sets required to transmit data from one party to another, secure from interception by a third party. As presented in [11], the parts of a cryptosystem include:

- an encryption function,
- a decryption function,
- a set of possible plaintexts,
- a set of possible ciphertexts, and
- a set of possible keys or keyspace.

The encryption function is used to transform data in the set of possible plaintexts to data in the set of possible ciphertexts, based on a key from the keyspace. This function must be one to one in order for the plaintext to be restored by the decryption function [12]. Figure 2.1 shows the standard model of a general cryptosystem.

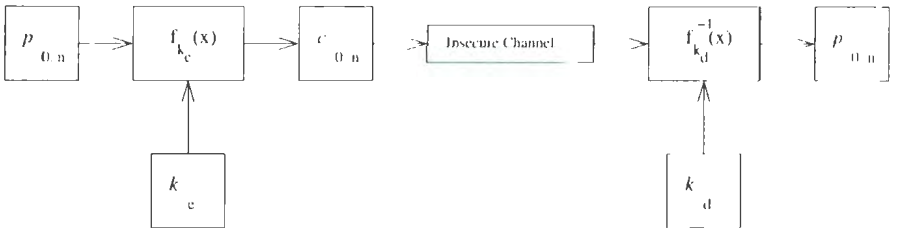

Figure 2.1: A Model of a Crytopsystem

If Alice has plaintext data $p_0, p_1, ..., p_{n-1}$ and wants to send it to Bob, but wants to make sure that only he can read it, she needs to encrypt the data. To do this she can use the encryption function f_e to produce ciphertext $c_i = f_e(p_i)$, with a key e. She can then send the resulting ciphertext data $c_0, c_1, ..., c_{n-1}$ to Bob over an insecure channel. When Bob receives the data he uses the decryption function, g_d to compute $p_i = g_d(c_i)$ with key d to reproduce the plaintext.

In an effective cryptosystem, the decryption key d (and sometimes the encryption key e) is the only piece of information that needs to be kept from Oscar in order to ensure confidentiality of the transmitted data. Therefore, Alice must have a secure mechanism to provide Bob with the key. Oscar is therefore left with trying to guess the key to decrypt the data, or trying to exploit flaws in the cryptosystem to recover the original data. This latter approach is known as cryptanalysis.

In the following sections, an introduction to cryptography is presented. There is a discussion of the different types of cryptographic functions (or ciphers) and the ways in which they are used. This is followed by a discussion of general properties for secure ciphers. Finally there is an introduction to cryptanalysis, the practice of

attacking ciphers.

2.1 Types of Cryptographic Algorithms

Cryptographic algorithms are normally classified into two groups based on the type of key which they use. When the encryption key employed in the cipher must be kept secret, it is referred to as a symmetric (sometimes called shared or secret) key cipher. When the security of the cipher does not depend on the encryption key used being kept secret, the cipher is referred to as an asymmetric (or public) key cipher. These ciphers get their names from the fact that they are designed in such a way that one key can be distributed publicly, without compromising the security of the function. Symmetric and public key ciphers are discussed in the following sections. However, symmetric key algorithms are explored more since such algorithms are the focus of this thesis.

2.1.1 Public Key Ciphers

In [13] Diffie and Hellman outlined a new method for encrypting data which uses two separate keys rather than the traditional one key. Of the two, one is to be kept private while the other is made public. They called the scheme public key cryptography.

In a public key cryptosystem as proposed by Diffie and Helman, the sender encrypts the data using the public key which is unique for each receiver. The encryption is nominally done using a mathematical computation which is easy for the sender to perform. Security is gained by choosing a problem which is intractable with the publicly known information. The receiver (of course) has extra information, the private key. This key is used in a function, commonly referred to as a trapdoor function, that allows the receiver to easily recover the original data.

The most frequenty used public key algorithm, referred to as RSA, was proposed

by Rivest, Shamir and Adleman in [14] and is presented here as an example. The security of the RSA algorithm is based on the assumption that factoring the product of two large primes is an intractable problem. To generate keys, the receiver must choose two large prime numbers p and q. From these numbers,

$$n = p \times q \tag{2.1}$$

and

$$\phi(n) = (p-1)(q-1) \tag{2.2}$$

are generated. Then another number b is chosen such that $1 < b < \phi(n)$ and the greatest common divisor of b and $\phi(n)$ is 1. The final part of the key a is calculated such that

$$a = b^{-1} \bmod \phi(n). \tag{2.3}$$

The public key consists of n and b, while the private key consists of p, q and a.

To encrypt data the sender must convert it to an integer form between 0 and $n-1$. If the message is large, it will need to be broken into multiple integers. Once in this form, the data x is encrypted using the function

$$f_{(n,b)}(x) = x^b \bmod n. \tag{2.4}$$

Once received, the ciphertext can be decrypted using the inverse trapdoor function

$$g_{(n,a)}(y) = y^a \, mod \, n. \tag{2.5}$$

The security of RSA is based on the widely held belief that given a choice of large enough primes, knowing $f_{(n,b)}(x)$, b and n solving for x presents a computationally

infeasible problem of factoring large numbers that are a product of two primes. Knowing either a or the primes p and q so that a can be computed, allows the receiver to reverse the encryption using the trapdoor function $g_{(n,a)}(y)$. This leaves an attacker with the most obvious attack possibility of factoring n into its two prime factors p and q so that a can be computed. However, prime factorization is considered to be computationally difficult problem for large n (e.g. n of a few hundred bits).

Other popular public key ciphers include those based on the Discrete Logarithm Problem. The algorithms are based on logarithms in finite groups which are one-way functions such as the ordinary logarithm found in RSA. One such cipher is the ElGamal cryptosystem [11].

2.1.2 Symmetric Key Ciphers

Due to the mathematics involved in public key cryptosystems, the ciphers are inherently slow in encryption speed. In applications where speed is a priority, symmetric key ciphers are normally employed. In some applications the two are combined, using public key cryptography to transmit a secret key that can then be used by symmetric key ciphers.

Symmetric key ciphers, also called shared or secret key, come in two varieties stream ciphers and block ciphers. The following sections outline the two types, giving more weight to block ciphers since they are studied closely in the chapters that follow.

Stream Ciphers

Stream ciphers are designed to encipher data with minimal delay. To do this, they operate on single symbols at a time, thus eliminating the requirement to wait for data to build in a queue. The primary component of a stream cipher is a keystream generator, the purpose of which is to generate a continuous stream of pseudorandom bits. This stream of bits is know as the keystream and is combined with the plaintext

to produce the ciphertext. The combining operator can vary from cipher to cipher; however it is most commonly a simple modulo two addition or bitwise exclusive or.

The decryption of the data is achieved by running the ciphertext through the same cipher to obtain the plaintext. In order for this to work, the sender and receiver must not only have access to the same key, they must also ensure that the keystream generation is synchronized on both ends of the communication. Figure 2.2 shows the basic stream cipher model.

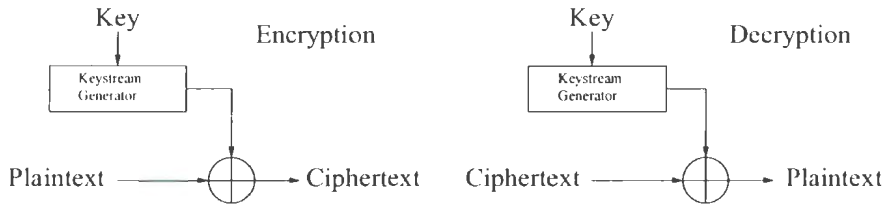

Figure 2.2: Stream Cipher Model

Stream ciphers are commonly employed in communication networks. The protocols used in these networks commonly require limited amounts of buffering to reduce the amount of latency. Example stream ciphers found in common networks are the E_0 cipher, used to secure communications between Bluetooth devices, and the Wireless Equivalent Protocol, or WEP, found in IEEE 802.11b wireless networks [15][16].

Block Ciphers

Block ciphers are symmetric (or shared) key cryptographic algorithms that encipher a fixed length of data, or block, at a time. They are typically product ciphers, meaning they have a simple function known as a round, and the final ciphertext is a result of passing the data through many rounds. Each round uses a different key known as a subkey; therefore, there is a requirement to be able to derive these keys from the shared key. The process used to derive the subkeys is known as a key schedule.

Block ciphers are designed to take advantage of the word length of the hardware on

which they will be used. If designing a block cipher for a modern personal computer, it would most likely be designed with 32 bit or 64 bit words to take full advantage of the instruction set provided by the processor.

The Substitution Permutation Network (SPN) is commonly used as an example of a block cipher. Presented by Feistel in [17], the SPN is a simple cipher consisting only of substitutions and permutations, as its name suggests. The components of an SPN are quite similar to that of DES and AES, but the algorithm is simpler, making it a good learning tool. The simple example presented in Figure 2.3 is used by Heys in [18], in a tutorial to illustrate cryptanalysis techniques.

As can be seen in Figure 2.3, the algorithm takes as input a fixed data block of 16 bits. The first three rounds use an identical round structure: add the subkey (in this case by simple modulo two addition); perform a substitution for each sub block of four bits comprising the data; and permute the result. The fourth round does not include the permutation, but an additional subkey is added following the substitution. This is done to prevent an attacker from ignoring the final substitution since without the added key it could easily be reversed.

The Advanced Encryption Standard

From 1997 to 2001, The United States National Institute of Standards and Technology (NIST) held a competition to replace the DES. The winner was to be chosen based on security, cost and characteristics of the cipher [19]. Security was the most important criterion, and as such, any cipher that showed vulnerability during the competition was eliminated. The other two criteria were then used to differentiate the remaining candidates. The cost criterion looked at the complexity of each algorithm with respect to both speed and memory. Finally the characteristics of each cipher were compared, including such qualities as flexibility and simplicity of the cipher.

In the end, the *Rijndael* cipher by Daemen and Rijmen was chosen as the new

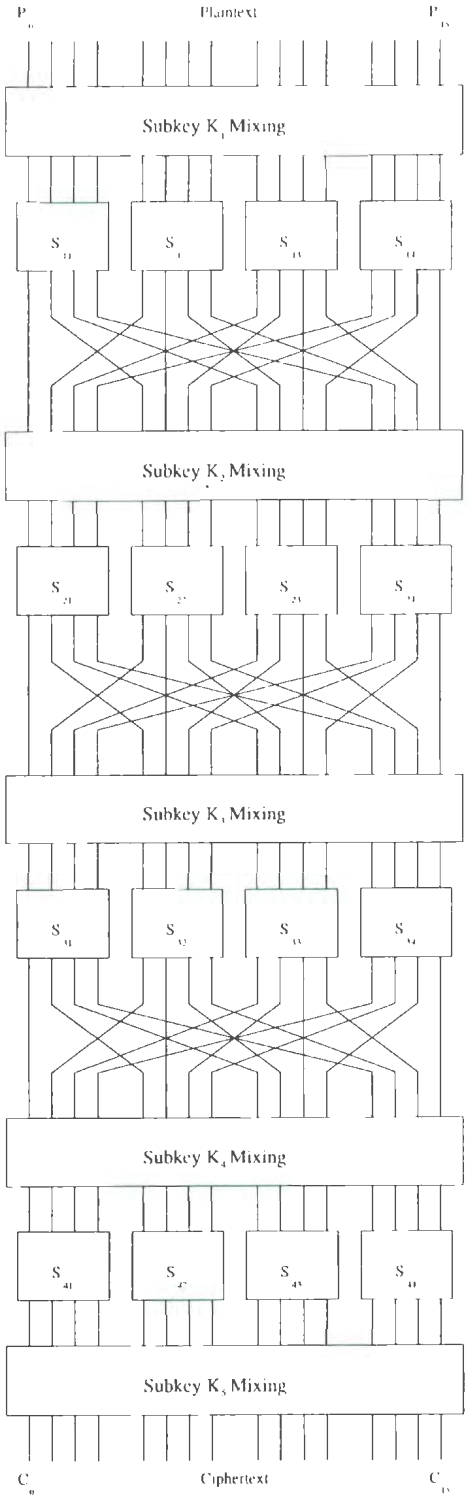

Figure 2.3: A SPN Cipher

AES. The cipher is an iterated cipher with varying key length depending on the number of rounds employed. Table 2.1 shows the key size for the three round lengths defined for the cipher. All operations used in the cipher are word oriented.

Number of Rounds	Key Size
10	128 bits
12	192 bits
14	256 bits

Table 2.1: Number of Rounds Versus Key Size for AES

AES has a structure quite similar to the SPN presented in the previous section. There is an initial whitening of the data by mixing the first round subkey with the plaintext via an exclusive or operation. Then the next r 1 rounds involve a substitution, linear transformation and subkey mixing just as with the SPN. The final round excludes the column mixing portion of the linear transformation.

The substitution used in AES is an 8 by 8 substitution box, or S Box. For each byte in the cipher data, the substitution replaces the data based on a conceptual 256–by–8 bit lookup table. These values can also be calculated using finite field mathematics.

The linear transformation that is performed next is a combination of a simple shifting of bytes and column mixing. The data in the AES is held in a variable called *State* which is a 4–by–4 byte matrix defined as

$$State = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \tag{2.6}$$

where $s_{m,n}$ is the state byte s in row m, column n. The shifting rotates each row m bytes to the left with wrapping. To mix the columns of data in the state variable

each column of data is multiplied by a column of data in a finite field $\mathbb{F}_{2^8}$. The result of the multiplication is used to replace the original data of the column. Finally, for each round there is subkey mixing which is done via an exclusive or as was the case with the whitening [11].

Modes of Operation

In [20], the U.S. National Bureau of Standards (now NIST) introduced four modes of operation for block ciphers: Electonic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB) and Output Feedback (OFB), under which DES could be run. These modes specify how the cipher could be used in various scenarios, and though intended originally for use with DES, they are commonly used with many current block ciphers.

ECB is the native mode for all block ciphers. It is defined as the most simple use of the cipher, encrypting one block at a time, independent of all other blocks. Due to this independence, the sender is guaranteed that an error in transmission in a single block will not propagate to others. ECB mode works straightforwardly for fixed length data that is a multiple of the block size. If the data does not completely fill multiple blocks, padding is required in the final block. When using this mode, the sender should be aware that repeating blocks of data using the same key will result in a repeat in ciphertext at the output, which could be exploited for an attack.

The next mode defined in [20] is CBC mode. In this mode, each output block is dependent on the last with an Initialization Vector (IV) being used to produce the first round output block. To use this mode, the sender first selects an IV, c_0. The data is then encrypted using

$$c_i = f_k(p_i \oplus c_{i-1}) \tag{2.7}$$

where $i \geq 1$ and f_k is the encryption algorithm in use with key k. As can be seen, each

plaintext block p_i is added modulo two with the previous ciphertext block c_{i-1} before being encrypted (with the exception of the first block, which is added modulo–two with the IV). The inverse of this operation is

$$p_i = g_k(c_i) \oplus c_{i-1} \tag{2.8}$$

where g_k is the decryption algorithm, using the same key k.

The advantage this mode has over ECB is that identical inputs do not result in identical output when using the same key. However, this mode still requires padding when the data size is not a multiple of the block size. As well, the introduction of dependence on previous ciphertexts comes with error propagation. An error in c_i will result in errors in both p_{i+1} and p_i. Also, the loss of a single block of ciphertext (c_i) in transmission results in a loss of two plaintext blocks in decryption.

To overcome the limitation of requiring the plaintext size to be a multiple of the cipher block size, a mode that converts a block cipher into a stream cipher can be used. Both CFB and OFB accomplish this task. When using these modes the encryption and decryption operations need to be synchronized to ensure the plaintext at both ends of the cryptosystem match.

In CFB mode, m bits are encrypted at a time where $m \leq n$, the number of bits in a block. Again, the CFB uses an IV, x_0. Once chosen, the plaintext blocks are encrypted using the operation

$$c_i = p_i \oplus MSB_m(f_k(x_i)) \tag{2.9}$$

where MSB_m is the m most significant bits of the encryption of the input variable; $f_k(x_i)$ and p_i and c_i represent m bits of plaintext and ciphertext respectively. This mode also requires the calculation of the next block cipher input, x_{i+1} using

$$x_{i+1} = LSB_{n-m}(x_i)||c_i \tag{2.10}$$

where LSB_{n-m} is the $(n-m)$ least significant bits of x_i and "||" is a concatenation operation. Decryption is a similar operation to encryption.

CFB can be used to reduce the padding requirement, however it comes with a cost in error propagation. Depending on the location of the error in the ciphertext, a single bit error in transmission can create errors in as many as $(\frac{m}{n} + 1)$ plaintext blocks. After the error has propagated through, CFB will start producing the correct plaintext again at the receiver, or self synchronize. The result of losing a block in CFB is equivalent to that in CBC.

The last mode outlined in [20] is OFB mode. This mode is similar to CFB mode in that it transforms a block cipher to a stream cipher; however, instead of feeding back ciphertext, keystream bits are fed back. Again, an IV x_0 is chosen and the ciphertext of m bits, $1 \leq m \leq n$, is created using

$$c_i = p_i \oplus MSB_m(f_k(x_i)), \tag{2.11}$$

the same operation as in CFB. However the next value of input is calculated as

$$x_{i+1} = LSB_{n-m}(x_i)||MSB_m(f_k(x_i)) \tag{2.12}$$

thus depending only on the previous values of the keystream. This has the advantage that an error in transmission of the ciphertext will not propagate to the decryption of other plaintext bits. However, a lost block of ciphertext results in loss of synchronization and continual error in the recovered plaintext.

Even though CFB and OFB can be used to convert a block cipher into a stream cipher, it should be noted that this does not automatically overcome the latency problems with block ciphers. This is due to the fact that the block cipher usually

functions on larger data inputs.

Now that the various types of ciphers and their operation have been introduced, the next section looks at properties considered desirable for secure ciphers. After this, an introduction to cryptanalysis is presented.

2.2 Properties of Secure Block Ciphers

In the body of literature for cryptography research there are many theories presented for what makes a secure block cipher. For modern cryptography, one of the first works to approach the topic was [12] by Shannon. In this paper, Shannon approaches the study of cryptosystems from an information theory point of view. In doing so, he presents properties that are desirable to make a cipher secure. Since then, others have added to this list, and a subset of these properties is presented here.

It should be noted that though these properties can be used as a guide in design, in [21] Knudsen warns against overvaluing them. He notes that Shannon's principle to ensure that a cipher is secure against all known attacks is still considered the best design principle for ciphers.

2.2.1 Confusion and Diffusion

Confusion and diffusion are related properties presented by Shannon in [12]. Confusion is a relationship between the key and the ciphertext. The goal is to make the relationship as complex as possible. If the relationship is simple (for example a simple rotation of symbols such as a Caesar Cipher), the attacker can use simple statistical analysis, based on knowledge of the plaintext language, to break the cipher. However, as more complexity (or confusion) is added to the relationship, the statistical analysis required by the attacker grows in complexity as well, with the ultimate goal of being infeasible.

Diffusion is the concept of hiding any redundancy of input bits from being discovered in the output bits. The goal is to have each bit of plaintext influence as much of the ciphertext as possible. This makes it more difficult for the attacker to detect any statistical relationship between the plaintext and ciphertext.

2.2.2 Spurious Keys and Unicity Distance

Spurious keys are keys used in decrypting ciphertext that lead to a plaintext that has meaning, but is not actually the plaintext. If an attacker has only the ciphertext and through trying multiple keys comes across two keys in which the plaintext result is meaningful, without further information it would be impossible to determine the intended message.

Using this concept, Shannon presents the idea of unicity distance in [12] . This is defined as the amount of ciphertext that would be required to reduce the number of spurious keys to zero. Put another way, unicity distance is the amount of ciphertext the attacker would require to ensure the correct key has been found, given enough time to exhaustively search all keys. Obviously, the larger the unicity distance, the better for the security of the cipher, as it has a direct relationship on the complexity of a ciphertext only attack.

2.2.3 Completeness, Avalanche Effect and the Strict Avalanche Criteria

Completeness, avalanche effect and the Strict Avalanche Criteria (SAC) are related measures of how well a cipher is designed. In fact, they are an expansion on Shannon's concept of diffusion. In [22], a cipher is said to be complete if each output bit is dependent on all of the plaintext bits in the output. Feistel introduces a related concept, the avalanche effect, [17] whereby a change in any one input bit should

result in a change of $\frac{1}{2}$ of the output bits.

In [23], Webster and Tavares combine the previous ideas to define the concept of SAC. By definition of SAC, a change in one input bit should change each output bit with a probability of $\frac{1}{2}$. Ciphers exhibiting this property do not have a strong correlation between input and output bits.

2.3 Introduction to Cryptanalysis

In the past, many ciphers depended on the attacker not knowing how they worked to ensure their security. In modern cryptography this is no longer the normal procedure. Modern cryptographers design their ciphers according to Kerckhoff's Principle: assume that your attacker is familiar with the algorithm in use. This means that the attacker cannot just work on decrypting intercepted data, but also work on finding flaws in the cipher itself.

There are four common types of cryptanalytic attacks: known ciphertext, known plaintext, chosen plaintext and chosen ciphertext. A known ciphertext attack is the same as the attacker intercepting ciphertext during transmission. No other information is known. In a known plaintext attack, the attacker has the benefit of not only having the ciphertext, but also the corresponding plaintext. For the chosen plaintext attack, not only is the plaintext known, but the attacker is assumed to have been able to access the encryption device and thus is able to choose a particular plaintext of interest and to determine the corresponding ciphertext. The chosen ciphertext attack is similar to the chosen plaintext; however, access to the decryption device has been gained and therefore the attacker can choose a ciphertext of interest and obtain the plaintext [11].

In all cases, the objective of the attack is to gain information about plaintext encrypted given only ciphertext. The ultimate goal is to obtain the key, but any

information revealed will allow the attacker to reduce the number of keys needed to search. In fact, for an ideal cipher, an exhaustive key search (trying all possible keys) is the only method of attack. In practice, many ciphers leak information about their key, reducing the size of the search space. An attack on a cipher is considered to be successful when is able to reduce the search space to less than the 2^n possible keys (where n is the key size in bits). Although many attacks are theoretical in nature and not possible to implement due to the time or amount of data required, any weakness found is usually considered an indication that there may be others to find, and therefore the cipher will be considered inadequate.

The following sections outline some examples of attacks found in the literature.

2.3.1 Meet–in–the–Middle Attack

In [24], Merkle and Hellman introduce the concept of a meet in-the middle attack. When investigating ways to improve ciphers by running data through a cipher multiple times, they noted a flaw in blindly choosing two keys and running the data through two iterations of the cipher. Even though the key length is effectively larger, they found that with a trade off on the amount of memory required, the attacker can attack the cipher in only double the time.

To do this, the attacker must have access to a set of plaintexts and corresponding ciphertexts. The attacker first encrypts a portion of the plaintext for all possible keys through the first half of the combined cipher and stores the result. The attacker then takes the corresponding portion of the original ciphertexts and does a partial decryption of the data for all known keys. If a match is found between the encrypted and decrypted data (in the middle of the cipher), a second test with more data can be performed to show the correct combined key has been found.

This attack exploits the fact that in doubling the key size, it is expected that the number of exhaustive key search operations will go from 2^n to 2^{2n}. However, if there

is enough available memory (2^n blocks), this attack can be performed in 2×2^n or 2^{n+1} operations. When designing a cipher, care should be employed to ensure this flaw is not made available to an attacker.

2.3.2 Linear Cryptanalysis

In [25], Matsui presented an attack on the DES cipher which exploited a bias derived from the probability of linear equations of data in the cipher. This attack is known as linear cryptanalysis and is outlined in a more general form in [18].

The idea behind the attack is to find linear equations involving both input and output bits which have a high or low probability of being satisfied. These equations are commonly of the form

$$x_{i_1} \oplus x_{i_2} \oplus \cdots \oplus x_{i_j} \oplus y_{k_1} \oplus y_{k_2} \oplus \cdots \oplus y_{k_l} = 0 \tag{2.13}$$

where the i and k subscripts are indicators of the position of the bit in the input (x) and output (y), respectively. If a cipher is completely random, an equation of this form should hold with a probability of $\frac{1}{2}$. If an equation of this form can be found with a large bias in the probability, it can be used to extract subkey bits within the cipher.

The first step in setting up a linear cryptanalysis attack is to examine the non–linear elements of the cipher for possible linear approximations. This can be achieved by running all possible inputs through the cipher component (e.g. s box), masking off all combinations of input and output bits, and counting the number of times the exclusive or of the masked input bits is equal to the exclusive or of the masked output bits. Once this procedure is complete, the combinations with the largest and smallest counts are those with the largest bias and can therefore be used for the attack.

When the analysis of all non–linear components is complete, the attacker must

then string together the inputs of interest to provide a complete path through to the second last round of the cipher. For instance, if you have approximation for one component of

$$x_1 \oplus x_3 = y_3 \tag{2.14}$$

the next component in the chain should use the output bit y_3 as its input. In using this method, the total bias of the approximation can be calculated using Matsui's Piling Up Lemma

$$\epsilon_{1,2,3,\ldots,n} = 2^{n-1} \prod_{i=1}^{n} \epsilon_i. \tag{2.15}$$

where n represents the number of active components and ϵ_i is bias of probabilities, or the amount by which they deviate from $\frac{1}{2}$.

This equation is derived with the assumption that all components are independent, which they are strictly not. However, in most cases it provides a close enough approximation.

Once an approximation through to the second last round has been found, the attack can commence. The output bits from the approximation indicate which bits of the final subkey one can attack, based on their relationship to the final subkey in the last round. The attack is a known plaintext attack where the attacker uses input bits involved in the linear approximation. The output is run back through the final round using all possible values for the partial subkey under attack in the final round. With each decryption, a count is kept of how many times the expected value of the approximation at the output of the second last round is consistent with the input bits. Once all possible subkeys have been tried, the partial subkey with the largest bias is chosen as the actual value of the bits in the subkey. In [25], the data complexity of the attack is given to be approximately $\frac{1}{\epsilon^2_{1,2,3,\ldots,n}}$.

2.3.3 Differential Cryptanalysis

Introduced by Biham and Shamir in [26], differential cryptanalysis is a chosen plaintext attack. It is based on the fact that in an ideal cryptographic algorithm, given any difference in the input, ΔX, the probability of resulting in a particular difference at the output, ΔY, would be $p_{\Delta X,\Delta Y} = \frac{1}{2^n}$ where n is the number of bits in X. If we can find a difference in the input that deviates from this probability, we can exploit it to extract information about the cipher key.

Differential cryptanalysis classically uses sets of pairs of inputs that have a common difference $\Delta X = X_0 \oplus X_1$, where X_0 and X_1 represent two different input values and $\oplus$ represents bit wise exclusive or. These pairs, when used as input to the cipher, result in an output difference $Y_0 \oplus Y_1$ which with high probability is equal to a value ΔY. The pair, $(\Delta X, \Delta Y)$ is referred to as a differential.

To find differentials of interest, the various primitives of the cipher are usually analyzed to find any pattern that may be exploitable. For example, if a cipher uses substitution boxes, or s boxes, each individual s–box can be analyzed to find a differential that occurs with a probability greater than expected. The larger the deviation, generally the greater the likelihood of success for the attack on the cipher. Once the individual components are analyzed, the differential must be combined to find a total differential that passes through the entire cipher with the greatest probability.

Highly likely differentials can be exploited to determine key bit information. Once a highly probable differential $(\Delta X, \Delta Y)$ is found for r rounds of an $r+1$ round cipher, the last round subkey can be attacked. Many pairs with the input difference ΔX are encrypted over $r+1$ rounds. Each pair is then decrypted for the last round using all possible subkeys. A check for the output difference ΔY of round $r-1$ is performed and a count for the number of time it occurred with the given subkey is incremented. The actual subkey will result in the largest count of the ΔY occurrences as predicted

by the differential.

2.3.4 Side–Channel Attacks

Side channel attacks are attacks that are not focused on the cipher itself but rather its implementation. When implementing an algorithm, care must be taken such that details of the data within are not revealed due to implementation measurements. Two types of common side channel attacks are outlined in the following sections: timing attacks and power attacks.

Timing Attacks

Introduced by Kocher in [27], timing attacks exploit implementations of a cipher which take an amount of time, dependent on the input data. The author lists possible reasons for the variability as: performance optimizations to bypass unnecessary operations, branching and conditional statements, RAM cache hits, and processor instructions such as multiplication and division that run in non–fixed time.

A simple example of an implementation vulnerable to this type of attack would be one such that certain operations are skipped based on a condition of the data. In his paper, Kocher gives an example of a modular exponent algorithm that could be used with RSA, which includes the following pseudocode:

```
if (bit k of x) is 1 then
        Let R_k = (s_k · y) mod n
else
        Let R_k = s_k.
```

It is obvious from this code that the *if* case will take more time than the *else* case. The author goes on to show how by measuring this time difference an attacker can extract the bits from the exponent x, the key to decrypting the ciphertext. It should also be noted that though this attack was originally implemented on a public key

cipher, it can also be used on symmetric key ciphers, an example of which is given in the next chapter.

Power Attacks

In [28], Kocher et al. investigate implementation attacks based on measurement of power consumption. They put a 50 ohm resistor in series with power or ground for the circuit, and measured the voltage across the resistor, sampling it at a high rate of speed. Since processing circuits are made up of many transistors which change state, dynamic power consumption is proportional to transistor state changes and hence processing of the different data results in unique power traces measured as current entering the circuit.

An example of an implementation that can be attacked is the DES key schedule, which involves the rotation of 28 bit key registers. One implementation method would be to use a conditional branch to check the bit being rotated off, to determine if the bit rotated onto the other side need to be set. Using this method, rotating a bit that is set would have a different signature than a bit that is not. These signatures could be measured in order to determine the number of set and unset bits in the registers.

2.4 Summary

This chapter has given an introduction to cryptography including types of algorithms and desired properties. An outline of common cryptanalysis techniques was also presented. The following chapter looks more closely at a single cryptographic primitive, the DDP. Example ciphers using DDPs are presented, including an in depth look at CIKS 1. This cipher is also analyzed using some of the techniques presented in this chapter.

Chapter 3

Data-Dependent Permutations and CIKS–1

Data–Dependent Permutations were introduced in [29], an IBM patent filed in 1977, in the form of Data Dependent Rotations. In the author's patent "System for Machine Enciphering," data within the cipher is subject to one of a set of operations depending on other data. This did not make the primitive strictly a DDP as other operations could be used; however, it was the first example of permutations being selected via the data itself.

It was not until Ron Rivest published [1] that a DDP primitive began to gain attention. In his paper, Rivest proposed RC5, a simple cipher based mainly on a DDR (a subset of the DDP functions) that was considered both fast and secure. Further study of the DDR in [10] showed it to be resistant to both linear and differential cryptanalysis.

In this chapter we look at the DDP, a class of nonlinear cryptographic primitives that are gaining popularity in cryptography. The properties of the DDP are reviewed, as well as various implementations. We then look at example ciphers using DDPs.

In the latter part of the chapter we focus on the CIKS 1. Since this cipher is built with a major dependency on DDPs for nonlinearity, it is of particular interest.

We look at the various components used and their role in the cipher. We investigate the data flow through the cipher and examine how the cipher performs for accepted cryptographic properties. Finally, we take a look at known attacks on the CIKS 1 cipher.

3.1 Data–Dependent Permutations

A permutation, Π, of width n is a one–to-one total function from a finite set, $\{0, 1, ..., n-1\}$ to itself. A Controlled Permutation (CP) is an indexed set of p permutations, $\{\Pi_0, \Pi_1, ..., \Pi_{p-1}\}$ in which the input data is permuted by a member of the set chosen by the Control Vector (CV) to produce the output. Given an input x and a control vector i, where $0 \leq i < p$, then the output is y such that $y = \Pi_i(x)$. The maximum size of the set is $n!$ different permutations. However, due to the number of control bits required and the difficulty of structuring the selection of the permutation, this is impractical. In practice, CPs are normally used with smaller CVs when used as cryptographic primitives. When a CP is controlled by a subset of the data in the function, we consider it a DDP.

In [2] $P_{n/m}$ is defined as a controlled permutation of n input bits such that there are 2^m permutations, defined by a CV of size m. $P_{n/m}$ is of order h, if for each sequence of $h \leq n$ input bits, $x_0, x_1, ..., x_{h-1}$, and $h \leq n$ outputs, $y_0, y_1, ..., y_{h-1}$, there exists at least one CP which moves x_i to y_i for all $i = 1, 2, ..., h$. CV is of maximal order if $h = n$ and it therefore contains the set of all $n!$ possible permutations. Also, $P_{n/m}$ is considered strict if and only if $\Pi_j \neq \Pi_k$, where $i, k \in \{0, ..., 2^m - 1\}$ and $j \neq k$. Finally, $P^{-1}_{n/m}$ is defined to be the inverse of $P_{n/m}$ if and only if each Π_k from $P_{n/m}$ is the inverse of $\Pi_k{}^{-1}$ from $P^{-1}_{n/m}$, for each k.

In order to be used for cryptographic application, DDPs should be designed with no bias. Therefore, over all permutations in the set $P_{n/m}$, if the control vector k is

chosen uniformly, then $Pr(\Pi_k(i) = j) = \frac{1}{n}$ for all i and j. DDPs exhibiting this property are considered to be uniform. In practice, it is difficult to build uniform DDPs of a useful size, so they are instead designed to be approximately uniform and combined with other elements to compensate [2].

A DDP can be designed in many ways depending on the requirement of the cipher. In some ciphers (such as RC5 and RC6), the DDPs are implemented as a rotation of the bits. Other possibilities include bit swapping, block swapping or combinations of each of these techniques.

3.2 Ciphers with Data–Dependent Permutations

There are many ciphers that use DDPs, but there are two in particular that have received more attention in the literature: RC5 and RC6. Although neither of these ciphers use DDPs in their more general form (both of these ciphers use DDRs), the permutations are a major component of their algorithm. We take a cursory look at these ciphers here before we move on to ciphers using the more general DDP: Spectr H64, Cobra H64, Cobra H128 and CIKS 1.

3.2.1 RC5

Much of the interest in data dependent primitives came about as a result of RC5, published by Ron Rivest in [1]. The cipher was designed to be simple, fast (in both hardware and software), and variable in input size, key size and number of rounds. It was also a stated goal of the author to highlight the use of DDRs. In fact, the DDRs are the only nonlinear component of the cipher.

The algorithm is shown in Figure 3.1 where $x <<< y$ is the rotation of x by y bits to the left and $\oplus$ is exclusive or. RC5 requires a key expansion array, S, which is also computed using DDRs. The input is broken into two w bit plaintext words,

$A := A + S[0]$
$B := B + S[1]$
for $i := 1$ to r **do**
  $A := ((A \bigoplus B) <<< B) + S[2i]$
  $B := ((B \bigoplus A) <<< A) + S[2i + 1]$
end for

Figure 3.1: RC5 Encryption Algorithm

A and B (where w is a parameter of the cipher implementation) and initially added to the first two subkeys. Then r rounds of the cipher are executed where the input is combined with the other via an exclusive-or and then rotated by the $\log_2(w)$ lower bits of the second input. Subkeys are also added to each input. The final values of A and B are the ciphertext.

The algorithm is deceptively simple at only five lines long, but has held up well against attack. In [10], the authors showed that the rotations used in RC5 help to make linear and differential attacks on RC5 impractical for implementations using 12 or more rounds. In [9], the cipher's author claims that although there have been theoretical attacks on RC5 (mostly to due the effective limit on the number of data bits influencing the rotations), there have been no practical attacks.

3.2.2 RC6

RC6, presented in [9], is a direct descendant of the RC5 cipher and was a candidate for the AES. Both the key schedule and the cipher itself are derived from the original cipher. In the case of the key schedule, the algorithm is similar in all aspects except the number of keys derived, of which RC6 has more.

The encryption algorithm itself has additional elements. Fixed rotations based on the logarithm (base 2) of the size of the data, w, have been added along with integer multiplication modulo w. Both of these operations are employed to ensure more data bits are involved in the determination of the rotation amount in the DDRs employed,

$B := B + S[0]$
$D := D + S[1]$
for $i := 1$ to r **do**
  $t := ((B \times (2B + 1)) <<< \log_2(w)$
  $u := ((D \times (2D + 1)) <<< \log_2(w)$
  $A := ((A \bigoplus t) <<< B) + S[2i]$
  $B := ((B \bigoplus u) <<< A) + S[2i + 1]$
  (A, B, C, D) = (B, C, D, A)
end for
$A := A + S[2r + 2]$
$C := C + S[2r + 3]$

Figure 3.2: RC6 Encryption Algorithm

thus overcoming a perceived weakness in RC5. The algorithm itself is given in Figure 3.2.

In [30] the authors found that using the chi squared (χ^2) test, they could distinguish the results of the cipher from random data when using up to 15 rounds. They present an algorithm for extracting keys which will work in less time than exhaustive key search, requiring 2^{119} plaintexts with a time complexity of 2^{215} for a cipher key of 256 bits. The authors also note they detected weak keys in the analysis of the cipher. The keys are not enumerated, however they are visible due a large deviation in the χ^2 tests.

The authors of [31] also use the χ^2 test as a distinguishing algorithm in an attack of RC6. The authors propose a simplification to the cipher, removing the whitening steps (the calculations of B and D at the beginning of the cipher and A and C at the end), calling it RC6W (RC6 without Whitening). On this new variant of the cipher, they present an attack which can recover the key of an r round cipher using $2^{8.1r-13.8}$ plaintexts with a probability of 90%.

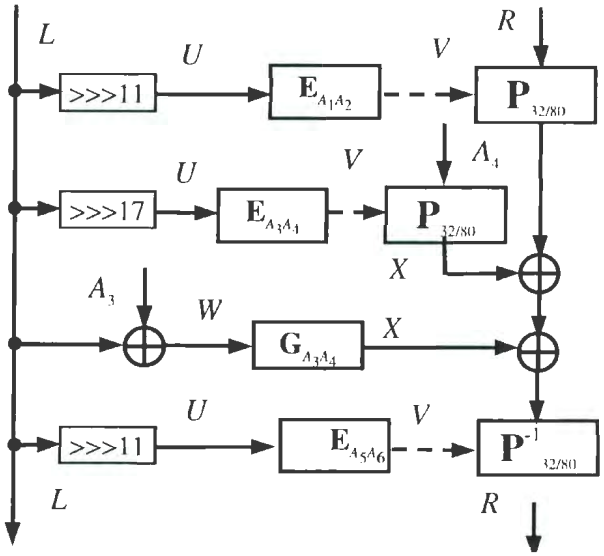

Figure 3.3: The Spectr–H64 Encryption Function [3]

3.2.3 Spectr–H64

In [3], Spectr–H64 is presented as a fast, hardware–oriented cipher with extensive use of DDPs . The same algorithm is used for both encryption and decryption as shown in Figure 3.3. The cipher has 12 rounds with two additional transformations at the beginning and end of the function. The additional transformations are a combination of DDPs and inverters that function on groups of two bits. In the case of the initial transformation, one bit is inverted at the output of the DDP. In the case of the final transformation, one of the bits is inverted at the input of the DDP.

Each round of the cipher uses a 192–bit subkey which is derived from the 256–bit key for the cipher via the given key schedule. Of the 192 bits, 32 are added to the data via an exclusive or and 160 are input into the extension function **E** and used as control bits for the DDPs. In addition to the DDPs, Spectr H64 includes a nonlinear function **G** which operates on key and data, the result of which is exclusive ored back into the cipher data.

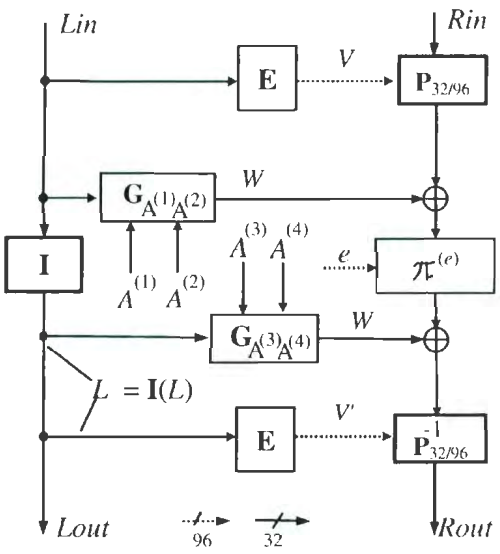

Figure 3.4: The Cobra–H64 Encryption Function [4]

3.2.4 Cobra-H64 and Cobra-H128

In [4], Sklavos, Moldovyan and Moldovyan present the Cobra family of ciphers. These ciphers are again based on DDPs and are the result of analysis of their previous ciphers Spectr H64 and CIKS 1 (presented in the next section). The cipher has two variants, Cobra H64 (shown in Figure 3.4), which has a 64 bit block size, and Cobra–H128 (shown in Figure 3.5) which has a block of 128 bits. The ciphers each have four subkeys of lengths 32–bits and 64 bits, and are comprised of 10 and 12 rounds respectively.

Each round starts with an initial transformation and ends with a final transformation during which part of the subkey is added modulo–two to the left and right halves of the data. Each round consists of a *crypt* function during which the Right Hand Side (RHS) of the data is permuted by a DDP using the Left Hand Side (LHS) as the control. The ciphers use DDPs, but do not depend on them entirely to provide nonlinearity. The cipher also includes a nonlinear function, **G** similar to the one

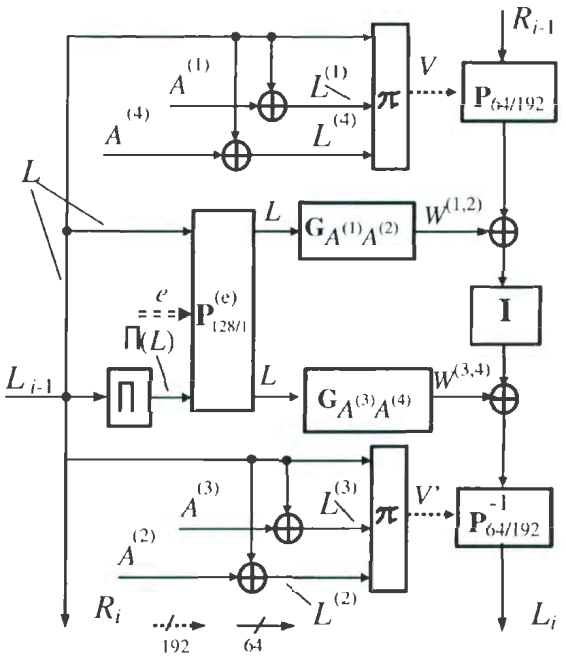

Figure 3.5: The Cobra H128 Encryption Function [4]

found in Spectr H64 which is a function of half the round data and half the subkey, the result of which is exclusive ored into the other half of the data. This is done twice, with the LHS and RHS being permuted in between the operations. Finally, the RHS is once again permuted via the inverse of the DDP used previously, with the LHS data being used as the control once again. Other components found in the ciphers are: I, a permutation involution; H, a fixed permutation; and π, a switchable fixed permutation.

The authors' analysis of the ciphers shows that they are resistant to differential cryptanalysis after eight and ten rounds for Cobra H64 and Cobra H128, respectively. They also state both ciphers are resistant to linear cryptanalysis after five rounds. In [32], the authors propose implementation of the Cobra ciphers, Cobra-S128, which is designed to be implemented in software. In the next section, we look at another cipher proposed by Moldovyan and Moldovyan which depends almost exclusively on

DDPs for nonlinearity.

3.3 The CIKS–1 Cryptographic Cipher

In [2], Moldovyan and Moldovyan proposed a new 8 round cipher based on DDPs. CIKS–1 was presented as a fast, hardware oriented cipher. It relies on DDPs for their speed in hardware and is designed to lack pre computation of key scheduling. Preliminary analysis of the cipher by the authors showed that it can easily obtain speeds of 2Gb/s and was resistant to both linear and differential cryptanalysis.

We chose CIKS–1 for the investigation of DDPs as cryptographic primitives for three main reasons. First, the cipher is relatively new, and as such has not been extensively tested. Second, it uses DDPs in a more general form, rather than using the subset of DDR functions. Finally, the cipher relies mainly on the DDP for nonlinearity in the output.

In this section, we will look at the DDPs defined for use in the CIKS–1 cipher by its authors. We will then look at the other components used before moving on to the algorithm as a whole. We will then investigate CIKS–1 with respect to some common cryptographic principles before examining known attacks on the cipher.

3.3.1 CIKS–1 Data–Dependent Permutations

The CIKS–1 DDPs are designed to be constructed from basic building blocks. The most basic block, the $P_{2/1}$ controlled permutation, takes two input bits, x_0 and x_1 and a single control bit cv_0. If $cv_0 = 0$, x_0 and x_1 are swapped to form the output; otherwise, they pass through, positions unchanged. This simple permutation is combined in layers to make the more complex DDPs that appear in the cipher.

For instance, in order to make a $P_{4/4}$, two layers of two $P_{2/1}$ blocks are used. As seen in Figure 3.6, a "butterfly" pattern is used to connect the two layers of

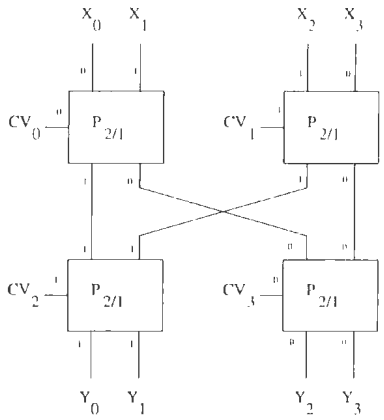

Figure 3.6: $P_{4/4}$ Data Dependent Permutation with Example Inputs.

permutations. This ensures that bits that are grouped together in the input are not continually swapped with each other as they move through the layers. It also guarantees that a CV which is comprised of mostly "1"s will not result in a poorly permuted output.

In order to decrypt the ciphertext created using the DDPs, an inverse function is required. This is produced simply by running the ciphertext through the permutations in reverse using the original control vector. An example of the $P_{8/12}$ DDP and its inverse, ${P^{-1}}_{8/12}$ are shown in Figure 3.7. Both of the example DDPs shown, $P_{4/4}$ and $P_{8/12}$ are strict and have an order of 1.

The CIKS–1 DDPs are both fast in operation and efficient in hardware implementation. Since the CVs are available at the time of the permutation operation, there is no setup time required and the time delay is only that of the permutations themselves. The hardware cost for the $P_{n/m}$ DDP is given as $4m$ NAND gates [2].

3.3.2 Other Cipher Components

Although it depends heavily on its DDPs for nonlinearity, CIKS–1 does employ a few other primitives to complete the cipher. Here we present a discussion of each one,

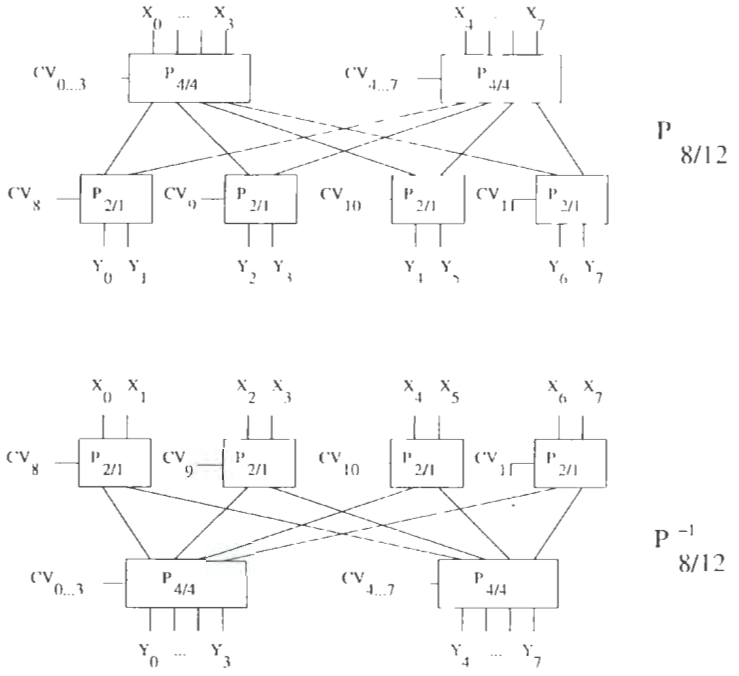

Figure 3.7: $P_{8/12}$ Data Dependent Permutation and its Inverse.

including any design decisions indicated by the authors in [2].

Fixed Permutations and Rotations

In addition to the DDPs used in the cipher, the authors have also included two fixed permutations and three fixed 7-bit rotations to the right. The fixed permutations are applied to control vector data and in [2] the authors indicate that this is done to increase diffusion (the relationship of the effect of changing an input bit on the output defined by Shannon in [12]). The rotations affect both the cipher data and the CVs; however, no reason for their inclusion is given.

Two–bit Additions

CIKS 1 uses modulo 2^2 addition to combine the left and right data at the end of each round. Sixteen of these addition blocks are used in parallel, each operating

on only two bits of the input data with the carry bit out of each 2 bit block being ignored. The cipher authors state that modulo 2^2 addition was chosen over a full 32 bit addition to avoid the long carry propagation delay associated with the latter. However, since each addition is isolated to affecting only two bits of data, it also introduces a limit on the propagation of change within the cipher.

Key Addition

Key addition in the cipher is achieved using an exclusive or operation. This is a common way of mixing the key into the ciphertext in cryptographic algorithms. The authors choose not to directly add in each cipher subkey; they instead first permute it using a DDP based on the LHS cipher data. In [2] this is referred to as an internal key schedule, a way of ensuring the same subkey is not continually added between key changes.

3.3.3 Description of Cipher

The CIKS 1 cipher is a fast, hardware oriented cipher, with its principle security component being DDPs. It is a block cipher with a block size of 64 bits. The cipher is composed of eight rounds, each with a 32 bit subkey, $K_0, K_1, ..., K_7$, for a total key size of 256 bits. A single round of the cipher is shown in Figure 3.8. The solid lines in the diagram show the flow of data and the dashed lines are control vectors. Permutations are labeled n/m, where n is the number of bits permuted and m is the number of bits of control. Additional labels of the form P_i are given to identify the individual permutations.

The 64 bit data is split into half for input to the left and right hand sides. The RHS is expanded and used to permute the data on the LHS using P_1. The LHS data is then expanded, with a portion being permuted using P_3 based on the subkey data, and permuted using the fixed permutation Π_1, the result of which is used to

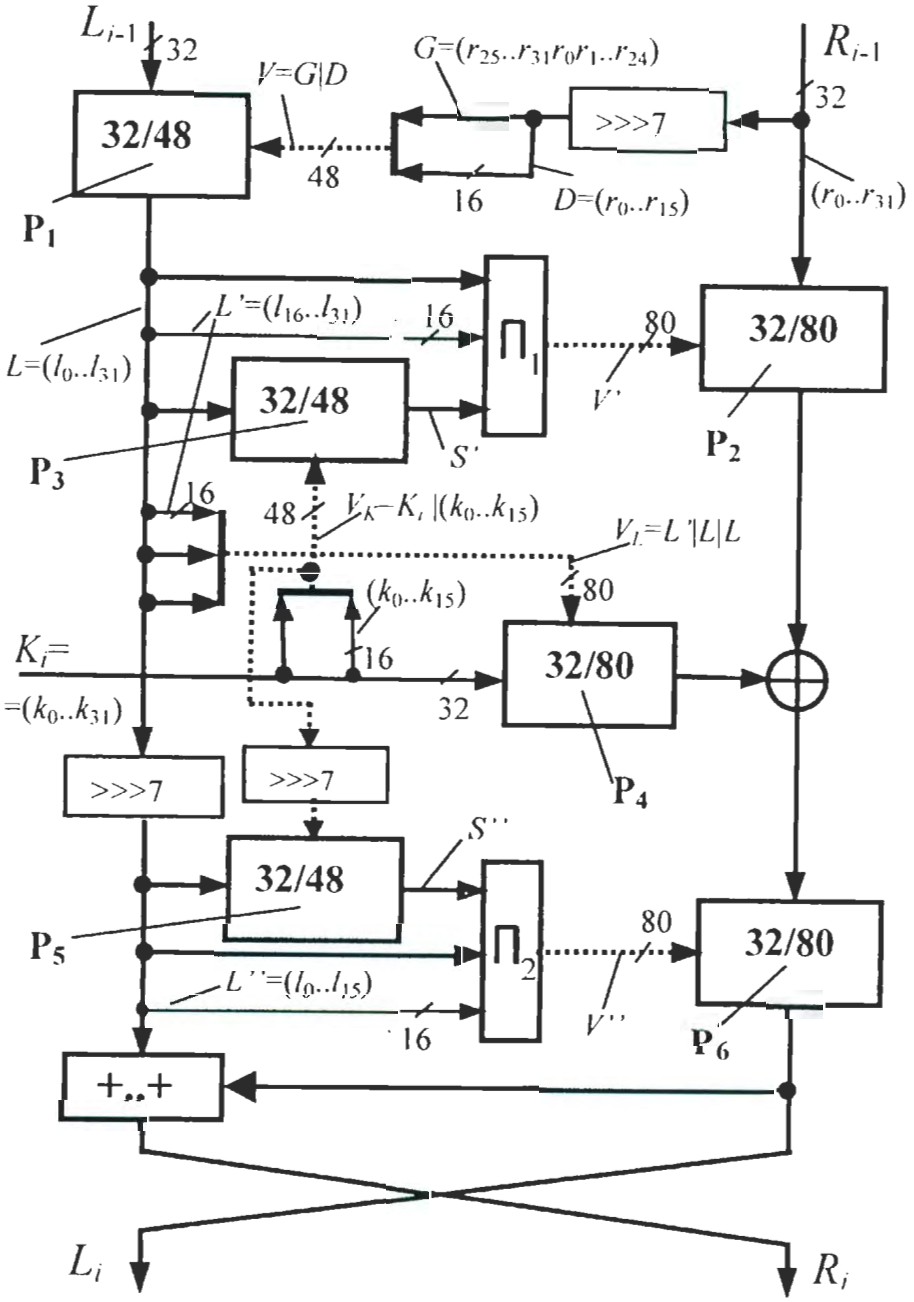

Figure 3.8: The CIKS 1 Encryption Algorithm.[2]

Input: 64 bit plaintext $L \mid R$, where L and R are the 32 bit sub-blocks of data.
for $i = 0$ to 6 rounds **do**
  $L \mid R := Round(L \mid R); L \leftrightarrow R;$
end for
$L \mid R := Round(L \mid R);$
Output: ciphertext $L \mid R$

Figure 3.9: Encryption Algorithm for Full 8 Rounds of CIKS–1

permute the RHS data with P_2. The subkey is permuted next using an expanded version of the LHS data as the CV for P_4 before combining it with the RHS data via exclusive or. Once again, the LHS data is expanded, with a portion being permuted using P_5 based on the subkey data, and permuted using the fixed permutation Π_2, the result of which is used to permute the RHS data with P_6. Finally, the RHS data is added to the LHS data via 16 parallel modulo–2^2 additions, before swapping the left and right sides at the end of the round (in all but the last round). The full 8 rounds of the cipher are defined algorithmically in Figure 3.9 where *Round* is a single round function of CIKS–1, excluding the final swap.

The decryption algorithm for CIKS–1 is shown in Figure 3.10. The 16 parallel 2–bit addition blocks are replaced with the same number of 2–bit subtractions. As well, all other primitives that act directly upon the data are replaced by their inverse functions. Other rotations and DDPs remain the same as they are required to build identical CVs. The swapping of the left and right sides remains at the end of all but the last round.

3.3.4 Initial Analysis of CIKS–1

An initial analysis of the CIKS–1 cipher was done when looking for weaknesses that could be exploited. In the following sections, we look at general cryptographic concepts, applying them to CIKS–1. More detailed analysis of the cipher is done in the following chapters.

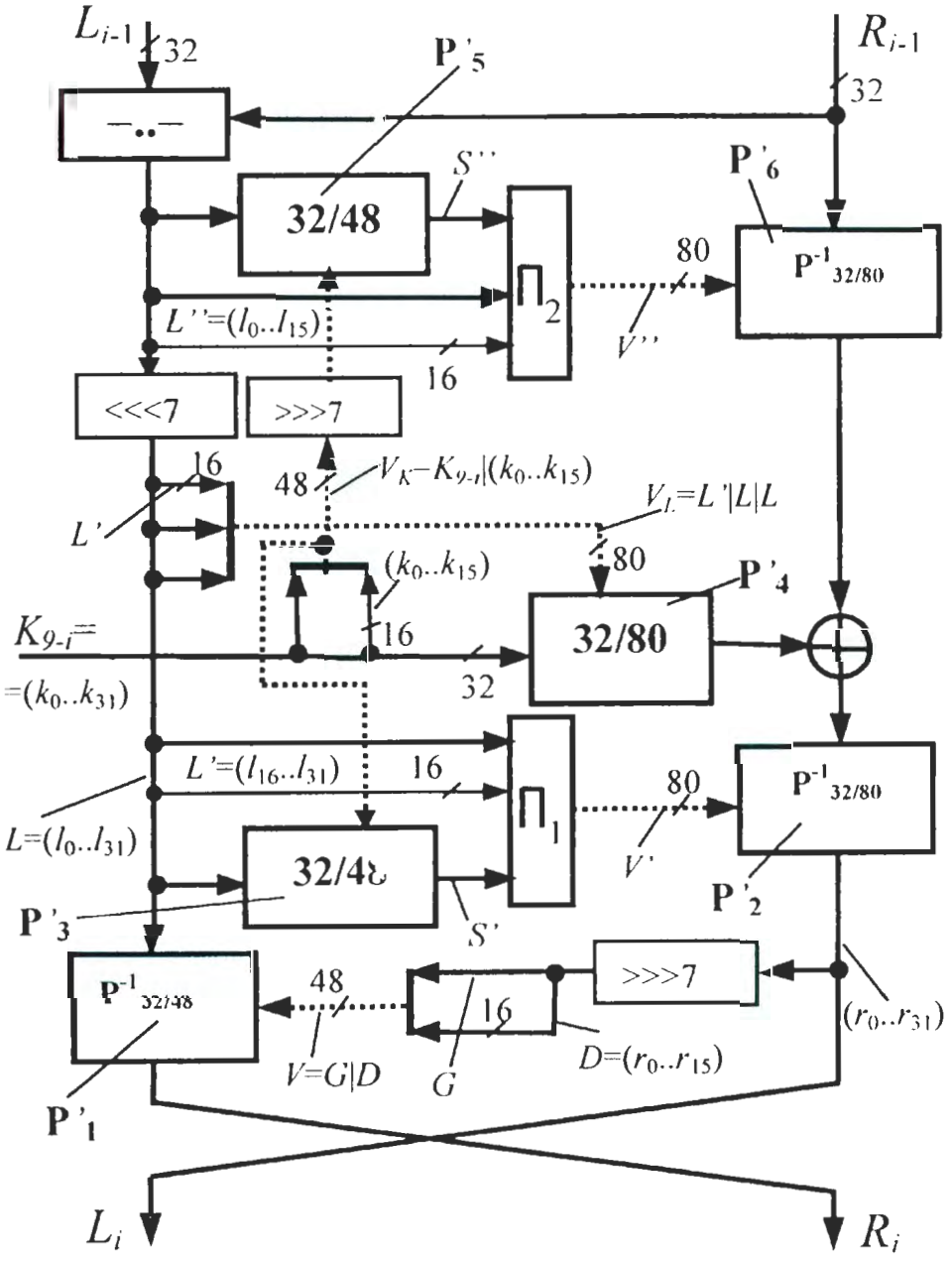

Figure 3.10: The CIKS 1 Decryption Algorithm.[2]

Avalanche Effect and the Strict Avalanche Criteria

The authors of CIKS–1 presented a brief analysis of the avalanche effect in the original paper. They noted that each control vector bit influences two output bits, with a probability of swapping them equal to $\frac{1}{2}$. Taking into account the swap at the end of each round, they state that after two rounds, every input bit should influence all output bits. During the analysis of the cipher, it was found that the SAC criteria holds. Choosing random inputs and keys, the data was encrypted over eight rounds. A version of the data with a one–bit difference was also encrypted. Comparison of the different results showed that the number of output bits changed matched a binomial distribution with probability of $\frac{1}{2}$, thereby confirming the cipher conformed to the SAC.

Key Schedule

There is no key schedule specified for CIKS 1. The authors note that there is an internal key scheduling (IKS) due to permutation P_4, which scrambles each subkey, controlled by the data on the LHS of the cipher. This is considered to be beneficial to the cipher, as the key scheduling can be done in parallel with other parts of the cipher, eliminating pre computations and thus adding no time delay due to frequent key changes.

However, the CIKS 1 paper does not give any indication as to how these subkeys should be derived from the master key. Leaving this to the implementor of the cipher allows for the increased chance of using weak keys. For example, the cipher can be implemented such that each round uses the same subkey, depending on the IKS to scramble it differently for each round. In later chapters we examine an attack based on a set of weak keys and show how it can exploit a poorly chosen key schedule.

3.3.5 Known Attacks

In [33], a chosen plaintext attack is presented on a reduced 5 round version of the CIKS 1 cipher. The authors exploit the fact that DDPs have no bearing on the parity of the data and the parallel additions can be shown to be close to linear in nature. In order to cancel out the effect of the first round, they choose the LHS plaintext such that the jth bit of the plaintext, x_j, is 0, for all $j = 2 \times i + 1$ where $0 \leq i \leq 15$. This ensures that the probability of their linear approximation for the addition in the first round holding is one.

The authors then attack the following four rounds, revealing key information for the fifth round. This is achieved by decrypting 2^{36} ciphertext pairs using all possible 2^{32} subkeys. The resulting LHS data has the RHS data subtracted so that the value of the LHS before the fourth round addition is obtained. The parity of this data is then compared to that of a linear approximation and a record of matches is recorded for each subkey. The subkey with the best results is chosen as the last round subkey. The authors calculate that the attack will succeed in revealing the final round subkey with a probability of 78.5%, with a data complexity of 2^{36} and time complexity of $2^{65.7}$.

In [34], the authors implemented a timing attack on CIKS 1. Though designed for hardware, the authors note that the cipher can potentially have a weakness if naively implemented in software, specifically in single threaded environments such as a micro controller or smart card. The attack presented exploits the case when the DDPs are simply implemented as swaps controlled by the CV as a conditional statement such as:

```
if (cv == 0)
        swap(x1,x2);
```

In this case the author's show that the time taken for encryption reveals information about the Hamming weight of the key when the plaintext and ciphertext are

known. The authors also propose an implementation of the DDPs using pure boolean expressions that is immune to this method of attack.

3.4 Summary

In this chapter we have given an introduction to DDPs and examples of ciphers in which they are used. We then introduced the CIKS 1 cipher, as introduced by Moldovyan and Moldovyan in [2]. This introduction included an analysis of the components of CIKS 1 and known attacks on the cipher. In the following two chapters, two new attacks on CIKS–1 are presented. The first exploits a class of weak keys for the cipher. The second is a differential attack that focuses on the weight of the input differentials.

Chapter 4

Weight Based Attack on CIKS–1

In [2], Moldovyan and Moldovyan provided a preliminary analysis for the strength of the proposed cipher CIKS–1. The evaluation was done for two of the most successful cryptanalysis techniques known, linear and differential cryptanalysis. Using these attacks, while making assumptions in the attacker's favour, the authors estimated that approximately 2^{64} and 2^{66} plaintexts would be required to attack the cipher with differential and linear cryptanalysis, respectively. In this chapter, we look at an alternate approach to breaking the cipher that exploits low Hamming weight subkeys.

In [33], Lee *et al.* presented a chosen plaintext attack designed for use on a 5 round version of the CIKS–1 cipher. They present linear approximations of three rounds of the cipher, based on the parity of the data. The time complexity of the attack is estimated by the authors to be approximately $2^{65.7}$ encryptions. Unfortunately, although this attack can be applied to all possible keys, its success is only demonstrated on a five–round version of the cipher. The authors propose that this attack will work on the full eight round cipher through "canonical extension," but do not explain how this can be achieved.

The attack introduced in this chapter will also be a chosen plaintext attack. A look at the effects of the data–dependent permutations will be presented, focusing on their effect on the Hamming weight of the ciphertext. It will be shown that one could

exploit the fact that since there is no prescribed way of implementing a key schedule for the cipher, there is the potential for a class of weak keys to reveal information about the first or last round of the cipher. The attack is then presented which builds on this information to reveal information of subkeys, allowing for a brute force attack on the first subkey using a reduced search space.

4.1 Analysis of the CIKS–1 Components

The following sections analyze the components that make up the entire CIKS 1 algorithm with respect to a Hamming weight based attack. It will look at all the primitives used in the cipher, as well as discuss the problems with not specifying a key schedule. There will then be an analysis of the weight propagation through the cipher as a whole.

4.1.1 The CIKS–1 Permutations

As presented in Chapter 3, the CIKS 1 cipher uses DDPs, which in turn use a portion of key or data to control the permutation of the individual bits of data as they pass through the cipher. The CIKS 1 DDPs are labeled $P_{n/m}$, where n is the number of bits to be permuted and m is the the number of bits in the CV. For example, the 2-bit permutation $P_{2/1}$ produces an output of the two input bits using a 1 bit control vector. If the CV is a 0, the input bits are swapped, otherwise they pass through the permutation unchanged. All of the CIKS 1 DDPs are formed using the smaller two bit permutations in layers, interconnecting each layer with a fixed permutation.

This "butterfly" patterned permutation is used to keep the input bits of the permutation from being grouped together as they pass through DDPs. This keeps the individual bits from continually being swapped in the same pair, and prevents the

existence of a single control vector that would make the permutation a unity operation with respect to data position. A CV of all 1s would not permute the bits at the smaller $P_{2/1}$ permutations in the DDPs, therefore the "butterfly" pattern prevents the entire input passing through the larger DDP unchanged.

It is important to note that the DDPs in CIKS–1 have absolutely no effect on the weight of the ciphertext. The individual values of the input bits of each permutation remain unchanged; only their position is modified. Therefore, the DDP has done nothing to affect the Hamming weight of the output data.

4.1.2 Fixed Permutations

CIKS–1 contains two fixed permutations, Π_1 and Π_2. These permutations never operate on the actual data of the cipher itself, only the data used in CVs. This appears to be done to prevent the user from processing data backward through the cipher to reveal information about the subkeys. Since they perform no direct operations on the cipher data itself, the fixed permutations have no effect on the output Hamming weight.

4.1.3 Key Addition

The subkey for each round of CIKS–1 is always added to the RIIS data. It is first permuted via a DDP and then added to the data via an exclusive or operation. This is a standard way of inserting the key into many ciphers as it is simple and efficient to implement.

The exclusive or is one of only two operations used in the cipher where the weight of the data can be affected. This operation has a probability of changing the data bit $\frac{1}{2}$ of the time, assuming random key inputs. A problem arises when a characteristic of the key is not consistent with random behavior, particularly in the case of having a key with a significantly low Hamming weight. In this case, the majority of the subkey

being added via exclusive–or will be binary zeros. By definition, when a binary zero is exclusive ored with another binary bit, x, the result will be x. Thus, in the case of low weight keys, the Hamming weight of the data is only modestly affected by the exclusive or operation.

4.1.4 Addition

The only other place the weight of the data in the CIKS 1 cipher can be changed is the parallel addition block at the end of each round. Here, 16 parallel modulo 2^2 additions combine the left and right hand side data before they are swapped to form the output of the round. Each addition operation carries out 16 2 bit binary additions, with the carry being ignored. This has the advantage of being efficient to implement in hardware, but has the disadvantage of grouping small blocks of data together, affecting each other in isolation of other groups.

If x is the data from the LHS entering the addition and y is the data from the RHS, then z is the modified LHS data as a result of the addition. The weight change in the output, Δwt, is defined as a difference in the weight of z with respect to x. An analysis of the the modulo–2^2 addition shows that although there is an influence on the weight of the LHS data, there is still a significant probability that it will remain unchanged. Consider two bit addition as shown in,

$$z_1 z_0 = x_1 x_0 + y_1 y_0. \tag{4.16}$$

We can break Equation 4.16 into two equations:

$$z_0 = x_0 \oplus y_0 \tag{4.17}$$

and

y_1	y_0	z_1	z_0
0	0	x_1	x_0
0	1	$x_1 \oplus x_0$	$\overline{x_0}$
1	0	$\overline{x_1}$	x_0
1	1	$\overline{x_1} \oplus x_0$	$\overline{x_0}$

Table 4.1: Addition output as a function of x_0 and x_1

$$z_1 = x_1 \oplus y_1 \oplus (x_0 \wedge y_0). \tag{4.18}$$

Using 4.17 and 4.18, we can consider the output as a function of x_0 and x_1, for all possible values of y_0 and y_1. This is shown in Table 4.1.

It can be seen from Table 4.1 that the case where $y_0 = y_1 = 0$ results in an output where the weight always remains the same. Conversely, the case where $y_0 = 1$ and $y_1 = 0$ always equates to a change in the weight. In the two remaining cases, any change in the output weight is dependent on the inputs for x_0 and x_1. Table 4.2 is an expansion of these remaining cases. From the table we can observe that there is a probability of $\frac{1}{4}$ that the output weight will remain unchanged.

y_1	y_0	x_1	x_0	z_1	z_0	Δwt
1	0	0	0	1	0	1
1	0	0	1	1	1	0
1	0	1	0	0	0	1
1	0	1	1	0	1	1
1	1	0	0	1	1	2
1	1	0	1	0	0	1
1	1	1	0	0	1	0
1	1	1	1	1	0	1

Table 4.2: Hamming weight change of modulo 2^2 addition.

This analysis shows that each 2–bit addition block has a significant chance the weight of the data will be unchanged after the operation is performed. In fact, six

of 16 possible outcomes result in no change to the weight of the input data at the output.

4.2 Analysis of Weight Change Propagation

When designing a cipher, the goal is to create an algorithm which produces an output that looks completely random for the set of all possible inputs over the set of all possible keys. Ideally, it should not be possible to distinguish between the output of the cipher and the output of a random number generator. One quick check for this property is to examine the mean Hamming weight of the output of the cipher. If truly random, this weight would fit a binomial distribution, thus giving a 64-bit output an average weight of 32 [35].

Since there are very few elements of this cipher which affect the weight of the data as it is encrypted, if the weight of the plaintext input is low, the weight of the data grows slowly as the data progresses through the rounds, particularly if the key has a low weight. This was confirmed by performing five million encryptions for keys with weights from one to eight, and a maximum plaintext input weight of six. The key and input weights were chosen based on initial testing that showed these values constrained the output weights without overly limiting the number of possible inputs. The number of tests was chosen to be large enough to highlight the deviation from the expected mean without being time prohibitive for testing.

The test was executed by encrypting a randomly selected plaintext with a Hamming weight less than or equal to six. After each round of the cipher we noted the weight of the output and calculated the mean overall results. As can be seen in Table 4.3, the weight of the output grows slowly under these conditions. Figure 4.1 further illustrates this by plotting deviation of the expected mean weight against the actual mean weight of low Hamming weight keys (keys with a Hamming weight of eight or

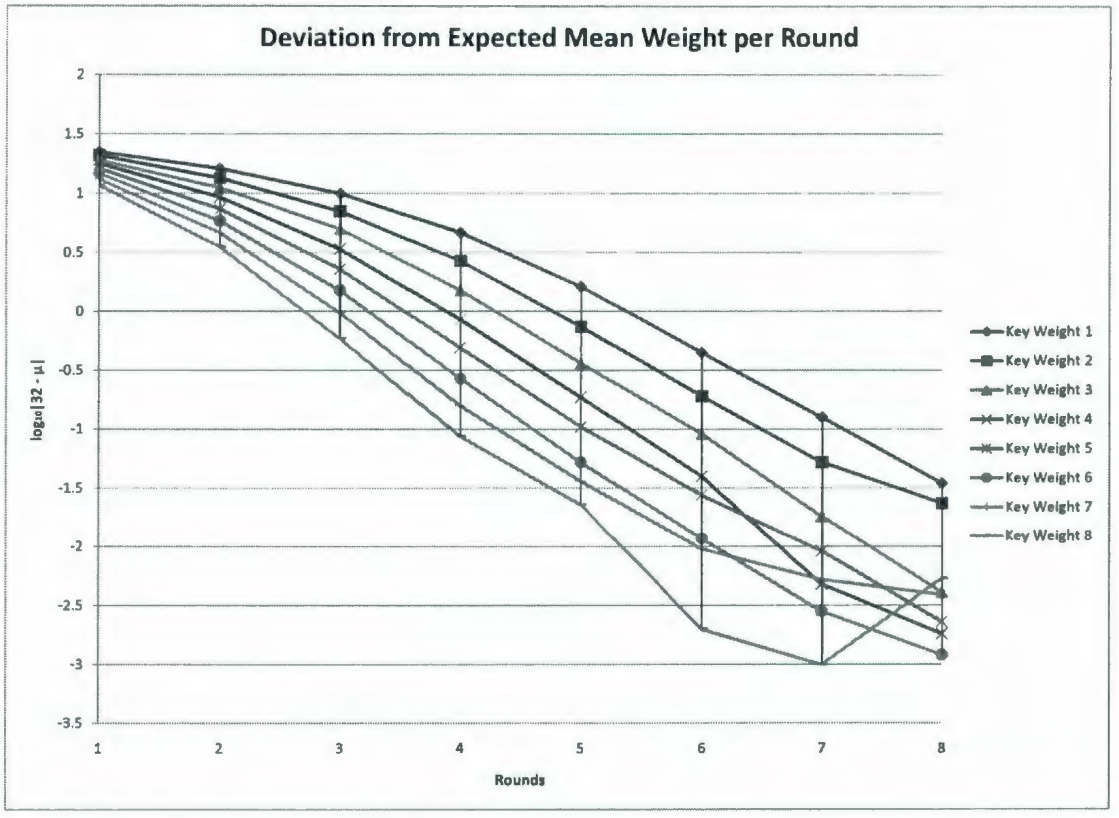

Figure 4.1: Deviation of Output Weight for Low Weight Keys over 8 Rounds

less) over 8 rounds. If fact, with the given input conditions, it can be seen that the mean does not get to within two one–thousandths to the ideal of 32 until the sixth round of the test using key weights of 8, which would be easily distinguishable from random.

Another test to see if the cipher output looks random is to do a "goodness–of–fit" test. For this, the chi–squared (χ^2) test is chosen to test if the distribution of the output weights matches the binomial distribution as would be expected in any strong cipher. We are able to use this test since all inputs, though constrained, are chosen randomly (satisfying the requirement of independence) and the outcomes are mutually exclusive.

The first step in the χ^2 test is to state a null hypothesis. In this case, the null

		Rounds							
		1	2	3	4	5	6	7	8
Key Weight	1	9.80562	15.8245	21.9752	27.3286	30.3768	31.5483	31.8731	31.9652
	2	11.3139	18.4168	24.8688	29.3186	31.2509	31.8091	31.948	31.9767
	3	12.8101	20.7126	27.0043	30.4733	31.6386	31.9094	31.9818	32.0041
	4	14.3018	22.7463	28.5732	31.1414	31.8139	31.9601	31.9952	32.0018
	5	15.8009	24.5132	29.6916	31.5112	31.8963	31.9727	31.9909	31.9977
	6	17.2943	26.0585	30.4913	31.7317	31.9477	31.9882	31.9972	31.9988
	7	18.7754	27.3571	31.0336	31.8431	31.9641	31.9904	32.0005	32.0039
	8	20.2617	28.4782	31.4082	31.9131	31.9769	31.998	32.001	32.0054

Table 4.3: Average output weight (μ) with maximum input weight of 6 over 5000000 encryptions

hypothesis is

$$H_0: \text{ The sample is from the binomial distribution } b(y; 65, 0.5) \tag{4.19}$$

where $b(y; 65, 0.5)$ is the binomial distribution with 65 trials and a 50% probability of success (i.e., obtaining a '1' bit as opposed to a '0' bit) in each trial. Simply put, the weight of the output data from the CIKS–1 cipher will fit the binomial distribution.

The degrees of freedom, ν, for the test are the number of possible outcomes (i.e. number of different Hamming weights for the output) minus one. Therefore, in this case there are $\nu = 64$ degrees of freedom. For the test, a probability of error threshold of $\alpha = 0.05$ is chosen, making unlikely outlier cases where the result is a fit on chance alone. Using these values the critical value for the test, $\chi^2_{\alpha,\nu}$, is calculated to be $\chi^2_{0.05,64} = 83.675$. This value is used to determine if the sample data is fit to the expected data. For this, the χ^2 value must be calculated, using the formula,

$$\chi^2 = \sum_{i=1}^{J} \frac{(O_i - E_i)^2}{E_i} \tag{4.20}$$

where O_i is the observed frequency, and E_i is the expected frequency of the event.

From Equation 4.20, it can be seen that the larger the deviation between the observed and expected values, the larger the χ^2 value. The critical value is used to determine what result is too large; thus, if the calculated value exceeds the critical value, the null hypothesis, H_0, is rejected.

Up to five rounds, the CIKS–1 output for each key weight up to six and plaintext inputs of weights less than or equal to six gives large values of χ^2. This indicates there is no fit to the binomial distribution for this constrained use of the cipher. In fact, this property holds for CIKS–1 as long as the given limits to Hamming weights are used, and is the basis for the attack on the cipher presented in the next section.

4.3 Proposed Attack

As shown, CIKS 1 depends almost entirely on the subkeys to contribute to the growth of Hamming weight for the data. In fact, when low weight keys are used with low weight input data, it is possible to distinguish between a random set of bits that conforms to the binomial distribution and the output of the cipher. The analysis reveals that the set of low weight subkeys, with weight of around six or less, results in a weight distribution that is easily distinguished as non random using the χ^2 test when we constrain the weight of the input to about six or less as well. This set of keys should be considered to be weak keys.

Exploiting this weakness, an attack can be mounted on a reduced version of the cipher, limited to six rounds, to extract information about the first subkey. A subkey is guessed for the first round. Next, a large set (one million in the case of the six round attack) of random values for the LHS data after P_1 is generated. The lower the Hamming weight for each of these, the better for exploiting weak keys. However we need enough possible plaintext to complete the test, and therefore a Hamming weight of six or less is used. Each value is used as an input to P_4 to permute the guessed subkey. The permuted subkey is then run backwards through P_2 to generate a value for the RHS input. This input is then used to as the control vector data when the original generated LHS data is run backward through P_1 to produce an input value for the LHS.

The values for LHS and RHS are then encrypted over six rounds. The idea is to produce a value for the right hand input which is close to the actual subkey being attacked. A correct guess nullifies the effect of the subkey (thus requiring no constraint on its Hamming weight) in the first round, thus leaving only the effects of the last five rounds. If the keys used in rounds two through six are low weight, the χ^2 test shows the weight distribution of the output does not match the binomial distribution. Pseudocode for the attack (using notation from Figure 3.8) is given in Figure 4.2 and

```
Choose the number of required inputs, k
for all possible 2^32 subkeys in round 1 do
  for y = 1 to k do
    Create a random value for LHS after P_1, L*, with weight i ≤ 6
    Form Control vector V_L
    Run subkey through P_4, call result PSK
    Form control vector V'
    Run PSK backward through P_2 to get RHS
    Form control vector V
    Run vector L* back through P_1 to get LHS
    Encrypt RHS and LHS over 6 rounds, record resulting weight
  end for
  Calculate the χ^2 value for the results
end for
Compile a list of weights with poor fit to the Binomial Distribution
```

Figure 4.2: Proposed weight based attack on CIKS 1 cipher

the full code can be found in Appendix B.

Test results show the guessed subkey does not have to be exact to give useful results. Guessed keys with the same weight as the actual key and those with similar weights produce similar results to guessing the actual key. With this knowledge, the search space for the first subkey can be reduced based on the result of the χ^2 tests.

Table 4.4 is an example of results obtained with the attack[1]. This example used an actual subkey of all zeros and only 100 randomly guessed subkeys for each of the possible weights. This test shows that the search area can be reduced to keys with a similar weight within two of the correct key. It is important to note that although this test used a low weight first round key, this is not required. The attack only requires low weight subkeys in the subsequent rounds to be successful.

This attack on the 6 round reduced cipher requires approximately 2^{20} random data inputs for each of the 2^{32} possible subkeys tested, giving it a total time complexity of 2^{52} encryption operations. This is an improvement over the 5 round attack with time complexity of $2^{65.7}$ presented in [33]. However, that attack makes no assumptions

[1]Calculated using Microsoft Excel 2002 which has the limit on the maximum value in a cell of $1.79769313486232 \times 10^{308}$.

Weight Difference	χ^2
0	$> 1.79769313486232 \times 10^{308}$
1	$> 1.79769313486232 \times 10^{308}$
2	$> 1.79769313486232 \times 10^{308}$
3	118.52
4	69.33

Table 4.4: Test results for low weight attack

about the cipher keys, whereas the attack presented here requires that all but the first round key be low weight. The probability of choosing a single 2^{32} key at random and getting a key with Hamming weight of six or less is given by

$$\begin{aligned} P(\text{Low Weight Key}) &= \sum_{i=0}^{6} \frac{\binom{32}{i}}{2^{32}} \\ &= 2.68 \times 10^{-4}. \end{aligned} \tag{4.21}$$

The probability of choosing five keys with such a low Hamming weight would be far lower. However, not all key scheduling methods guarantee independence. For example, the DES key schedule specifies that a single 56–bit key is chosen for the cipher and then each 48–bit subkey is chosen as a subset of that key [36]. If a similar key schedule was used with CIKS-1, a single low weight key would result in a very high probability of weak subkeys throughout the cipher. If a weak (i.e. low Hamming weight) key is chosen in this implementation, then the entire cipher is compromised until the next key change, as all of the subkeys are derived from the main key. Even more complex key schedules can result in a significant probability of weak keys.

4.4 Conclusions

Due to the choice of primitives with limited effect on the Hamming weight of the cipher data, the CIKS 1 cipher depends heavily on the weight of subkeys to produce change in the data weight. It has been shown that the DDPs, fixed permutations and fixed rotations have no effect of the weight of data as it progresses through the cipher. The exclusive or operation depends heavily on the weight of the input data. By definition it changes the weight of the output half of the time based on random data, but when using low weight data, this probability decreases. Finally, the two bit parallel additions are shown to preserve the input weight with a probability of $\frac{3}{8}$.

We conclude that there are a class of low weight keys which should be considered weak keys for the cipher. Analysis of weight propagation through the cipher shows that when keys with weight of around six or less are used as subkeys, we can easily detect them by constraining our input data weight and comparing the output to the binomial distribution using the χ^2 test. Using this fact, an attack is proposed to distinguish the first subkey by dramatically reducing its entropy. Testing has been done on the attack, the results of which are shown to reduce the search area for the first subkey to within a weight of two from the actual weight. In the next chapter we look at another attack on the cipher which involves the Hamming weight of the cipher data.

Chapter 5

Differential Attack on CIKS–1

In this chapter a non–traditional differential attack on CIKS–1 is presented. First there is a look at the analysis of the cipher's resistance to conventional differential cryptanalysis. The focus then shifts to applying differential cryptanalysis in a new way. The attack deals with the weight differential rather than the actual specific differential in the data itself. Using this technique the data complexity for differential cryptanalysis on the CIKS 1 cipher is reduced to approximately 2^{51} chosen plaintext/ciphertext pairs.

5.1 Previous Differential Analysis of CIKS–1

In the original paper on CIKS 1 [2], Moldovyan and Moldovyan make the claim that the number of plaintext pairs required for a differential attack on the cipher is approximately 2^{64}. To determine this number they examine the maximal case of a non–zero difference passing through the two $P_{32/80}$ permutations on the right side of the cipher. Based on the analysis of the permutation blocks given, the maximal case corresponds to the case with differences using one active bit.

According to the authors' analysis, the probability that a difference passes through the permutations, p_{DA}, for a one–bit difference is $p_{DA}(1) = 2^{-5}$. However, due to the

implementation details, it is stated that maximum probability of the difference passing through the $P_{32/80}$ permutations on the RHS of the ciphers is $p'_{DA(max)} - 2p_{DA}(1) = 2^{-4}$, where the added factor of two accounts for the non-uniform distribution of the control vectors V' and V''.

With this approximation and considering that the RHS data passes through two $P_{32/80}$ permutations, the maximum probability of the difference surviving one round of the cipher is $p_{DA(f-1)} = [p'_{DA(max)}]^2 = 2^{-8}$, where f is the number of times the difference passes through the RHS permutations. The best case analysis is done for the difference surviving all eight rounds of the cipher, focusing on the RHS permutations which the difference will pass through four times. Therefore, we have a probability of the difference propagating through the cipher of $p_{DA(f=4)} = [p_{DA(f-1)}]^4 = 2^{-32}$. With this, the authors use the approximation $N_{DA} \approx (p_{DA(f=4)})^{-2}$ to determine that the number of plaintext/ciphertext pairs, N_{DA}, required for a differential attack is approximately 2^{64}.

Since this analysis is done focusing on only the effects of the RHS permutations P_2 and P_6, it does not account for the effects of the LHS of the cipher. As such, the analysis is considered to be a best case in favour of the attacker; the authors conclude that the cipher is resistant to differential attack. The following section looks at applying the idea of differentials to a difference in weight and how these differences propagate through CIKS–1.

5.2 Data–Dependent Permutations and the Propagation of Differences

Being the main element of CIKS–1, it might be expected that the DDPs would play a major role in the propagation of differences through the cipher. In fact, other than the parallel addition, DDPs are involved in all operations of the cipher data. On

both sides of the cipher, data is scrambled by the permutations and the key itself is scrambled before being added.

Obviously, if the same control vector used on a DDP is used more than once, then the data in each instance will be permuted the same way. When the control vector does change, the likelihood of new differences in the output depends on the differences at the input. When there is only a one bit difference in the control vectors, the input to the $P_{2/1}$ permutation that the particular CV difference affects determines if there is a resulting difference in the output. The swap at this site is only noticeable if the input bits are different. Thus, there is a $\frac{1}{2}$ chance that the output will be unchanged by the control bit difference in the $P_{2/1}$ permutation. In general, the probability that the permutation's output will remain unchanged by the DDP is 2^{-n}, where n is the number of different bits in the control vector.

The case where there is a difference in the input as well as in the CV is more complex. When the CV difference is the control bit for a $P_{2/1}$ permutation where there is also a difference in one of the input bits, there is no new difference introduced in the data. In fact, if both input bits are changed, the output will have no new difference. These cases actually increase the chance of a given difference surviving a DDP. In our analysis, a one–bit difference input into a $P_{32/80}$ with a control vector containing two differences has approximately a 28% probability of retaining a one–bit difference at the output.

5.3 Analysis of Differentials

If an analysis of the cipher is done not on strict differentials, but the Hamming weight of differentials as they pass through the cipher, it is possible to construct an attack which has much lower complexity than the one in [2]. Three input difference

weights were examined for relationships to four output difference weights of interest. These are given in Table 5.1 for one round where $(wt(\Delta L_{i-1}), wt(\Delta R_{i-1}))$ and $(wt(\Delta L_i), wt(\Delta R_i))$ are the Hamming weights of the differences in the left and right halves of the inputs and outputs, respectively.

Differentials ($wt(\Delta L_{i-1})$, $wt(\Delta R_{i-1}) \rightarrow wt(\Delta L_i)$, $wt(\Delta R_i)$)	Probability
$(0,1) \rightarrow (1,2)$	$2^{-3.4}$
$(0,1) \rightarrow (1,1)$	$2^{-1.83}$
$(1,0) \rightarrow (0,1)$	$2^{-7.25}$
$(1,1) \rightarrow (1,0)$	$2^{-13.7}$
$(1,1) \rightarrow (1,1)$	$2^{-13.7}$
$(1,1) \rightarrow (1,2)$	$2^{-7.75}$

Table 5.1: Frequency of occurrence of transitions of interest with random keys.

Examining the case of $(wt(\Delta L_{i-1}) = 1, wt(\Delta R_{i-1}) = 1)$, we see that the right side difference can appear either once or twice in CV V (see Figure 3.8). Taking into account both cases, the probability of the one–bit difference surviving P_1 is 12.5%. Again, depending on where the one-bit difference in the left side occurs, it can appear in V' either two or three times. Thus, the probability of the one bit difference on the right side surviving P_2 is approximately 8.1%.

When the key is bit wise exclusive ored with the right side data, there are many possible cases to examine. Any case where the absolute difference of $wt(\Delta R)$ and $wt(\Delta K)$ (where $wt(\Delta R)$ and $wt(\Delta K)$ are the data and key inputs to the key addition) equals 1 could result in the output of the key additions being different by one bit. To simplify the analysis we consider only the most dominant of these cases, $(wt(\Delta R) = 1, wt(\Delta K) = 0)$ and $(wt(\Delta R) = 1, wt(\Delta K) = 2)$, and get a likelihood of the one bit difference surviving of approximately $2^{-5.7}$. Note that P_6 acts similarly to P_2. Overall, the probability of a $(wt(\Delta L_{i-1}) = 1, wt(\Delta R_{i-1}) = 1)$ difference leading to a $(wt(\Delta L_i) = 1, wt(\Delta R_i) = 0)$ is $2^{-13.7}$.

The other differential probabilities can be calculated in a similar way. Note that in the cases where the difference only appears on one side of the cipher, many of the

cipher's elements do not affect the difference on the other side. These one round differentials can be chained together to get an overall differential for the cipher. The notation $(wt(\Delta L_{i-1}), wt(\Delta R_{i-1})) \rightarrow (wt(\Delta L_i), wt(\Delta R_i))$ is used to represent the transition in the weight difference of each side of the input and output pair. If we use the 7–round chain of differentials $(0,1) \rightarrow (1,1) \rightarrow (1,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (1,0) \rightarrow (0,1) \rightarrow (1,1)$ there is a probability of $2^{-47.39}$ that we get the final differential assuming the differences from round to round can be considered to occur independently. Figure 5.1 shows the probability of transitions between all of the differentials of interest listed in Table 5.1.

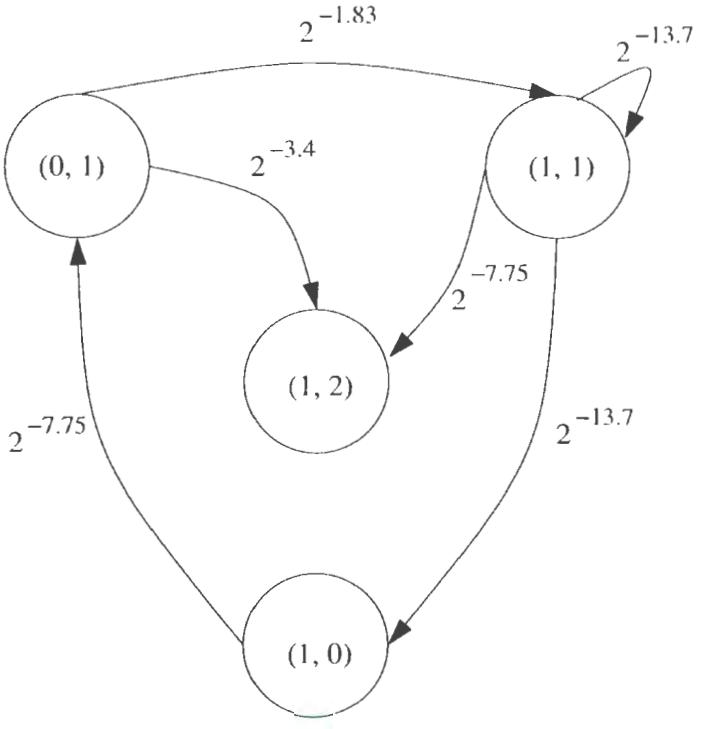

Figure 5.1: Probabilities of transitions of interest.

count = 0
r = Number of rounds for attack
Define state(w,x) = $(wt(\text{left}(w) \oplus \text{left}(x)), wt(\text{right}(w) \oplus \text{right}(x)))$
s_0 := initial state of interest
s_{r-1} := state of interest after $r-1$ rounds
Choose the number of required differential pairs, k
for $i = 0$ to $k-1$ **do**
  Initialize $X_{0,0}$ and $X_{1,0}$ such that state$(X_{0,0}, X_{1,0}) = s_0$
  Encrypt $X_{0,0}$ for r rounds to get $X_{0,r}$
  Encrypt $X_{1,0}$ for r rounds to get $X_{1,r}$
  for all Possible 2^{32} subkeys, j $0 \leq j \leq 2^{32}-1$ **do**
    Decrypt $X_{0,r}$ for one round using j to get $X_{0,r-1}$
    Decrypt $X_{1,r}$ for one round using j to get $X_{1,r-1}$
    if state$(X_{0,r-1}, X_{1,r-1}) = s_{r-1}$ **then**
      Increment $count[j]$
    end if
  end for
end for
Result is the set of all x such that $count[x] = \max_j (count[j])$

Figure 5.2: Proposed differential attack on CIKS–1 cipher

5.4 Proposed Attack

As shown in the last section there are certain differentials with a high probability of occurrence. The transitions $(0,1) \rightarrow (1,1)$, $(1,0) \rightarrow (0,1)$ and $(1,1) \rightarrow (1,2)$ are the most attractive. When chaining together multiple rounds these transitions are reused as frequently as possible to keep the overall probability high. Although the $(0,1) \rightarrow (1,2)$ and $(1,1) \rightarrow (1,2)$ transitions have a high probability, $(1,1) \rightarrow (1,2)$ is only useful at the end of a chain and $(0,1) \rightarrow (1,2)$ has a lower probability than the alternate branch from $(0,1)$, $(0,1) \rightarrow (1,1)$.

To attack the cipher, we first choose a chain of states with a relatively high probability of success to use. For example, to attack a six–round version, a five round chain would be required. The chain of weight difference $(1,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (1,0) \rightarrow (0,1) \rightarrow (1,1)$ could be used with a probability of approximately $2^{-31.86}$. A table of high probability chains versus numbers of rounds is given in Table

5.2.

Rounds	Differential Chain	Probability
3	$(1,0) \rightarrow (0,1) \rightarrow (1,1)$	$2^{-9.58}$
4	$(1,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (1,2)$	$2^{-17.33}$
5	$(1,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (1,0) \rightarrow (0,1)$	$2^{-31.03}$
6	$(1,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (1,0) \rightarrow (0,1) \rightarrow (1,1)$	$2^{-32.86}$
7	$(1,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (1,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (1,2)$	$2^{-40.61}$
8	$(0,1) \rightarrow (1,1) \rightarrow (1,0) \rightarrow (0,1) \rightarrow (1,1)$ $\rightarrow (1,0) \rightarrow (0,1) \rightarrow (1,1)$	$2^{-48.09}$

Table 5.2: Differential chains for attacking different numbers of rounds

Next, the number of differential pairs that will need to be generated is determined based on the probability of detecting the chosen outcome. In the six round example above, the output difference would be expected once in every 8 billion encryptions. In comparison, if the output difference weight were to occur randomly, the likelihood of output difference $(1,1)$ would be $(\frac{32}{2^{32}})^2 = 2^{-54}$. Hence, several times more than 8 billion encryptions would be required to clearly distinguish the occurrence of the expected difference.

When attacking r rounds of the cipher, for each of the chosen number of differential pairs, we encrypt the two inputs, $X_{0,0}$ and $X_{1,0}$, over r rounds labeling the results $X_{0,r}$ and $X_{1,r}$. These two values are then decrypted for all 2^{32} possible subkeys of round r and labeled $X_{0,r-1}$ and $X_{1,r-1}$. For each result corresponding to a subkey, a counter for the subkey is incremented if the weight of state($X_{0,r-1}$, $X_{1,r-1}$) matches the expected value.

When all of the differentials have been processed for all of the possible subkeys for round r, the subkey with the highest count is the most likely to be the actual subkey. The full pseudocode for the attack is given in Figure 5.2 and the implementation code can be found in Appendix C. The next section presents an experimental verification of this attack.

5.5 Experimental Verification

To verify the attack experimentally, an attack has been implemented on a 3 round reduced version of the cipher. The chain used for the attack was $(1,0) \rightarrow (0,1) \rightarrow (1,1)$, which has a calculated probability of occurrence of approximately $2^{-9.58}$. The expected desired output differential (1, 1) should appear in the result for the correct key approximately once in every 540 attempts. Therefore, the number of plaintext pairs for the test is chosen to be 10000, implying the expected state for the actual subkey will occur approximately 19 times.

The test was run using the actual key, 32 keys with a one bit difference from the actual key, and 10000 randomly generated keys. All of the keys with a one bit difference from the actual key resulted in the expected output state zero times, making them easily discountable as the actual key. The set of random keys resulted in the desired output differential between zero and two times, with the distribution given in Table 5.3. However, the actual key returned the expected difference 22 times, making it easily distinguishable from the other possible keys.

Score	Frequency
0	9760
1	238
2	2

Table 5.3: Frequency of occurrence of desired differential with random keys.

5.5.1 Attack Complexity

Although this has only been implemented on a 3 round version of CIKS 1, it could be extended to the 6 round version using the example chain given in the previous section. The probability for that chain is approximately $2^{-31.86}$, meaning one would expect the desired output difference once in every 2^{32} tries. To be distinguishable, one would want the expected differential to occur approximately 8 times, giving you a

data complexity of approximately $2^{32} \times 2^3 - 2^{35}$ plaintext/ciphertext pairs. The time complexity of the attack includes $2^{35} \times 2 = 2^{36}$ encryption operations and $2^{35} \times 2 \times 2^{32} = 2^{68}$ partial decryptions.

In fact, it is theoretically possible to extend this attack to the full cipher. For this extension, the seven round differential chain $(0,1) \rightarrow (1,1) \rightarrow (1,0) \rightarrow (0.1) \rightarrow (1,1) \rightarrow (1,0) \rightarrow (0,1) \rightarrow (1,1)$ could be used. This chain has a probability of occurrence of $2^{-48.09}$ and therefore is expected to give the desired output approximately once in every 2^{48} tries. Again, to be distinguishable from random noise, we would want to see the expected output difference approximately eight times, giving a data complexity of approximately $2^{48} \times 2^3 - 2^{51}$ plaintext/ciphertext pairs to recover the final round subkey. The time complexity of this attack includes $2^{51} \times 2 = 2^{52}$ encryption operations and $2^{51} \times 2^{32} = 2^{84}$ partial decryption operations. The remaining subkeys can then be found by stripping off the last round and implementing the attack again on the remaining rounds.

5.6 Conclusion

In the original paper for the CIKS 1 cipher, the authors' analysis of the possibility of a differential attack on the cipher showed that it would have a data complexity of 2^{64}. Instead of the usual approach to this attack, a non–traditional approach of exploiting the weight of differences in the cipher is given in this work. DDPs are shown to have a probability of passing one–bit differences introduced at their input to their output 28% of the time. This knowledge is then used to create a set of differences that can be used to retrieve key data from CIKS 1.

Using these differentials, an attack has been proposed to gain key information from the last round. As a proof of concept, the attack has been implemented on a 3–round version of the cipher. This attack showed that the actual key can be

easily distinguished from both random keys and keys with one bit different than the actual. This attack can be extended to six rounds with a data complexity of approximately 2^{35} plaintext/ciphertext pairs and time complexity of approximately 2^{36} encryption operations plus 2^{68} partial decryption operations. The attack can be further extended again to the full eight rounds of the cipher with a data complexity of 2^{51} plaintext/ciphertext pairs and a time complexity of 2^{52} encryption operations plus 2^{84} partial decryption operations.

Although two attacks on the CIKS 1 cipher have been presented, one simple way to increase the security of this cipher would be to increase the number of rounds used. This would decrease the likelihood of being able to carry out the attacks in this cipher. Since the cipher can still have value, in the next chapter we look at the implementation of CIKS 1 in software.

Chapter 6

Software Implementation of CIKS–1

In the original paper [2], the authors state that the CIKS 1 cipher is designed for hardware implementation. Their original goals were to implement a cipher that is free of pre computations (i.e., lacking a key schedule requiring the generation of subkeys) and efficient to implement in hardware. Unfortunately, such specifications can also lead to algorithms which are inefficient to implement in software. which is the case with CIKS 1.

In [32], the authors propose an efficient software implementation of the DDP based Cobra family of ciphers. However, this implementation depends heavily on the addition of a DDP instruction to general purpose processors. The focus of this chapter is to investigate an alternate method of implementing DDPs, and thus CIKS 1, efficiently in software. Such knowledge will be useful in implementing other DDP–based ciphers or a strengthened version of CIKS–1.

6.1 Implementing CIKS–1 in Software

Each component of CIKS–1 is designed with efficient hardware implementation in mind. The fixed permutations and rotations are easily implemented using simple electrical connections between the components. An addition modulo 2^2 is chosen to

avoid the gate propagation delay associated with addition components using larger words, such as modulo 2^{32}. The key addition is a simple 32–bit exclusive or. Finally, the main non linear component, the DDP, is designed such that the delay is proportional to the number of layers in the permutation.

The majority of the operations employed in CIKS–1 operate at a sub word level; most operations are at the bit level. While this works quite well when implementing custom hardware, a large portion of software is written for general purpose processors. These processors are usually optimized for instructions that process full words of data at a time (32 bit or 64 bit words for current mainstream processor technologies). In order to manipulate data n bits at a time, where n is less than the processor word size, many programming languages require overhead operations such as masking, shifting and temporarily storing bits in order to affect only those bits of interest.

One strategy to avoid the overhead of these operations would be to use arrays to contain the data, thus allowing direct access via the array index. This can be accomplished in a language such as C/C++ by storing each bit in a single *char*, a single byte variable, which is stored in an array. This system has an obvious advantage for bitwise data access, but it has the equally obvious storage inefficiency since an eight bit variable type is being used to hold a single bit of data. Also, some operations that can be done efficiently on a word basis (e.g. word–oriented exclusive–or) has to be done for each individual bit which is extremely inefficient. In the following sections we look at another method for efficient implementation of CIKS–1 which overcomes the single bit access problem while avoiding wasted memory.

6.2 Bitslice Implementation of Ciphers

In [37], Eli Biham introduced a new fast software implementation of DES. Unlike other implementations of DES, this one approached the problem from the point of

view of trying to fully utilize the entire word length of the instruction set for the processor. To achieve this, an n bit processor is treated as n one bit processors. Therefore, a modern 64 bit processor is used as 64 parallel one bit processors, performing the same operation 64 times simultaneously. This method, referred to as bitslicing, allows for single bit operations to be implemented more efficiently on general purpose processors by operating on large numbers of inputs simultaneously.

To implement DES at a bit level, Biham starts with the cipher represented as a series of logic gates. It is a straight forward translation to obtain a set of boolean equations for each component of the cipher. Using bitwise operators, these equations are easily implemented in code. Since a single bitwise operator will by definition apply the same operation to each bit in the word independently, this implementation can be used to encrypt (or decrypt) multiple plaintext inputs simultaneously.

To be able to use this version of the cipher, the data used needs to be stored in a non standard arrangement. The normal way of treating data in software is to store it in variables with each one holding a single piece of information. In order to use the bitslicing technique, n pieces of data (where n is the n bit size of the word form the processor) are grouped together into an array. That array can then be transposed such that each value accessed by index i now holds the ith bit for each input. Arranged in this manner, the data can be input into the new version of DES, allowing for parallel encryption (or decryption). To get the final results from the output of this code, it must be transposed again into the standard arrangement.

Biham shows in the paper that this implementation of DES provides a speed up of approximately 1.6 times over Eric Young's *lib DES*, a standard fast implementation of the cipher, when run on a 300MHz Alpha 8400 processor. He also notes that this technique can be extended to other ciphers. Biham states the approach can be useful for most ciphers which do not take advantage of the full word size of the processor. He notes that the cipher should have no complex operations, such as multiplication or

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 \\ c_0 & c_1 & c_2 & c_3 \\ d_0 & d_1 & d_2 & d_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_0 & b_0 & c_0 & d_0 \\ a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \end{bmatrix}$$

Figure 6.1: Example transposition of input for bitslice implementation

large substitution boxes, whose implementation would require a much larger number of instructions compared to a straight forward implementation. Since the CIKS–1 cipher meets these criteria, a bitslice version has been implemented and is presented in the next section.

6.3 Bitslice Implementation of CIKS–1

The simplicity of the components used in CIKS–1 make it an ideal candidate for a bitsliced implementation. The cipher is mainly comprised of bit permutations, either fixed or data–dependent. The only other operations are addition modulo–2^2, rotations and an exclusive or key addition, all of which can be reduced to simple gate level representations. The next sections discuss the implementation of each operation for the bitsliced version of CIKS–1.

6.3.1 Preparing the Input

The bitslice method requires the user to prepare the input into an unconventional format. For n–bit bitslice implementation, the user is required to create an n by m matrix of inputs where n is the number of simultaneous pieces of data to be encrypted and m is the size of the data in bits. This matrix is then transposed such that row i of the transposed matrix will contain bit i of each original input. Figure 6.1 illustrates an example of the transposition where $n = m = 4$.

The success of this method is highly dependent on this transposition being done

in an efficient manner. If the amount of time required to transpose the data is larger than that of the actual speedup, then bitslicing the algorithm provides no benefits. In [38], the author provides an efficient method for transposing a 32–by 32 bit matrix by breaking the matrix into smaller sub matrices that can be handled more effectively by the processor's registers; transposing those sub matrices before combining them to form the transposed 32 by 32 matrix.

The algorithm starts by dividing the 32 by 32 matrix into 4 smaller matrices of size 16 by 16 bits. These matrices are transposed such that the second half (16 bits) of the first 16 words is swapped with the first half of the second 16 words. The results of this swap are then subdivided into 16 4 by 4 bit matrices and grouped such that bits 8–15 and 24 31 of the first eight words are swapped with bits 0 7 and 16 23 with the second group of eight words. Similar swaps are done on the third and forth groups of words. This pattern is repeated with smaller sub matrices until the entire 32–by 32 bit matrix is transposed. In our tests, the 32–bit bitslice version uses a 32 by 32 bit matrix. The 64–bit version uses a 64 by 32 bit matrix that is transposed to a 32 by 64 bit matrix.

6.3.2 Data–Dependent Permutations

To implement the CIKS–1 DDPs, larger DDPs are broken down into their smallest single element, the $P_{2/1}$ block, a simple data controlled swap of two bits. This primitive can be modeled as two parallel multiplexers with equations

$$x_0 = (cv \wedge x_0) \vee (\neg cv \wedge x_1) \tag{6.22}$$

and

$$x_1 = (\neg cv \wedge x_0) \vee (cv \wedge x_1). \tag{6.23}$$

By the definition of the cipher, if the control vector bit is zero then the inputs are swapped, whereas a control vector input bit of one allows them to pass through unchanged.

As we can see in Figure 3.6, the DDPs are constructed of layers of the smaller $P_{2/1}$ permutations. Using Equations 6.22 and 6.23 to form $P_{2/1}$ permutations for each layer, all that is required to complete the larger permutations is to connect the layers via the fixed permutations. Using bitslicing, the $P_{32/80}$ permutation is implemented using equations 6.22 and 6.23 using bitwise operations on words, thereby completing n permutations in parallel for the n bit bitslice implementation.

6.3.3 Modulo–2^2 Additions

The modulo 2^2 additions can be expressed as a series of three equations:

$$c = x_0 \wedge y_0, \tag{6.24}$$

$$x_0 = x_0 \oplus y_0 \tag{6.25}$$

and

$$x_1 = x_1 \oplus y_1 \oplus c \tag{6.26}$$

where x_0 is the Least Significant Bit (LSB) and x_1 is the Most Significant Bit (MSB) of the LHS, y_0 is the LSB and y_1 is the MSB of the RHS and c is the carry bit.

First, the carry bits can be calculated in parallel by using bitwise "and" on the words holding the LSBs. Then, the final values for the LSB and MSB can be calculated as an exclusive or of the respective bits together; the MSB also being exclusive ored with the carry bit. There are 16 parallel additions, so these operations are repeated.

6.3.4 Fixed Permutations, Rotations and Key Addition

In our original implementation of CIKS–1, the fixed permutations were very computationally expensive since the individual bits need to be isolated within the data and moved to different position. In the bitslice implementation of CIKS 1, each value in the array contains a processor word which is comprised of a group of bits from a single position in multiple pieces of data. Therefore, in order to swap bits it is only necessary to change the order of the groups. This also holds for rotations, making each of these operations very efficient when operating on many blocks in parallel.

Key addition for CIKS 1 is a simple exclusive or of the key (permuted via a DDP) with the RHS of the data. In the bitslice implementation, this is achieved as 32 exclusive–ors of the groups of bits.

6.4 Experimental Results and Discussion

The bitsliced implementation of CIKS- 1 was compared against two other implementations of the cipher. The first uses the *bitset* class in the C++ Standard Template Library (STL). This class makes it convenient to access individual bits of the cipher data, but it is not written to maximize the usage of the processor's instruction set. The second implementation stores each bit in a *char* variable. This implementation allows for easy access to each bit, but uses eight times more storage than required for the data and is only effective when operating on a per–bit basis.

For the comparison of each implementation, 320 million randomly generated plaintexts are enciphered. The number is chosen as a multiple of 32 since the word sizes for the processor being tested are multiples of this value (32 bit and 64 bit). The factor of 10 million is chosen to be long enough to allow for measurement of differences between tests without making the tests too long to run.

In the case of the bitsliced implementation of CIKS–1, the required number of

plaintexts (32 or 64) are generated and added to an array. The array is transposed, the data encrypted and then the array is transposed again to get the ciphertexts. For the other two implementations, each plaintext is encrypted as it is generated. Each of the tests was run on an Intel Core 2 Duo 2.4GHz 64 bit processor with 2GB of RAM. The 32 bit version was compiled using only 32 bit instructions and the 64 bit version was compiled to use 64 bit instructions by changing the target architecture use by the *GCC* compiler. The total processing time is measured using the POSIX command *time*. This command gives the total time for the command to complete, as well as the breakdown of the time used by the actual program and the system overhead.

The initial bitslice version was implemented using 32 parallel encryptions, which was considered portable to the majority of processors commonly available which use 32–bit words. The test consisted of 320 million encryptions through one round of the cipher using random inputs for both the data and key. For the *char* array and *STL bitset* based implementations, each input was generated, then encrypted before moving on to the next. For the bitsliced version, inputs were generated in groups of 32, transposed, encrypted and then transposed again to get the output.

A sample set of results from the testing for each version of the cipher is given in Table 6.1. The bitslice version of the cipher outperformed both the *STL bitset* and *char* array based versions. By bitslicing the cipher an overall speedup of 130 times is achieved over the next fastest method, the *char* array implementation. The time required to generate 320 million plaintexts was measured to be approximately 23 seconds. Taking this into account, the actual speedup of approximately 234 times is achieved by the 32 bit bitsliced version of the cipher over the *char* array based implementation.

Since 64 bit processors are becoming more popular and there are now 64–bit versions of the most common operating systems available, a 64 bit bitslice version

Test	Time
Generate Plaintexts	23s
STL bitset CIKS 1	126m 1s
char array CIKS 1	113m 41s
Bitslice (32 bit) CIKS 1	52s
Bitslice (64 bit) CIKS 1	39s

Table 6.1: Elapsed time for 320 million encryptions

was also implemented. The inputs to the cipher were 64 values (each of size 32-bits) for each of the left side, right side and key inputs stored in arrays. To prepare the inputs to be used in the bitsliced code, the previous algorithm from [38] was extended to fit the word size. To do this, the inputs were split into two 32 by 32 bit arrays. These arrays were individually transposed and then concatenated into an array of 32 words. Since the CIKS 1 cipher itself was written at the word level, its code did not need to be changed.

The 64 bit bitsliced version of CIKS 1 was tested by encrypting 320 million random plaintexts. The time to perform the encryptions was about 13 seconds less than that of the 32 bit bitsliced version. Since a two times increase in speed was not achieved, it is obvious that the overhead of formatting the inputs for the bitsliced ciphers is a large portion of the overall time. However, it is not large enough to negate the increase in speed of the implementation method as compared to the others.

6.5 Limitations of Bitslicing

Although these implementations show a significant improvement over the more conventional implementation techniques, they come with restrictions on their use. Since they require 32 (or 64) inputs to be encrypted at a time, they cannot be used for any of the standard modes of operation other than ECB or counter mode. Other modes require the output of one block into the next, however, these outputs are not available

when needed.

Other modes can be implemented using non standard methods. CBC can be used by using 32 (or 64) parallel chains by choosing the appropriate number of initialization vectors. This mode would be implementing 32 (or 64) standard CBC modes in parallel and CFB or OFB mode can also be implemented in a similar way.

6.6 Conclusions

The CIKS 1 cipher was designed for hardware and as a result utilizes many operations that are not well suited for general purpose processors. In this chapter, an implementation has been presented that overcomes these limitations. The bitsliced implementation of CIKS-1 has a speedup of approximately 234 times over the nearest competitor in tests run. When a 64 bit architecture is used, that speedup increases to approximately 425 times.

The bitsliced version of CIKS 1 is only compatible with two standard implementations of cipher modes, ECB and counter mode. However, it is possible to implement the cipher using CBC, CFB or OFB modes. In these cases, there are parallel implementations of the standard modes where the output of each is passed into the input of the next encryption for each implementation. Bitslicing can also be applied to other DDP–based ciphers to improve the efficiency of software implementation.

Chapter 7

Conclusions and Future Directions

The purpose of this thesis was to investigate the use of data dependent structures as an element in cryptographic ciphers. In particular, the CIKS-1 cipher was used for the research since it was written to primarily rely on DDPs for security. Two attacks on the cipher were presented as well as a look at implementing the cipher efficiently in software. The following is a summary of the conclusions of this research. As well, a list of future directions for this research is given.

7.1 Conclusions

The primary focus of the attacks presented in this thesis was the Hamming weight of the data as it was encrypted. Since the DDPs only permute the data bits, they have no effect on the weight of the data. Therefore, if the effect of the other components used is similarly limited, there arises the potential to attack the cipher by manipulating the data in a chosen plaintext attack. This approach is used in the weight based attack presented in Chapter 4.

CIKS-1 has very few components that change the Hamming weight of the data. The rotations and permutations have no effect and the parallel modulo 2^2 additions are shown to preserve the input weight of the data in $\frac{3}{8}$ of the cases. Therefore, the

cipher depends heavily on the exclusive or addition of the round subkeys to modify the weight of the data. Given this fact, keys with a low Hamming weight can be detected by choosing low weight inputs.

Therefore, it is concluded that there is a class of low weight keys which should be considered weak keys for the cipher. Through experimentation it is found that keys with weight around six or less can be detected by constraining the input data weight and comparing the resulting outputs to the binomial distribution through the χ^2 goodness–of–fit test. Testing shows that the search area for the first subkey can be reduced to within a weight of two from the actual weight on a 6 round reduced version of the cipher with a total time complexity for the attack of 2^{52} encryption operations.

In Chapter 5, a differential style attack is presented which focuses on the differential weight of the inputs rather than the individual differences of bits. The CIKS–1 DDPs are shown to pass a one–bit difference introduced at their input to their output approximately 28% of the time. Using this information the CIKS 1 round function is analyzed to create a set of likely differentials with which the cipher can be attacked.

These differentials are used to gain information about the subkey of the last round of the cipher. An experimental result is presented which can retrieve the last round subkey of a six round version of CIKS 1 with a data complexity of approximately 2^{35} plaintext/ciphertext pairs and time complexity of approximately 2^{36} encryption operations plus 2^{68} partial decryption operations. This attack is then extended to show theoretically that the entire eight round cipher could be attacked with a data complexity of 2^{51} plaintext/ciphertext pairs and a time complexity of 2^{52} encryption operations plus 2^{84} partial decryption operations.

Implementing these attacks in software can be inefficient due to the primitives chosen to satisfy the criterion of being fast in hardware. Many of the operations require swapping of individual bits which is not well suited for general purpose processors.

In testing and experimenting with the cipher, two versions were implemented, the first using the C++ STL class *bitset* and the second using byte arrays in which each element held a single bit of the cipher data. In Chapter 6, a third implementation method, bitslicing, was investigated.

Bitslicing overcame the inefficiency of bit level operations in general purpose processors by treating an n bit processor as n one bit processors. This allows bitwise operation on a specific bit of n inputs to be calculated in parallel. The bitsliced implementation of CIKS–1 presented has a speedup of approximately 234 times over the nearest competitor in the test runs when implemented on a 32 bit architecture. When compiled for a 64 bit bitslice architecture, the speedup increased to approximately 425 times.

Although DDPs can be efficiently implemented in both software and hardware, care needs to be exercised in the design of a cipher which uses them extensively. Specifically from the results presented, elements should be added that counteract the permutations' lack of effect on the weight of the cipher data. The following section provides suggestions for future directions for the study of DDPs.

7.2 Future Directions

This thesis focused solely on the use of DDPs in the CIKS–1 cipher because it relies so heavily on them for its security. Future study of the data dependent structures should be extended to other ciphers, specifically Spectr–H64 and the Cobra family of ciphers. These ciphers have DDPs of similar structure to CIKS–1 but were designed based on lessons learned from attacks on the original cipher. Specifically, a study of the weight propagation in these ciphers would help to understand if they have solved this weakness and if so how.

An attempt at redesigning CIKS–1 could also provide insight into avoiding the

weaknesses of the current design. Keeping in mind the primary goals of the cipher, to be secure while fast in hardware, replacements for fixed permutations, rotations and addition could be investigated. This could result in a catalog of components that are both fast when implemented in hardware and complementary to DDPs.

From the point of view of software implementation, the design of the permutations themselves could be reconsidered. Although the bitslicing technique worked well with CIKS 1, it has limitations for the mode of operation with which it could be used. It would be beneficial to have a DDP designed such that it can be efficiently implemented in software as well as hardware.

References

[1] R. L. Rivest, "The RC5 encryption algorithm," in *K. U. Leuven Workshop on Cryptographic Algorithms*, December 1994.

[2] A. Moldovyan and N. Moldovyan, "A cipher based on data dependent permutations," *Journal of Cryptology*, vol. 15, pp. 61–72, January 2002.

[3] N. D. Goots, A. A. Moldovyan, and N. A. Moldovyan, "Fast encryption algorithm Spectr H64," in *MMM-ACNS '01: Proceedings of the International Workshop on Information Assurance in Computer Networks*, vol. 2052 of *Lecture Notes in Computer Science*, (London, UK), pp. 275–286, Springer-Verlag, 2001.

[4] N. Sklavos, N. A. Moldovyan, and O. Koufopavlou, "High speed networking security: design and implementation of two new DDP-based ciphers," *Mobile Networks and Applications*, vol. 10, no. 1-2, pp. 219–231, 2005.

[5] S. Singh, *The Code Book*. Anchor Books, 1999.

[6] Society for Worldwide Interbank Financial Telecommunications, "Annual report 2006: Achieving more, together." Online at `http://www.swift.com`, 2006.

[7] Statistics Canada, "The daily: Canadian internet use survey." Online at `http://www.statcan.ca/Daily/English/060815/d060815b.htm`, August 2006.

[8] B. Schneier, *Applied Cryptography*. John Wiley and Sons, Inc., 1996.

[9] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, "The RC6^{TM} block cipher," in *First Advanced Encryption Standard (AES) Conference*, 1998.

[10] B. S. Kaliski Jr. and Y. L. Yin, "On differential and linear cryptanalysis of the RC5 encryption algorithm," in *Advances in Cryptology — CRYPTO '95* (D. Coppersmith, ed.), vol. 963 of *Lecture Notes in Computer Science*, pp. 171–184, Springer Verlag Berlin, 1995.

[11] D. R. Stinson, *Cryptography: Theory and Practice*. New York: Chapman & Hall/CRC, 2nd ed., 2002.

[12] C. E. Shannon, "Communications theory of secrecy systems," *Bell Systems Technical Journal*, no. 28, pp. 656 — 715, 1949.

[13] W. Diffie and M. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, vol. 22, pp. 644–654, Nov 1976.

[14] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[15] Bluetooth, "Bluetooth core V2.1 + EDR specification." Availible Online: `http://www.bluetooth.com`.

[16] S. Weatherspoon, "Overview of IEEE 802.11b security," *Intel Technology Journal*, 2000. Available Online: http://developer.intel.com/technology/itj/q22000/pdf/art_5.pdf.

[17] H. Feistel, "Cryptography and computer privacy," *Scientific American*, vol. 228, pp. 15–23, May 1973.

[18] H. M. Heys, "A tutorial on linear and differential cryptanalysis," *Cryptologia*, vol. XXVI, no. 3, pp. 189–221., 2002.

[19] N. I. of Standards and Technology, "Advanced encryption standard (AES) development effort." Online at `http://csrc.nist.gov/archive/aes/index.html`, 2001.

[20] National Bureau of Standards, "DES modes of operation, FIPS PUB 81," 1980.

[21] L. R. Knudsen, "Contemporary block ciphers," in *Lectures on Data Security, Modern Cryptology in Theory and Practice, Summer School, Aarhus, Denmark, July 1998*, vol. 1561 of *Lecture Notes in Computer Science*, (London, UK), pp. 105–126, Springer-Verlag, 1999.

[22] J. Kam and G. Davida, "Structured design of substitution-permutation encryption networks," *Computers, IEEE Transactions on*, vol. C-28, pp. 747–753, Oct. 1979.

[23] A. F. Webster and S. E. Tavares, "On the design of S-boxes," in *Advances in cryptology—CRYPTO 85*, vol. 218 of *Lecture Notes in Computer Science*, pp. 523–534, Springer-Verlag, 1986.

[24] R. C. Merkle and M. E. Hellman, "On the security of multiple encryption," *Communications of the ACM*, vol. 24, no. 7, pp. 465–467, 1981.

[25] M. Matsui, "Linear cryptanalysis method for DES cipher," in *EUROCRYPT '93: Workshop on the Theory and application of Cryptographic Techniques on Advances in cryptology*, vol. 765 of *Lecture Notes in Computer Science*, pp. 386–397, Springer-Verlag, 1994.

[26] E. Biham and A. Shamir, "Differential cryptanalysis of DES-like cryptosystems," *Journal of Cryptology*, vol. 4, no. 1, pp. 3–72, 1991.

[27] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS,

and other systems," in *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, vol. 1109 of *Lecture Notes in Computer Science*, (London, UK), pp. 104–113, Springer-Verlag, 1996.

[28] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, vol. 1666 of *Lecture Notes in Computer Science*, (London, UK), pp. 388–397, Springer-Verlag, 1999.

[29] W. Becker, "Method and system for machine enciphering and deciphering." U.S. Patent #4157454, 1979.

[30] L. R. Knudsen and W. Meier, "Correlations in RC6 with a reduced number of rounds," in *FSE '00: Proceedings of the 7th International Workshop on Fast Software Encryption*, vol. 1978 of *Lecture Notes in Computer Science*, (London, UK), pp. 94–108, Springer-Verlag, 2001.

[31] A. Miyaji and M. Nonaka, "Cryptanalysis of the reduced-round RC6," in *ICICS '02: Proceedings of the 4th International Conference on Information and Communications Security*, vol. 2513 of *Lecture Notes in Computer Science*, (London, UK), pp. 480–494, Springer-Verlag, 2002.

[32] N. Goots, N. Moklovyan, P. Moldovyanu, and D. Summerville, "Fast DDP-based ciphers: from hardware to software," *Micro-NanoMechatronics and Human Science, 2003 IEEE International Symposium on*, vol. 2, pp. 770–773 Vol. 2, Dec. 2003.

[33] C. Lee, D. Hong, S. Lee, S. Lee, H. Yang, and J. Lim, "A chosen plaintext linear attack on block cipher CIKS-1," in *Information and Communications Security: 4th International Conference, ICICS 2002, Singapore, December 9-12,*

2002. Proceedings, vol. 2513 of *Lecture Note in Computer Science*, pp. 456–468. Springer-Verlag Heidelberg, January 2002.

[34] M. Furlong and H. Heys, "A timing attack on the CIKS-1 block cipher," *Canadian Conference on Electrical and Computer Engineering, 2005.* pp. 231–234. May 2005.

[35] S. Dowdy and S. Wearden, *Statistics for Research.* New York: Wiley Publishing, Inc., second ed., 1991.

[36] National Bureau of Standards, "Data encryption standard, FIPS PUB 46, january 15, 1977," 1977.

[37] E. Biham, "A fast new DES implementation in software," in *FSE '97: Proceedings of the 4th International Workshop on Fast Software Encryption*, vol. 1267 of *Lecture Notes in Computer Science*, (London, UK), pp. 260–272, Springer-Verlag, 1997.

[38] H. S. Warren, *Hacker's Delight.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

Appendix A

CIKS–1 Implementations

A.1 CIKS–1 Bitset Implementation

```
1  /*****************************************************************************
2
3    Filename: ciks.cpp
4    Author: Brian Kidney, P.Eng
5
6    Description:
7     Straight forward implementation of the CIKS 1 block cipher using the
8       STL bitset class.
9
10 *****************************************************************************/
11 #include "ciks.h"
12
13 struct State {
14         bitset<32> L;
15         bitset<32> R;
16         bitset<32> K;
17         bitset<48> v;
18         bitset<48> v_k;
19         bitset<80> v_p;
20         bitset<32> s_p;
21         bitset<80> v_l;
22         bitset<32> s_pp;
23         bitset<80> v_pp;
24         bitset<32> xored;
25         bitset<32> key_permuted;
26         bitset<32> after_p1;
27         bitset<32> after_p2;
28         bitset<32> after_p6;
29 };
30
31 void swap_bits(bitset<32>& plaintext, int x_0, int x_1)
32 {
33         bool temp = plaintext.test(x_0);
34         plaintext[x_0] = plaintext[x_1];
35         plaintext[x_1] = temp;
36 }
37
38
39 void rotate_lsb_7(bitset<48>& input)
40 {
41     bitset<48> temp = input;
42     temp <<= 7;
43     input >>= (input.size() - 7);
44     input |= temp;
45 }
46
```

```
47  void rotate_lsb_7(bitset<32>& input)
48  {
49      bitset<32> temp = input;
50      temp <<= 7;
51      input >>= (input.size()    7);
52      input |= temp;
53  }
54
55  void DDP2_1( bitset<32>& plaintext, int x_0, int x_1, bool control_vector)
56  {
57      if (control_vector == false)
58      {
59          swap_bits(plaintext, x_0, x_1);
60      }
61  }
62
63
64  void DDP4_4( bitset<32>& plaintext, int x_0_pos,
65               int x_3_pos, bitset<4>& control_vector)
66  {
67      // Do initial controlled bit swaps
68      DDP2_1(plaintext, x_0_pos, x_0_pos + 1, control_vector[0]);
69      DDP2_1(plaintext, x_3_pos - 1, x_3_pos, control_vector[1]);
70
71      // Swap internal bits
72      swap_bits(plaintext, x_0_pos + 1, x_3_pos - 1);
73
74      // Do final controller bit swaps
75      DDP2_1(plaintext, x_0_pos, x_0_pos + 1, control_vector[2]);
76      DDP2_1(plaintext, x_3_pos - 1, x_3_pos, control_vector[3]);
77  }
78
79
80  void DDP8_12( bitset<32>& plaintext, int x_0_pos,
81                int x_7_pos, bitset<12>& control_vector)
82  {
83      // Do initial controlled bit swaps
84      for(int i = 0; i < 2; i++ )
85      {
86          bitset<4> cv;
87          for(int j = 0; j < 4; j++)
88          {
89              cv.set(j, control_vector[j + (i * 4)]);
90          }
91          DDP4_4(plaintext, x_0_pos + (i * 4), x_0_pos + (i * 4) + 3, cv);
92      }
93
94      // Do the bit swaps
95      bitset<32> temp;
96      for (int i = 0; i < x_0_pos; i++)
97          temp[i] = plaintext[i];
98      for(int i = 0; i < x_7_pos - x_0_pos; i++)
99      {
100         temp[i + x_0_pos] = plaintext[(i*((x_7_pos    x_0_pos + 1) /2))
101             % (x_7_pos - x_0_pos) + x_0_pos];
102     }
103     temp[32 - 1] = plaintext[32 - 1];
104     for (int i = x_7_pos; i < 32; i++)
105         temp[i] = plaintext[i];
106
107     plaintext = temp;
108
109     // Do final controlled permutations
110     for(int i = 0; i < 4; i++ )
111     {
112         DDP2_1(plaintext, x_0_pos + (i * 2),
113                 x_0_pos + (i * 2) + 1, control_vector[i + 8]);
114     }
```

```
115 }
116
117 void DDP32_48( bitset<32>& plaintext, bitset<48>& control_vector)
118 {
119     for (int i = 0; i < 4; i++)
120     {
121         bitset<12> cv;
122         for (int j = 0; j < 12; j++)
123         {
124             cv.set(j, control_vector[j + (i * 12)]);
125         }
126         DDP8_12(plaintext, i * 8, (i * 8) + 7, cv);
127     }
128 }
129
130 void DDP16_32( bitset<32>& plaintext, int x_0_pos,
131                int x_15_pos, bitset<32>& control_vector)
132 {
133     // Do initial controlled bit swaps
134     for(int i = 0; i < 2; i++ )
135     {
136         bitset<12> cv;
137         for(int j = 0; j < 12; j++)
138         {
139             cv.set(j, control_vector[j + (i * 12)]);
140         }
141         DDP8_12(plaintext, x_0_pos + (i * 8), x_0_pos + (i * 8) + 7, cv);
142     }
143
144     // Do the bit swaps
145     bitset<32> temp;
146     for (int i = 0; i < x_0_pos; i++)
147         temp[i] = plaintext[i];
148     for(int i = 0; i < x_15_pos - x_0_pos; i++)
149     {
150         temp[i + x_0_pos] = plaintext[(i*((x_15_pos - x_0_pos + 1) /2))
151             % (x_15_pos - x_0_pos) + x_0_pos];
152     }
153     temp[x_15_pos] = plaintext[x_15_pos];
154     for (int i = x_15_pos + 1; i < 32; i++)
155         temp[i] = plaintext[i];
156
157     plaintext = temp;
158
159     // Do final controlled permutations
160     for(int i = 0; i < 8; i++ )
161     {
162         DDP2_1(plaintext, x_0_pos + (i * 2), x_0_pos + (i * 2) + 1,
163             control_vector[i + 24]);
164     }
165 }
166
167 void DDP32_80( bitset<32>& plaintext, bitset<80>& control_vector)
168 {
169     // Do initial controlled bit swaps
170     for(int i = 0; i < 2; i++ )
171     {
172         bitset<32> cv;
173         for(int j = 0; j < 32; j++)
174         {
175             cv.set(j, control_vector[j + (i * 32)]);
176         }
177         DDP16_32(plaintext, (i * 16), (i * 16) + 15 , cv);
178     }
179
180     // Do the bit swaps
181     bitset<32> temp;
182     for(int i = 0; i < 32 - 1; i++)
```

```
183     {
184         temp[i] = plaintext[(i * (32 / 2)) % (32 - 1) ];
185     }
186     temp[32 - 1] = plaintext[32 - 1];
187     plaintext = temp;
188
189     // Do final controlled permutations
190     for(int i = 0; i < 16; i++ )
191     {
192         DDP2_1(plaintext, (i * 2), (i * 2) + 1, control_vector[i + 64]);
193     }
194 }
195
196 void P1_1( bitset<80>& plaintext )
197 {
198     bitset<80> temp;
199
200     for (int i = 0; i < 24; i++)
201         temp[i] = plaintext[i + 8];
202     for (int i = 24; i < 32; i++)
203         temp[i] = plaintext[i + 24];
204     for (int i = 32; i < 48; i++)
205         temp[i] = plaintext[i];
206     for (int i = 48; i < 56; i++)
207         temp[i] = plaintext[i - 48];
208     for (int i = 56; i < 80; i++)
209         temp[i] = plaintext[i];
210
211     plaintext = temp;
212 }
213
214 void P1_2( bitset<80>& plaintext )
215 {
216     bitset<80> temp;
217
218     for (int i = 0; i < 8; i++)
219         temp[i] = plaintext[i + 16];
220     for (int i = 8; i < 12; i++)
221         temp[i] = plaintext[i + 24];
222     for (int i = 12; i < 20; i++)
223         temp[i] = plaintext[i + 12];
224     for (int i = 20; i < 32; i++)
225         temp[i] = plaintext[i + 16];
226     for (int i = 32; i < 40; i++)
227         temp[i] = plaintext[i - 32];
228     for (int i = 40; i < 44; i++)
229         temp[i] = plaintext[i + 8];
230     for (int i = 44; i < 52; i++)
231         temp[i] = plaintext[i - 36];
232     for (int i = 52; i < 64; i++)
233         temp[i] = plaintext[i];
234     for (int i = 64; i < 80; i++)
235         temp[i] = plaintext[i];
236
237     plaintext = temp;
238 }
239
240 void parallel_addition( bitset<32>& lhs, bitset<32>& rhs )
241 {
242     for (int i = 0; i < 32; i = i + 2)
243     {
244         bool carry = lhs[i] & rhs[i];
245         lhs[i] = lhs[i] ^ rhs[i];
246         lhs[i + 1] = lhs[i + 1] ^ rhs[i + 1] ^ carry;
247     }
248 }
249
250 void parallel_xor( bitset<32>& lhs, bitset<32>& rhs )
```

```
251 {
252     lhs = lhs ^ rhs;
253 }
254
255 void ciks1_round( struct State& state )
256 {
257     // Form control vector v
258     for (int i = 0; i < 7; i++)
259         state.v[i] = state.R[25 + i];
260     for (int i = 7; i < 32; i++)
261         state.v[i] = state.R[i - 7];
262     for (int i = 32; i < 48; i++)
263         state.v[i] = state.R[i - 32];
264
265     // Apply P_1
266     DDP32_48(state.L, state.v);
267
268     // Form control vector v_k
269     for (int i = 0; i < 32; i++)
270         state.v_k[i] = state.K[i];
271     for (int i = 32; i < 48; i++)
272         state.v_k[i] = state.K[i - 32];
273
274     // Form control vector v_p
275     for (int i = 0; i < 32; i++)
276         state.v_p[i] = state.L[i];
277     for (int i = 32; i < 48; i++)
278         state.v_p[i] = state.L[i - 16];
279     state.s_p = state.L;
280     DDP32_48(state.s_p, state.v_k);
281     for (int i = 48; i < 80; i++)
282         state.v_p[i] = state.s_p[i - 48];
283
284     // Permute v_p
285     PI_1(state.v_p);
286
287     // Apply P_2
288     DDP32_80(state.R, state.v_p);
289
290     // Form Control vector v_l
291     for (int i = 0; i < 16; i++)
292         state.v_l[i] = state.L[i + 16];
293     for (int i = 16; i < 48; i++)
294         state.v_l[i] = state.L[i - 16];
295     for (int i = 48; i < 80; i++)
296         state.v_l[i] = state.L[i - 48];
297
298     // Apply P_4 to key
299     DDP32_80(state.K, state.v_l);
300
301     // XOR round key with Right Side
302     state.R = state.R ^ state.K;
303
304     // Rotate Left side
305     rotate_lsb_7(state.L);
306
307     // Form v_pp
308     state.s_pp = state.L;
309     rotate_lsb_7(state.v_k);
310     DDP32_48(state.s_pp, state.v_k);
311     for (int i = 0; i < 32; i++)
312         state.v_pp[i] = state.s_pp[i];
313     for (int i = 32; i < 64; i++)
314         state.v_pp[i] = state.L[i - 32];
315     for (int i = 64; i < 80; i++)
316         state.v_pp[i] = state.L[i - 64];
317
318     // Permute v_pp
```

```
319         P1_2(state.v_pp);
320
321         // Apply P_6
322         DDP32_80(state.R, state.v_pp);
323         state.after_p6 = state.R;
324
325         // Add to sides
326         parallel_addition(state.L, state.R);
327
328         // Swap side
329         bitset<32> temp = state.L;
330         state.L = state.R;
331         state.R = temp;
332     }
```

A.2 CIKS–1 Array Based Implementation

```
1  /*****************************************************************************
2
3    Filename: ciks.cpp
4    Author: Brian Kidney, P.Eng
5
6    Description:
7     Straight forward implementation of the CIKS-1 block cipher using
8       char arrays to hold individual bits.
9
10 *****************************************************************************/
11 #include "ciks.h"
12
13 struct State
14 {
15         char L[32];
16         char R[32];
17         char K[32];
18 };
19
20 void swap_bits(char* plaintext, int x_0, int x_1)
21 {
22         char temp = plaintext[x_0];
23         plaintext[x_0] = plaintext[x_1];
24         plaintext[x_1] = temp;
25 }
26
27 void rotate_lsb_7_48(char* input)
28 {
29     char temp[48];
30     memcpy(&temp, input, 48);
31
32     for (int i = 0; i < 48; i++)
33         input[i] = temp[(41 + i) % 48];
34 }
35
36 void rotate_lsb_7_32(char* input)
37 {
38     char temp[32];
39     memcpy(&temp, input, 32);
40
41     for (int i = 0; i < 32; i++)
42         input[i] = temp[(25 + i) % 32];
43 }
44
45 void DDP2_1( char* plaintext, int x_0, int x_1, char control_vector)
46 {
47     if (control_vector == 0)
48     {
49         swap_bits(plaintext, x_0, x_1);
50     }
51 }
52
53
54 void DDP4_4( char* plaintext, int x_0_pos, int x_3_pos, char* control_vector)
55 {
56     // Do initial controlled bit swaps
57     DDP2_1(plaintext, x_0_pos, x_0_pos + 1, control_vector[0]);
58     DDP2_1(plaintext, x_3_pos - 1, x_3_pos, control_vector[1]);
59
60     // Swap internal bits
61     swap_bits(plaintext, x_0_pos + 1, x_3_pos - 1);
62
63     // Do final controller bit swaps
64     DDP2_1(plaintext, x_0_pos, x_0_pos + 1, control_vector[2]);
65     DDP2_1(plaintext, x_3_pos - 1, x_3_pos, control_vector[3]);
```

```
66  }
67
68  void DDP8_12( char* plaintext, int x_0_pos, int x_7_pos, char* control_vector)
69  {
70      // Do initial controlled bit swaps
71      for(int i = 0; i < 2; i++ )
72      {
73          char* cv = control_vector;
74          cv +=  (i * 4);
75          DDP4_4(plaintext, x_0_pos + (i * 4), x_0_pos + (i * 4) + 3, cv);
76      }
77
78      // Do the bit swaps
79      char temp[8];
80      for(int i = 0; i < x_7_pos    x_0_pos; i++)
81      {
82          temp[i] = plaintext[(i*((x_7_pos - x_0_pos + 1) /2))
83              % (x_7_pos - x_0_pos) + x_0_pos];
84      }
85      for (int i = x_0_pos; i < x_7_pos; i++) // x_7_pos does not change
86          plaintext[i]    temp[i - x_0_pos];
87
88
89      // Do final controlled permutations
90      for(int i = 0; i < 4; i++ )
91      {
92          DDP2_1(plaintext, x_0_pos + (i * 2), x_0_pos + (i * 2) + 1,
93              control_vector[i + 8]);
94      }
95  }
96
97  void DDP32_48( char* plaintext, char* control_vector)
98  {
99      for (int i = 0; i < 4; i++)
100     {
101         char* cv = control_vector;
102         cv += (i * 12);
103         DDP8_12(plaintext, i * 8, (i * 8) + 7, cv);
104     }
105 }
106
107 void DDP16_32( char* plaintext, int x_0_pos, int x_15_pos, char* control_vector){
108     // Do initial controlled bit swaps
109     for(int i = 0; i < 2; i++ )
110     {
111         char* cv = control_vector;
112         cv += (i * 12);
113         DDP8_12(plaintext, x_0_pos + (i * 8), x_0_pos + (i * 8) + 7, cv);
114     }
115
116     // Do the bit swaps
117     char temp[16];
118     for(int i = 0; i < x_15_pos    x_0_pos; i++)
119     {
120         temp[i] = plaintext[(i*((x_15_pos - x_0_pos + 1) /2))
121             % (x_15_pos - x_0_pos) + x_0_pos];
122     }
123     for (int i = x_0_pos; i < x_15_pos; i++)
124         plaintext[i] = temp[i - x_0_pos];
125
126     // Do final controlled permutations
127     for(int i = 0; i < 8; i++ ) {
128         DDP2_1(plaintext, x_0_pos + (i * 2), x_0_pos + (i * 2) + 1,
129             control_vector[i + 24]);
130     }
131
132 }
133
```

```
134
135 // Function: DDP32_80( bitset<32> plaintext, bitset<80> control_vector)
136 //  Implements the CP Box P32/80.
137 void DDP32_80( char* plaintext, char* control_vector){
138
139     // Do initial controlled bit swaps
140     for(int i = 0; i < 2; i++ ) {
141         char* cv = control_vector;
142         cv += (i * 32);
143         DDP16_32(plaintext, (i * 16), (i * 16) + 15 , cv);
144     }
145
146     // Do the bit swaps
147     char temp[32];
148     for(int i = 0; i < 32 - 1; i++){
149         temp[i] = plaintext[(i * 16) % 31 ];
150     }
151
152     for (int i = 0; i < 32 - 1; i++)
153         plaintext[i] = temp[i];
154
155
156     // Do final controlled permutations
157     for(int i = 0; i < 16; i++ )
158     {
159         DDP2_1(plaintext, (i * 2), (i * 2) + 1, control_vector[i + 64]);
160     }
161 }
162
163 void PI_1( char* plaintext )
164 {
165     char temp[80];
166
167     for (int i = 0; i < 24; i++)
168         temp[i] = plaintext[i + 8];
169     for (int i = 24; i < 32; i++)
170         temp[i] = plaintext[i + 24];
171     for (int i = 32; i < 48; i ++)
172         temp[i] = plaintext[i];
173     for (int i = 48; i < 56; i++)
174         temp[i] = plaintext[i - 48];
175     for (int i = 56; i < 80; i++)
176         temp[i] = plaintext[i];
177
178     for (int i = 0; i < 80; i++)
179         plaintext[i] = temp[i];
180 }
181
182 void PI_2( char* plaintext )
183 {
184     char temp[80];
185
186     for (int i = 0; i < 8; i++)
187         temp[i] = plaintext[i + 16];
188     for (int i = 8; i < 12; i++)
189         temp[i] = plaintext[i + 24];
190     for (int i = 12; i < 20; i++)
191         temp[i] = plaintext[i + 12];
192     for (int i = 20; i < 32; i++)
193         temp[i] = plaintext[i + 16];
194     for (int i = 32; i < 40; i++)
195         temp[i] = plaintext[i - 32];
196     for (int i = 40; i < 44; i++)
197         temp[i] = plaintext[i + 8];
198     for (int i = 44; i < 52; i++)
199         temp[i] = plaintext[i - 36];
200     for (int i = 52; i < 64; i++)
201         temp[i] = plaintext[i];
```

```
202     for (int i = 64; i < 80; i++)
203         temp[i] = plaintext[i];
204
205     for (int i = 0; i < 80; i++)
206         plaintext[i] = temp[i];
207 }
208
209 void parallel_addition( char* lhs, char* rhs )
210 {
211     for (int i = 0; i < 32; i = i + 2)
212     {
213         char carry = lhs[i] & rhs[i];
214         lhs[i] = lhs[i] ^ rhs[i];
215         lhs[i + 1] = lhs[i + 1] ^ rhs[i + 1] ^ carry;
216     }
217 }
218
219 void parallel_xor( char* lhs, char* rhs )
220 {
221     for (int i = 0; i < 32; i++)
222         lhs[i] = lhs[i] ^ rhs[i];
223 }
224
225 void ciks1_round( struct State& state ){
226
227     char v[48];
228     // Form control vector v
229     for (int i = 0; i < 7; i++)
230         v[i] = state.R[25 + i];
231     for (int i = 7; i < 32; i++)
232         v[i] = state.R[i - 7];
233     for (int i = 32; i < 48; i++)
234         v[i] = state.R[i - 32];
235
236     // Apply P_1
237     DDP32_48(state.L, v);
238
239     // Form control vector v_k
240     char v_k[48];
241     for (int i = 0; i < 32; i++)
242         v_k[i] = state.K[i];
243     for (int i = 32; i < 48; i++)
244         v_k[i] = state.K[i - 32];
245
246     // Form control vector v_p
247     char v_p[80];
248     for (int i = 0; i < 32; i++)
249         v_p[i] = state.L[i];
250     for (int i = 32; i < 48; i++)
251         v_p[i] = state.L[i - 16];
252     char s_p[32];
253     memcpy(s_p, state.L, 32);
254     DDP32_48(s_p, v_k);
255
256     for (int i = 48; i < 80; i++)
257         v_p[i] = s_p[i - 48];
258
259     // Permute v_p
260     PI_1(v_p);
261
262     // Apply P_2
263     DDP32_80(state.R, v_p);
264
265     // Form Control vector v_l
266     char v_l[80];
267     for (int i = 0; i < 16; i++)
268         v_l[i] = state.L[i + 16];
269     for (int i = 16; i < 48; i++)
```

```
270             v_l[i] = state.L[i - 16];
271         for (int i = 48; i < 80; i++)
272             v_l[i] = state.L[i - 48];
273
274         // Apply P_4 to key
275         DDP32_80(state.K, v_l);
276
277         // XOR round key with Right Side
278         parallel_xor(state.R, state.K);
279
280         // Rotate Left side
281         rotate_lsb_7_32(state.L);
282
283         // Form v_pp
284         char v_pp[80];
285         char s_pp[32];
286         memcpy(s_pp, state.L, 32);
287         rotate_lsb_7_48(v_k);
288         DDP32_48(s_pp, v_k);
289         for (int i = 0; i < 32; i++)
290             v_pp[i] = s_pp[i];
291         for (int i = 32; i < 64; i++)
292             v_pp[i] = state.L[i - 32];
293         for (int i = 64; i < 80; i++)
294             v_pp[i] = state.L[i - 64];
295
296         // Permute v_pp
297         PI_2(v_pp);
298
299         // Apply P_6
300         DDP32_80(state.R, v_pp);
301
302         // Add to sides
303         parallel_addition(state.L, state.R);
304
305         // Swap side
306         char temp[32];
307         memcpy(temp, state.L, 32);
308
309         for (int i = 0; i < 32; i++)
310         {
311             state.L[i] = state.R[i];
312             state.R[i] = temp[i];
313         }
314     }
```

A.3 CIKS–1 Bitsliced Implementation

```
1  /*****************************************************************************
2
3    Filename: ciks.cpp
4    Author: Brian Kidney, P.Eng
5
6    Description:
7     Bitsliced implementation of the CIKS-1 block cipher.
8
9  *****************************************************************************/
10 #include "ciks.h"
11 #include "bitutil.h"
12
13 static inline void DDP32_80( unsigned long* x, unsigned long* c)
14 {
15         unsigned long temp[32];
16
17         // Layer 0
18         temp[0] = (c[0] & x[0]) | (~c[0] & x[1]);
19         temp[1] = (~c[0] & x[0]) | (c[0] & x[1]);
20         temp[2] = (c[1] & x[2]) | (~c[1] & x[3]);
21         temp[3] = (~c[1] & x[2]) | (c[1] & x[3]);
22         temp[4] = (c[4] & x[4]) | (~c[4] & x[5]);
23         temp[5] = (~c[4] & x[4]) | (c[4] & x[5]);
24         temp[6] = (c[5] & x[6]) | (~c[5] & x[7]);
25         temp[7] = (~c[5] & x[6]) | (c[5] & x[7]);
26         temp[8] = (c[12] & x[8]) | (~c[12] & x[9]);
27         temp[9] = (~c[12] & x[8]) | (c[12] & x[9]);
28         temp[10] = (c[13] & x[10]) | (~c[13] & x[11]);
29         temp[11] = (~c[13] & x[10]) | (c[13] & x[11]);
30         temp[12] = (c[16] & x[12]) | (~c[16] & x[13]);
31         temp[13] = (~c[16] & x[12]) | (c[16] & x[13]);
32         temp[14] = (c[17] & x[14]) | (~c[17] & x[15]);
33         temp[15] = (~c[17] & x[14]) | (c[17] & x[15]);
34         temp[16] = (c[32] & x[16]) | (~c[32] & x[17]);
35         temp[17] = (~c[32] & x[16]) | (c[32] & x[17]);
36         temp[18] = (c[33] & x[18]) | (~c[33] & x[19]);
37         temp[19] = (~c[33] & x[18]) | (c[33] & x[19]);
38         temp[20] = (c[36] & x[20]) | (~c[36] & x[21]);
39         temp[21] = (~c[36] & x[20]) | (c[36] & x[21]);
40         temp[22] = (c[37] & x[22]) | (~c[37] & x[23]);
41         temp[23] = (~c[37] & x[22]) | (c[37] & x[23]);
42         temp[24] = (c[44] & x[24]) | (~c[44] & x[25]);
43         temp[25] = (~c[44] & x[24]) | (c[44] & x[25]);
44         temp[26] = (c[45] & x[26]) | (~c[45] & x[27]);
45         temp[27] = (~c[45] & x[26]) | (c[45] & x[27]);
46         temp[28] = (c[48] & x[28]) | (~c[48] & x[29]);
47         temp[29] = (~c[48] & x[28]) | (c[48] & x[29]);
48         temp[30] = (c[49] & x[30]) | (~c[49] & x[31]);
49         temp[31] = (~c[49] & x[30]) | (c[49] & x[31]);
50
51         // Layer 1 (P4_4 butterflies)
52         for (int i = 0; i < 32; i = i + 4)
53         {
54                 x[i] = temp[i];
55                 x[i + 1] = temp[i + 2];
56                 x[i + 2] = temp[i + 1];
57                 x[i + 3] = temp[i + 3];
58         }
59
60         // Layer 2
61         temp[0] = (c[2] & x[0]) | (~c[2] & x[1]);
62         temp[1] = (~c[2] & x[0]) | (c[2] & x[1]);
63         temp[2] = (c[3] & x[2]) | (~c[3] & x[3]);
64         temp[3] = (~c[3] & x[2]) | (c[3] & x[3]);
65         temp[4] = (c[6] & x[4]) | (~c[6] & x[5]);
```

```
66          temp[5] = (~c[6] & x[4]) | (c[6] & x[5]);
67          temp[6] = (c[7] & x[6]) | (~c[7] & x[7]);
68          temp[7] = (~c[7] & x[6]) | (c[7] & x[7]);
69          temp[8] = (c[14] & x[8]) | (~c[14] & x[9]);
70          temp[9] = (~c[14] & x[8]) | (c[14] & x[9]);
71          temp[10] = (c[15] & x[10]) | (~c[15] & x[11]);
72          temp[11] = (~c[15] & x[10]) | (c[15] & x[11]);
73          temp[12] = (c[18] & x[12]) | (~c[18] & x[13]);
74          temp[13] = (~c[18] & x[12]) | (c[18] & x[13]);
75          temp[14] = (c[19] & x[14]) | (~c[19] & x[15]);
76          temp[15] = (~c[19] & x[14]) | (c[19] & x[15]);
77          temp[16] = (c[34] & x[16]) | (~c[34] & x[17]);
78          temp[17] = (~c[34] & x[16]) | (c[34] & x[17]);
79          temp[18] = (c[35] & x[18]) | (~c[35] & x[19]);
80          temp[19] = (~c[35] & x[18]) | (c[35] & x[19]);
81          temp[20] = (c[38] & x[20]) | (~c[38] & x[21]);
82          temp[21] = (~c[38] & x[20]) | (c[38] & x[21]);
83          temp[22] = (c[39] & x[22]) | (~c[39] & x[23]);
84          temp[23] = (~c[39] & x[22]) | (c[39] & x[23]);
85          temp[24] = (c[46] & x[24]) | (~c[46] & x[25]);
86          temp[25] = (~c[46] & x[24]) | (c[46] & x[25]);
87          temp[26] = (c[47] & x[26]) | (~c[47] & x[27]);
88          temp[27] = (~c[47] & x[26]) | (c[47] & x[27]);
89          temp[28] = (c[50] & x[28]) | (~c[50] & x[29]);
90          temp[29] = (~c[50] & x[28]) | (c[50] & x[29]);
91          temp[30] = (c[51] & x[30]) | (~c[51] & x[31]);
92          temp[31] = (~c[51] & x[30]) | (c[51] & x[31]);
93
94          // Layer 3 (P8_12 butterflies)
95          for (int i = 0; i < 32; i = i + 8)
96          {
97                  x[i] = temp[i];
98                  x[i + 1] = temp[i + 4];
99                  x[i + 2] = temp[i + 1];
100                 x[i + 3] = temp[i + 5];
101                 x[i + 4] = temp[i + 2];
102                 x[i + 5] = temp[i + 6];
103                 x[i + 6] = temp[i + 3];
104                 x[i + 7] = temp[i + 7];
105         }
106
107         // Layer 4
108         temp[0] = (c[8] & x[0]) | (~c[8] & x[1]);
109         temp[1] = (~c[8] & x[0]) | (c[8] & x[1]);
110         temp[2] = (c[9] & x[2]) | (~c[9] & x[3]);
111         temp[3] = (~c[9] & x[2]) | (c[9] & x[3]);
112         temp[4] = (c[10] & x[4]) | (~c[10] & x[5]);
113         temp[5] = (~c[10] & x[4]) | (c[10] & x[5]);
114         temp[6] = (c[11] & x[6]) | (~c[11] & x[7]);
115         temp[7] = (~c[11] & x[6]) | (c[11] & x[7]);
116         temp[8] = (c[20] & x[8]) | (~c[20] & x[9]);
117         temp[9] = (~c[20] & x[8]) | (c[20] & x[9]);
118         temp[10] = (c[21] & x[10]) | (~c[21] & x[11]);
119         temp[11] = (~c[21] & x[10]) | (c[21] & x[11]);
120         temp[12] = (c[22] & x[12]) | (~c[22] & x[13]);
121         temp[13] = (~c[22] & x[12]) | (c[22] & x[13]);
122         temp[14] = (c[23] & x[14]) | (~c[23] & x[15]);
123         temp[15] = (~c[23] & x[14]) | (c[23] & x[15]);
124         temp[16] = (c[40] & x[16]) | (~c[40] & x[17]);
125         temp[17] = (~c[40] & x[16]) | (c[40] & x[17]);
126         temp[18] = (c[41] & x[18]) | (~c[41] & x[19]);
127         temp[19] = (~c[41] & x[18]) | (c[41] & x[19]);
128         temp[20] = (c[42] & x[20]) | (~c[42] & x[21]);
129         temp[21] = (~c[42] & x[20]) | (c[42] & x[21]);
130         temp[22] = (c[43] & x[22]) | (~c[43] & x[23]);
131         temp[23] = (~c[43] & x[22]) | (c[43] & x[23]);
132         temp[24] = (c[52] & x[24]) | (~c[52] & x[25]);
133         temp[25] = (~c[52] & x[24]) | (c[52] & x[25]);
```

```
134         temp[26] = (c[53] & x[26]) | (~c[53] & x[27]);
135         temp[27] = (~c[53] & x[26]) | (c[53] & x[27]);
136         temp[28] = (c[54] & x[28]) | (~c[54] & x[29]);
137         temp[29] = (~c[54] & x[28]) | (c[54] & x[29]);
138         temp[30] = (c[55] & x[30]) | (~c[55] & x[31]);
139         temp[31] = (~c[55] & x[30]) | (c[55] & x[31]);
140
141         // Layer 5 (P16_32 butterflies)
142         for (int i = 0; i < 32; i = i + 16)
143         {
144                 x[i] = temp[i];
145                 x[i + 1] = temp[i + 8];
146                 x[i + 2] = temp[i + 1];
147                 x[i + 3] = temp[i + 9];
148                 x[i + 4] = temp[i + 2];
149                 x[i + 5] = temp[i + 10];
150                 x[i + 6] = temp[i + 3];
151                 x[i + 7] = temp[i + 11];
152                 x[i + 8] = temp[i + 4];
153                 x[i + 9] = temp[i + 12];
154                 x[i + 10] = temp[i + 5];
155                 x[i + 11] = temp[i + 13];
156                 x[i + 12] = temp[i + 6];
157                 x[i + 13] = temp[i + 14];
158                 x[i + 14] = temp[i + 7];
159                 x[i + 15] = temp[i + 15];
160         }
161
162         // Layer 6
163         temp[0] = (c[24] & x[0]) | (~c[24] & x[1]);
164         temp[1] = (~c[24] & x[0]) | (c[24] & x[1]);
165         temp[2] = (c[25] & x[2]) | (~c[25] & x[3]);
166         temp[3] = (~c[25] & x[2]) | (c[25] & x[3]);
167         temp[4] = (c[26] & x[4]) | (~c[26] & x[5]);
168         temp[5] = (~c[26] & x[4]) | (c[26] & x[5]);
169         temp[6] = (c[27] & x[6]) | (~c[27] & x[7]);
170         temp[7] = (~c[27] & x[6]) | (c[27] & x[7]);
171         temp[8] = (c[28] & x[8]) | (~c[28] & x[9]);
172         temp[9] = (~c[28] & x[8]) | (c[28] & x[9]);
173         temp[10] = (c[29] & x[10]) | (~c[29] & x[11]);
174         temp[11] = (~c[29] & x[10]) | (c[29] & x[11]);
175         temp[12] = (c[30] & x[12]) | (~c[30] & x[13]);
176         temp[13] = (~c[30] & x[12]) | (c[30] & x[13]);
177         temp[14] = (c[31] & x[14]) | (~c[31] & x[15]);
178         temp[15] = (~c[31] & x[14]) | (c[31] & x[15]);
179         temp[16] = (c[56] & x[16]) | (~c[56] & x[17]);
180         temp[17] = (~c[56] & x[16]) | (c[56] & x[17]);
181         temp[18] = (c[57] & x[18]) | (~c[57] & x[19]);
182         temp[19] = (~c[57] & x[18]) | (c[57] & x[19]);
183         temp[20] = (c[58] & x[20]) | (~c[58] & x[21]);
184         temp[21] = (~c[58] & x[20]) | (c[58] & x[21]);
185         temp[22] = (c[59] & x[22]) | (~c[59] & x[23]);
186         temp[23] = (~c[59] & x[22]) | (c[59] & x[23]);
187         temp[24] = (c[60] & x[24]) | (~c[60] & x[25]);
188         temp[25] = (~c[60] & x[24]) | (c[60] & x[25]);
189         temp[26] = (c[61] & x[26]) | (~c[61] & x[27]);
190         temp[27] = (~c[61] & x[26]) | (c[61] & x[27]);
191         temp[28] = (c[62] & x[28]) | (~c[62] & x[29]);
192         temp[29] = (~c[62] & x[28]) | (c[62] & x[29]);
193         temp[30] = (c[63] & x[30]) | (~c[63] & x[31]);
194         temp[31] = (~c[63] & x[30]) | (c[63] & x[31]);
195
196         // Layer 7 (P32_80 butterflies)
197         for (int i = 0; i < 16; i++)
198         {
199                 x[i * 2] = temp[i];
200                 x[(i * 2) + 1] = temp[i + 16];
201         }
```

```
202
203         // Layer 8
204         temp[0] = (c[64] & x[0]) | (~c[64] & x[1]);
205         temp[1] = (~c[64] & x[0]) | (c[64] & x[1]);
206         temp[2] = (c[65] & x[2]) | (~c[65] & x[3]);
207         temp[3] = (~c[65] & x[2]) | (c[65] & x[3]);
208         temp[4] = (c[66] & x[4]) | (~c[66] & x[5]);
209         temp[5] = (~c[66] & x[4]) | (c[66] & x[5]);
210         temp[6] = (c[67] & x[6]) | (~c[67] & x[7]);
211         temp[7] = (~c[67] & x[6]) | (c[67] & x[7]);
212         temp[8] = (c[68] & x[8]) | (~c[68] & x[9]);
213         temp[9] = (~c[68] & x[8]) | (c[68] & x[9]);
214         temp[10] = (c[69] & x[10]) | (~c[69] & x[11]);
215         temp[11] = (~c[69] & x[10]) | (c[69] & x[11]);
216         temp[12] = (c[70] & x[12]) | (~c[70] & x[13]);
217         temp[13] = (~c[70] & x[12]) | (c[70] & x[13]);
218         temp[14] = (c[71] & x[14]) | (~c[71] & x[15]);
219         temp[15] = (~c[71] & x[14]) | (c[71] & x[15]);
220         temp[16] = (c[72] & x[16]) | (~c[72] & x[17]);
221         temp[17] = (~c[72] & x[16]) | (c[72] & x[17]);
222         temp[18] = (c[73] & x[18]) | (~c[73] & x[19]);
223         temp[19] = (~c[73] & x[18]) | (c[73] & x[19]);
224         temp[20] = (c[74] & x[20]) | (~c[74] & x[21]);
225         temp[21] = (~c[74] & x[20]) | (c[74] & x[21]);
226         temp[22] = (c[75] & x[22]) | (~c[75] & x[23]);
227         temp[23] = (~c[75] & x[22]) | (c[75] & x[23]);
228         temp[24] = (c[76] & x[24]) | (~c[76] & x[25]);
229         temp[25] = (~c[76] & x[24]) | (c[76] & x[25]);
230         temp[26] = (c[77] & x[26]) | (~c[77] & x[27]);
231         temp[27] = (~c[77] & x[26]) | (c[77] & x[27]);
232         temp[28] = (c[78] & x[28]) | (~c[78] & x[29]);
233         temp[29] = (~c[78] & x[28]) | (c[78] & x[29]);
234         temp[30] = (c[79] & x[30]) | (~c[79] & x[31]);
235         temp[31] = (~c[79] & x[30]) | (c[79] & x[31]);
236
237         // Assign back to x[]
238         for (int i = 0; i < 32; i++)
239         {
240                 x[i] = temp[i];
241         }
242
243 }
244
245
246 static inline void DDP32_48( unsigned long* x, unsigned long* c, unsigned long* out)
247 {
248
249         unsigned long temp[32];
250
251         // Layer 0
252         temp[0] = (c[0] & x[0]) | (~c[0] & x[1]);
253         temp[1] = (~c[0] & x[0]) | (c[0] & x[1]);
254         temp[2] = (c[1] & x[2]) | (~c[1] & x[3]);
255         temp[3] = (~c[1] & x[2]) | (c[1] & x[3]);
256         temp[4] = (c[4] & x[4]) | (~c[4] & x[5]);
257         temp[5] = (~c[4] & x[4]) | (c[4] & x[5]);
258         temp[6] = (c[5] & x[6]) | (~c[5] & x[7]);
259         temp[7] = (~c[5] & x[6]) | (c[5] & x[7]);
260         temp[8] = (c[12] & x[8]) | (~c[12] & x[9]);
261         temp[9] = (~c[12] & x[8]) | (c[12] & x[9]);
262         temp[10] = (c[13] & x[10]) | (~c[13] & x[11]);
263         temp[11] = (~c[13] & x[10]) | (c[13] & x[11]);
264         temp[12] = (c[16] & x[12]) | (~c[16] & x[13]);
265         temp[13] = (~c[16] & x[12]) | (c[16] & x[13]);
266         temp[14] = (c[17] & x[14]) | (~c[17] & x[15]);
267         temp[15] = (~c[17] & x[14]) | (c[17] & x[15]);
268         temp[16] = (c[24] & x[16]) | (~c[24] & x[17]);
269         temp[17] = (~c[24] & x[16]) | (c[24] & x[17]);
```

```
270         temp[18] = (c[25] & x[18]) | (~c[25] & x[19]);
271         temp[19] = (~c[25] & x[18]) | (c[25] & x[19]);
272         temp[20] = (c[28] & x[20]) | (~c[28] & x[21]);
273         temp[21] = (~c[28] & x[20]) | (c[28] & x[21]);
274         temp[22] = (c[29] & x[22]) | (~c[29] & x[23]);
275         temp[23] = (~c[29] & x[22]) | (c[29] & x[23]);
276         temp[24] = (c[36] & x[24]) | (~c[36] & x[25]);
277         temp[25] = (~c[36] & x[24]) | (c[36] & x[25]);
278         temp[26] = (c[37] & x[26]) | (~c[37] & x[27]);
279         temp[27] = (~c[37] & x[26]) | (c[37] & x[27]);
280         temp[28] = (c[40] & x[28]) | (~c[40] & x[29]);
281         temp[29] = (~c[40] & x[28]) | (c[40] & x[29]);
282         temp[30] = (c[41] & x[30]) | (~c[41] & x[31]);
283         temp[31] = (~c[41] & x[30]) | (c[41] & x[31]);
284
285         // Layer 1 (P4_4 butterflies)
286         for (int i = 0; i < 32; i = i + 4)
287         {
288                 out[i] = temp[i];
289                 out[i + 1] = temp[i + 2];
290                 out[i + 2] = temp[i + 1];
291                 out[i + 3] = temp[i + 3];
292         }
293
294         // Layer 2
295         temp[0] = (c[2] & out[0]) | (~c[2] & out[1]);
296         temp[1] = (~c[2] & out[0]) | (c[2] & out[1]);
297         temp[2] = (c[3] & out[2]) | (~c[3] & out[3]);
298         temp[3] = (~c[3] & out[2]) | (c[3] & out[3]);
299         temp[4] = (c[6] & out[4]) | (~c[6] & out[5]);
300         temp[5] = (~c[6] & out[4]) | (c[6] & out[5]);
301         temp[6] = (c[7] & out[6]) | (~c[7] & out[7]);
302         temp[7] = (~c[7] & out[6]) | (c[7] & out[7]);
303         temp[8] = (c[14] & out[8]) | (~c[14] & out[9]);
304         temp[9] = (~c[14] & out[8]) | (c[14] & out[9]);
305         temp[10] = (c[15] & out[10]) | (~c[15] & out[11]);
306         temp[11] = (~c[15] & out[10]) | (c[15] & out[11]);
307         temp[12] = (c[18] & out[12]) | (~c[18] & out[13]);
308         temp[13] = (~c[18] & out[12]) | (c[18] & out[13]);
309         temp[14] = (c[19] & out[14]) | (~c[19] & out[15]);
310         temp[15] = (~c[19] & out[14]) | (c[19] & out[15]);
311         temp[16] = (c[26] & out[16]) | (~c[26] & out[17]);
312         temp[17] = (~c[26] & out[16]) | (c[26] & out[17]);
313         temp[18] = (c[27] & out[18]) | (~c[27] & out[19]);
314         temp[19] = (~c[27] & out[18]) | (c[27] & out[19]);
315         temp[20] = (c[30] & out[20]) | (~c[30] & out[21]);
316         temp[21] = (~c[30] & out[20]) | (c[30] & out[21]);
317         temp[22] = (c[31] & out[22]) | (~c[31] & out[23]);
318         temp[23] = (~c[31] & out[22]) | (c[31] & out[23]);
319         temp[24] = (c[38] & out[24]) | (~c[38] & out[25]);
320         temp[25] = (~c[38] & out[24]) | (c[38] & out[25]);
321         temp[26] = (c[39] & out[26]) | (~c[39] & out[27]);
322         temp[27] = (~c[39] & out[26]) | (c[39] & out[27]);
323         temp[28] = (c[42] & out[28]) | (~c[42] & out[29]);
324         temp[29] = (~c[42] & out[28]) | (c[42] & out[29]);
325         temp[30] = (c[43] & out[30]) | (~c[43] & out[31]);
326         temp[31] = (~c[43] & out[30]) | (c[43] & out[31]);
327
328         // Layer 3 (P8_12 butterflies)
329         for (int i = 0; i < 32; i = i + 8)
330         {
331                 out[i] = temp[i];
332                 out[i + 1] = temp[i + 4];
333                 out[i + 2] = temp[i + 1];
334                 out[i + 3] = temp[i + 5];
335                 out[i + 4] = temp[i + 2];
336                 out[i + 5] = temp[i + 6];
337                 out[i + 6] = temp[i + 3];
```

```
338                 out[i + 7] = temp[i + 7];
339         }
340
341         // Layer 4
342         temp[0]  =  (c[8] & out[0]) | (~c[8] & out[1]);
343         temp[1]  = (~c[8] & out[0]) | (c[8] & out[1]);
344         temp[2]  =  (c[9] & out[2]) | (~c[9] & out[3]);
345         temp[3]  = (~c[9] & out[2]) | (c[9] & out[3]);
346         temp[4]  = (c[10] & out[4]) | (~c[10] & out[5]);
347         temp[5]  = (~c[10] & out[4]) | (c[10] & out[5]);
348         temp[6]  = (c[11] & out[6]) | (~c[11] & out[7]);
349         temp[7]  = (~c[11] & out[6]) | (c[11] & out[7]);
350         temp[8]  = (c[20] & out[8]) | (~c[20] & out[9]);
351         temp[9]  = (~c[20] & out[8]) | (c[20] & out[9]);
352         temp[10] = (c[21] & out[10]) | (~c[21] & out[11]);
353         temp[11] = (~c[21] & out[10]) | (c[21] & out[11]);
354         temp[12] = (c[22] & out[12]) | (~c[22] & out[13]);
355         temp[13] = (~c[22] & out[12]) | (c[22] & out[13]);
356         temp[14] = (c[23] & out[14]) | (~c[23] & out[15]);
357         temp[15] = (~c[23] & out[14]) | (c[23] & out[15]);
358         temp[16] = (c[32] & out[16]) | (~c[32] & out[17]);
359         temp[17] = (~c[32] & out[16]) | (c[32] & out[17]);
360         temp[18] = (c[33] & out[18]) | (~c[33] & out[19]);
361         temp[19] = (~c[33] & out[18]) | (c[33] & out[19]);
362         temp[20] = (c[34] & out[20]) | (~c[34] & out[21]);
363         temp[21] = (~c[34] & out[20]) | (c[34] & out[21]);
364         temp[22] = (c[35] & out[22]) | (~c[35] & out[23]);
365         temp[23] = (~c[35] & out[22]) | (c[35] & out[23]);
366         temp[24] = (c[44] & out[24]) | (~c[44] & out[25]);
367         temp[25] = (~c[44] & out[24]) | (c[44] & out[25]);
368         temp[26] = (c[45] & out[26]) | (~c[45] & out[27]);
369         temp[27] = (~c[45] & out[26]) | (c[45] & out[27]);
370         temp[28] = (c[46] & out[28]) | (~c[46] & out[29]);
371         temp[29] = (~c[46] & out[28]) | (c[46] & out[29]);
372         temp[30] = (c[47] & out[30]) | (~c[47] & out[31]);
373         temp[31] = (~c[47] & out[30]) | (c[47] & out[31]);
374
375         // Assign back to out[]
376         for (int i = 0; i < 32; i++)
377         {
378                 out[i] = temp[i];
379         }
380 }
381
382 static inline void add_key( unsigned long* x, unsigned long* k)
383 {
384         for (int i = 0; i < 32; i++)
385         {
386                 x[i] = x[i] ^ k[i];
387         }
388 }
389
390 // 16 parallel mod 4 additions where x = x + y
391 static inline void parallel_addition( unsigned long* x, unsigned long* y )
392 {
393         unsigned long c;
394
395         for (int i = 0; i < 32; i = i + 2)
396         {
397                 c = x[i] & y[i];
398                 x[i]  = x[i] ^ y[i];
399                 x[i + 1] = x[i + 1] ^ y[i + 1] ^ c;
400         }
401 }
402
403 static inline void rotate_right_7_32bits(unsigned long* x)
404 {
405         unsigned long t[32];
```

```
406
407         for (int i   0; i < 32; i++)
408         {
409                 t[i] = x[(i + 25) % 32];
410         }
411
412         // Assign back to x[]
413         for (int i = 0; i < 32; i++)
414         {
415                 x[i] = t[i];
416         }
417 }
418
419 static inline void rotate_right_7_48bits(unsigned long* x)
420 {
421         unsigned long t[48];
422
423         for (int i = 0; i < 48; i++)
424         {
425                 t[i] = x[(i + 41) % 48];
426         }
427
428         // Assign back to x[]
429         for (int i = 0; i < 48; i++)
430         {
431                 x[i] = t[i];
432         }
433 }
434
435
436 void ciks1_round( unsigned long* L, unsigned long* R, unsigned long* K )
437 {
438
439         // Form control vector v
440         {
441                 unsigned long v[48];
442                 for (int i = 0; i < 32; i++)
443                 {
444                         v[i] = R[(i + 25) % 32];
445                 }
446                 for (int i = 0; i < 16; i++)
447                 {
448                         v[i + 32] = R[i];
449                 }
450
451                 // Apply P_1(32/48)
452                 DDP32_48(L, v, L);
453         }
454
455
456
457         // Form control vector v_k
458         unsigned long v_k[48];
459         for (int i = 0; i < 48; i++)
460         {
461                 v_k[i] = K[i % 32];
462         }
463
464
465
466         {
467                 // Form control vector v_p (fixed permutation built in)
468                 unsigned long s_p[32];
469                 DDP32_48(L, v_k, s_p);
470
471
472                 unsigned long v_p[80];
473                 for (int i = 0; i < 24; i++)
```

```
474                 {
475                         v_p[i] = L[i + 8];
476                 }
477                 for (int i = 0; i < 8; i++)
478                 {
479                         v_p[i + 24] = s_p[i];
480                 }
481                 for (int i = 16; i < 32; i++)
482                 {
483                         v_p[i + 16] = L[i];
484                 }
485                 for (int i = 0; i < 8; i++)
486                 {
487                         v_p[i + 48] = L[i];
488                 }
489                 for (int i = 8; i < 32; i++)
490                 {
491                         v_p[i + 48] = s_p[i];
492                 }
493
494                 // Apply P_2(32/80)
495                 DDP32_80(R, v_p);
496         }
497
498         {
499                 // Form Control vector v_l
500                 unsigned long v_l[80];
501                 for (int i = 0; i < 16; i++)
502                 {
503                         v_l[i] = L[i + 16];
504                 }
505                 for (int i = 0; i < 32; i++)
506                 {
507                         v_l[i + 16] = L[i];
508                 }
509                 for (int i = 0; i < 32; i++)
510                 {
511                         v_l[i + 48] = L[i];
512                 }
513
514                 // Apply P_4(32/80) to key
515                 DDP32_80(K, v_l);
516
517                 // XOR round key with Right Side
518                 add_key( R, K );
519         }
520
521         // Rotate Left side
522         rotate_right_7_32bits(L);
523
524         {
525                 // Form v_pp (fixed permutation built in)
526                 rotate_right_7_48bits(v_k);
527
528                 unsigned long s_pp[32];
529                 DDP32_48(L, v_k, s_pp);
530
531                 unsigned long v_pp[80];
532                 for (int i = 0; i < 8; i++)
533                 {
534                         v_pp[i] = s_pp[i + 16];
535                 }
536                 for (int i = 0; i < 4; i++)
537                 {
538                         v_pp[i + 8] = L[i];
539                 }
540                 for (int i = 12; i < 20; i++)
541                 {
```

```
542                         v_pp[i] = s_pp[i + 12];
543                 }
544                 for (int i = 4; i < 15; i++)
545                 {
546                         v_pp[i + 16] = L[i];
547                 }
548                 for (int i = 0; i < 8; i++)
549                 {
550                         v_pp[i + 32] = s_pp[i];
551                 }
552                 for (int i = 16; i < 20; i++)
553                 {
554                         v_pp[i + 24] = L[i];
555                 }
556                 for (int i = 8; i < 16; i++)
557                 {
558                         v_pp[i + 36] = s_pp[i];
559                 }
560                 for (int i = 20; i < 32; i++)
561                 {
562                         v_pp[i + 32] = L[i];
563                 }
564                 for (int i = 0; i < 16; i++)
565                 {
566                         v_pp[i + 64] = L[i];
567                 }
568
569                 // Apply P_6(32/80)
570                 DDP32_80(R, v_pp);
571
572         }
573
574         // Add two sides with parallel additions
575         parallel_addition( L, R );
576
577         // Swap side
578         unsigned long* temp = R;
579         R = L;
580         L = temp;
581
582 }
```

Appendix B

Weight Based Attack Implementation Code

```
1  /*****************************************************************************
2
3    Filename: ciks_attack.cpp
4    Author: Brian Kidney, P.Eng
5
6    Description:
7     Weight based attack on CIKS 1 cipher.
8
9  *****************************************************************************/
10
11 #include "ciks_attack.h"
12
13 void top_down_low_weight_crack() {
14
15     // Guess a subkey
16     //
17     // For each of a million vectors
18     //  Run subkey through P4, call result PSK
19     //  Run PSK back through P^(-1)2 to get rhs
20     //  Run vector back through P^(-1)1 to get lhs
21     //  Encrypt rhs and lhs, record result
22     // Run statistical test on set of results
23
24     //-------------------------------------------
25     // Expected distribition for the results
26     long double expected[65] = {
27         0.000000000000542101,0.00000000000346945,0.000000000109288,
28         0.00000000225861,0.0000000344438,0.000000413326,0.00000406437,
29         0.000036762,0.000239943,0.001492978, 0.008211379,0.040310408,
30         0.178037636,0.712150543,2.594262693,8.64754231,26.48309832,
31         74.77580704,195.2479406,472.7055404,1063.587466,2228.468976,
32         4355.643908,7953.784527,13587.71523,21740.34437,32610.51656,
33         45896.28257,60648.65911,75287.99062,87835.98905,96336.24606,
34         99346.75375,96336.24606,87835.98905,75287.99062,60648.65911,
35         45896.28257,32610.51656,21740.34437,13587.71523,7953.784527,
36         4355.643908,2228.468976,1063.587466,472.7055404,195.2479406,
37         74.77580704,26.48309832,8.64754231,2.594262693,0.712150543,
38         0.178037636,0.040310408,0.008211379,0.001492978,0.000239943,
39         0.000036762,0.00000406437,0.000000413326,0.0000000344438,
40         0.00000000225861,0.000000000109288,0.00000000000346945,
41         0.000000000000542101};
42     // -----------------------------------------
43
44     // Set up for random number generator
45     long* seed = new long;
46     *seed = -1 * time( NULL );
47
```

```
48      // Create subkeys for six rounds of the cipher
49      bitset<32> key[6];
50      key[0] = 0x00000000;
51      for (int i = 1; i < 6; i++)
52      {
53          key[i] = 0x00000000;
54          int k = 1;
55          while (k <= 6)
56          {
57              int bit = ((int)((rand1(seed) * (float)(32.0))));
58              if (!key[i][bit])
59              {
60                  key[i].set(bit, true);
61              }
62              k++;
63          }
64      }
65
66      // Array to hold current round data.
67      int data[6][65];
68      for (int r = 0; r < 6; r++)
69          for (int i = 0; i < 64; i++)
70              data[r][i] = 0;
71
72      // Get statistics on guessing the key exactly
73      bitset<32> g_subkey = 0x00000000;
74
75      // For each of a million vectors (wt <= 6)
76      for (int i = 0; i < 1000000; i++)
77      {
78          //  Make random vector
79          bitset<32> lhs = 0x00000000;
80          int k = 0;
81          while (k < 6)
82          {
83              int bit = ((int)((rand1(seed) * (float)(32.0))));
84              if (!lhs[bit])
85              {
86                  lhs.set(bit, true);
87              }
88              k++;
89          }
90
91          //  Run subkey through P4, call result PSK
92          bitset<32> psk = g_subkey;
93
94          // Form Control vector v_l
95          bitset<80> v_l;
96          for (int c = 0; c < 16; c++)
97              v_l[c] = lhs[c + 16];
98          for (int c = 16; c < 48; c++)
99              v_l[c] = lhs[c - 16];
100         for (int c = 48; c < 80; c++)
101             v_l[c] = lhs[c   48];
102
103         DDP32_80(psk, v_l);
104
105         //  Run PSK back through P^(-1)2 to get rhs
106         // Form control vector v_k
107         bitset<48> v_k;
108         for (int c = 0; c < 32; c++)
109             v_k[c] = g_subkey[c];
110         for (int c = 32; c < 48; c++)
111             v_k[c] = g_subkey[c - 32];
112         // Form control vector v_p
113         bitset<80> v_p;
114         for (int c = 0; c < 32; c++)
115             v_p[c] = lhs[c];
```

```
116             for (int c = 32; c < 48; c++)
117                 v_p[c] = lhs[c - 16];
118             bitset<32> s_p;
119             s_p = lhs;
120             DDP32_48(s_p, v_k);
121             for (int c = 48; c < 80; c++)
122                 v_p[c] = s_p[c - 48];
123
124             // Permute v_p
125             PI_1(v_p);
126
127             DDP32_80_INV(psk, v_p);
128             bitset<32> rhs = psk;
129
130             //  Run vector back through P^(-1)1 to get lhs
131             // Form control vector v
132             bitset<48> v;
133             for (int c = 0; c < 7; c++)
134                 v[c] = rhs[25 + c];
135             for (int c = 7; c < 32; c++)
136                 v[c] = rhs[c - 7];
137             for (int c = 32; c < 48; c++)
138                 v[c] = rhs[c - 32];
139             DDP32_48_INV(lhs, v);
140
141             //  Encrypt rhs and lhs, record result
142             for (int r = 0; r < 6; r++)
143             {
144                 //ciks1_round(lhs, rhs, key[r]);
145                 // Count weight of output text and store in array
146                 data[r][lhs.count() + rhs.count()]++;
147             }
148
149         }
150
151         // RUN CHI-SQUARED TEST HERE
152         int temp[65];
153         for (int idx = 0; idx < 65; idx++)
154             temp[idx] = data[5][idx];
155
156         long double chiResult = chiSquaredTest(temp, expected);
157
158         cout << g_subkey << endl;
159         for (int rnd = 0; rnd < 6; rnd++)
160         {
161             cout << "Round " << rnd + 1;
162             for (int wt = 0; wt < 65; wt++)
163                 cout << "," << data[rnd][wt];
164             cout << endl;
165         }
166         cout << "Chi-Squared Result," << chiResult << endl;
167
168         // Clear the data array
169         for (int r = 0; r < 6; r++)
170             for (int i = 0; i < 65; i++)
171                 data[r][i] = 0;
172
173         // Get statists for 100 keys off by 1,2,...,5 bits
174         for (int i = 0; i < 5; i++)
175         {
176             for (int j = 1; j <= 100; j++)
177             {
178                 int bits_set = 0;
179                 g_subkey = 0x00000000;
180                 while (bits_set < i + 1)
181                 {
182                     int bit = ((int)((rand1(seed) * (float)(32.0)))) ;
183                     if (!g_subkey[bit])
```

```
184                 {
185                     g_subkey.set(bit, true);
186                     bits_set++;
187                 }
188             }
189
190             // For each of a million vectors (wt <= 6)
191             for (int vec = 0; vec < 1000000; vec++)
192             {
193                 // Make random vector
194                 bitset<32> lhs = 0x00000000;
195                 int k = 0;
196                 while (k < 6)
197                 {
198                     int bit = ((int)((randl(seed)
199                                 * (float)(32.0))));
200                     if (!lhs[bit])
201                     {
202                         lhs.set(bit, true);
203                     }
204                     k++;
205                 }
206
207                 // Run subkey through P4, call result PSK
208                 bitset<32> psk = g_subkey;
209
210                 // Form Control vector v_l
211                 bitset<80> v_l;
212                 for (int c = 0; c < 16; c++)
213                     v_l[c] = lhs[c + 16];
214                 for (int c = 16; c < 48; c++)
215                     v_l[c] = lhs[c - 16];
216                 for (int c = 48; c < 80; c++)
217                     v_l[c] = lhs[c - 48];
218
219                 DDP32_80(psk, v_l);
220
221                 // Run PSK back through P^(-1)2 to get rhs
222
223                 // Form control vector v_k
224                 bitset<48> v_k;
225                 for (int c = 0; c < 32; c++)
226                     v_k[c] = g_subkey[c];
227                 for (int c = 32; c < 48; c++)
228                     v_k[c] = g_subkey[c - 32];
229                 // Form control vector v_p
230                 bitset<80> v_p;
231                 for (int c = 0; c < 32; c++)
232                     v_p[c] = lhs[c];
233                 for (int c = 32; c < 48; c++)
234                     v_p[c] = lhs[c - 16];
235                 bitset<32> s_p;
236                 s_p = lhs;
237                 DDP32_48(s_p, v_k);
238                 for (int c = 48; c < 80; c++)
239                     v_p[c] = s_p[c - 48];
240                 // Permute v_p
241                 PI_1(v_p);
242
243                 DDP32_80_INV(psk, v_p);
244                 bitset<32> rhs = psk;
245
246
247                 // Run vector back through P^(-1)1 to get lhs
248                 // Form control vector v
249                 bitset<48> v;
250                 for (int c = 0; c < 7; c++)
251                     v[c] = rhs[25 + c];
```

```
252                 for (int c = 7; c < 32; c++)
253                     v[c] = rhs[c - 7];
254                 for (int c = 32; c < 48; c++)
255                     v[c] = rhs[c - 32];
256                 DDP32_48_INV(lhs, v);
257
258                 //  Encrypt rhs and lhs, record result
259                 for (int r = 0; r < 6; r++)
260                 {
261                     //ciks1_round(lhs, rhs, key[r]);
262                     // Count weight of output text
263                     // and store in array
264                     data[r][lhs.count() + rhs.count()]++;
265                 }
266
267             }
268             // RUN CHI-SQUARED TEST HERE
269             int temp[65];
270             for (int idx = 0; idx < 65; idx++)
271                 temp[idx] = data[5][idx];
272             long double chiResult = chiSquaredTest(temp, expected);
273
274             cout << g_subkey << endl;
275             for (int rnd = 0; rnd < 6; rnd++)
276             {
277                 cout << "Round_" << rnd + 1;
278                 for (int wt = 0; wt < 65; wt++)
279                     cout << "," << data[rnd][wt];
280                 cout << endl;
281             }
282             cout << "Chi-Squared_Result," << chiResult << endl;
283
284             // Clear the data array
285             for (int r = 0; r < 6; r++)
286                 for (int i = 0; i < 65; i++)
287                     data[r][i] = 0;
288         }
289     }
290 }
291
292 long double chiSquaredTest(int data[], long double expected[])
293 {
294     long double chiSqr = 0;
295
296     // for each in the 65 possible values
297     for (int i = 0; i < 65; i++)
298     {
299         long double temp;
300         // O - Expected
301         temp = (long double)data[i]    (expected[i]*1000000);
302         // (O E)^2
303         temp = temp * temp;
304         // (O-E)^2/E
305         temp = temp / (expected[i]*1000000);
306         // Sum
307         chiSqr = chiSqr + temp;
308     }
309     // Return to be summed
310     return chiSqr;
311 }
```

Appendix C

Differential Attack Implementation Code

```
1  /*****************************************************************************
2
3    Filename: differential_3_round_version.cpp
4    Author: Brian Kidney, P.Eng
5
6    Description:
7     Test to determine if the correct key can be obtain using a three round
8      version of CIKS-1 and the 10 > 01 > 11 differential.
9
10 *****************************************************************************/
11 #include "diff_3_round_version.h"
12 #include "util.h"
13
14 struct State
15 {
16         char L[32];
17         char R[32];
18         char K[32];
19 };
20
21
22 void three_round_diff_create_ciphertext()
23 {
24     const int rounds = 3;
25
26     char keys[32][rounds];
27     char diff_keys[32][rounds];
28
29     long* seed = new long;
30     *seed = -1 * time( NULL );
31
32     int total_test_size = 100000;
33
34     // Initialize keys to 4 random values to be use for the entire test
35     for (int p = 0; p < rounds; p++)
36     {
37         long key = ( ((int)((rand1(seed) * (float)(0xFFFF))) << 16)
38             + (int)((rand1(seed) * (float)(0xFFFF))) );
39         int mask = 0x00000001;
40         for (int i = 0; i < 32; i++)
41         {
42             keys[i][p] = (char)((bool)(mask & key));
43             diff_keys[i][p] = (char)((bool)(mask & key));
44
45             mask <<= 1;
46         }
47     }
48     char actualKey[32];
49     memcpy(actualKey, &keys[0][rounds-1], 32);
```

```
50      cout << "Actual key: " << create_ulong(actualKey) << endl;
51
52      for (int i = 0; i < total_test_size; i++)
53      {
54          // Set random values for the left and right plaintext
55          long left = ( ((long)((rand1(seed) * (float)(0xFFFF))) << 16)
56              + (long)((rand1(seed) * (float)(0xFFFF))) );
57          long right = ( ((long)((rand1(seed) * (float)(0xFFFF))) << 16)
58              + (long)((rand1(seed) * (float)(0xFFFF))) );
59
60          // Initialize state variables
61          State state;
62          State diff_state;
63
64          // The left and right values to be unaltered.
65          create_array(state.L, left);
66          create_array(state.R, right);
67
68          // The left and right values to be altered by 1 bit
69          create_array(diff_state.L, left);
70          create_array(diff_state.R, right);
71
72          // Introduce the difference into any of the 32 bits
73          int difference_bit = (int)(rand1(seed) * 32.0);
74          if (diff_state.L[difference_bit] == 0)
75              diff_state.L[difference_bit] = 1;
76          else
77              diff_state.L[difference_bit] = 0;
78
79          for (int j = 0; j < rounds; j++)
80          {
81              // Encrypt 1 round
82              memcpy(state.K, &keys[0][j], 32);
83              memcpy(diff_state.K, &keys[0][j], 32);
84              ciks1_round(state);
85              ciks1_round(diff_state);
86
87          }
88
89          // Record the result
90          cout << create_ulong(state.L) << " "
91               << create_ulong(state.R) << " "
92               << create_ulong(diff_state.L) << " "
93               << create_ulong(diff_state.R) << endl;
94      }
95      delete seed;
96  }
97
98  void three_round_diff_key_score()
99  {
100     int good_differential_count = 0;
101
102     // Load the file
103     fstream file;
104     file.open("ciphertext.txt");
105     if (!file)
106     {
107         cout << "Cannot open file." << endl ;
108     }
109     else
110     {
111
112         string temp1, temp2;
113         unsigned long key;
114         unsigned long l;
115         unsigned long r;
116         unsigned long dl;
117         unsigned long dr;
```

```
118
119         file >>temp1>>temp2>>key;
120
121         // For each data set
122         while (file >>l>>r>>dl>>dr)
123         {
124             // Set up data variables
125             char bL[32];
126             create_array(bL, l);
127             char bR[32];
128             create_array(bR, r);
129             char bDL[32];
130             create_array(bDL, dl);
131             char bDR[32];
132             create_array(bDR, dr);
133
134             char bK[32];
135             create_array(bK, key);
136             char bDK[32];
137             create_array(bDK, key);
138
139             int total_diff_left = 0;
140             int total_diff_right = 0;
141
142             State norm;
143             State diff;
144
145             memcpy(norm.L, bL, 32);
146             memcpy(norm.R, bR, 32);
147             memcpy(norm.K, bK, 32);
148
149             memcpy(diff.L, bDL, 32);
150             memcpy(diff.R, bDR, 32);
151             memcpy(diff.K, bDK, 32);
152
153             // decript with key.
154             ciks1_round_inv(norm);
155             ciks1_round_inv(diff);
156
157             for (int k = 0; k < 32; k++)
158             {
159                 if (norm.L[k] != diff.L[k])
160                     total_diff_right++;
161                 if (norm.R[k] != diff.R[k])
162                     total_diff_left++;
163             }
164
165             // see if there is a 1 - 1 difference.
166             // if so add 1 to count
167             if (total_diff_right == 1 && total_diff_left == 1)
168                 good_differential_count++;
169         }
170         // Output count.
171         cout << "Actual Count "<<  good_differential_count << endl;
172
173         for (int j = 0; j < 32; j++)
174         {
175             good_differential_count = 0;
176
177             file.close();
178             file.open("ciphertext.txt");
179
180             string temp1, temp2;
181             unsigned long key;
182             unsigned long l;
183             unsigned long r;
184             unsigned long dl;
185             unsigned long dr;
```

```
186
187                 file >>temp1>>temp2>>key;
188
189                 // For each data set
190                 while (file >>l>>r>>dl>>dr) {
191
192                     // Set up data variables
193                     char bL[32];
194                     create_array(bL, l);
195                     char bR[32];
196                     create_array(bR, r);
197                     char bDL[32];
198                     create_array(bDL, dl);
199                     char bDR[32];
200                     create_array(bDR, dr);
201
202                     char bK[32];
203                     create_array(bK, key);
204                     char bDK[32];
205                     create_array(bDK, key);
206
207                     if (bK[j] == 0)
208                         bK[j] = 1;
209                     else
210                         bK[j] = 0;
211
212                     if (bDK[j] == 0)
213                         bDK[j] = 1;
214                     else
215                         bDK[j] = 0;
216
217                     int total_diff_left = 0;
218                     int total_diff_right = 0;
219
220                     State norm;
221                     State diff;
222
223                     memcpy(norm.L, bL, 32);
224                     memcpy(norm.R, bR, 32);
225                     memcpy(norm.K, bK, 32);
226
227                     memcpy(diff.L, bDL, 32);
228                     memcpy(diff.R, bDR, 32);
229                     memcpy(diff.K, bDK, 32);
230
231                     // decript with key.
232                     ciksl_round_inv(norm);
233                     ciksl_round_inv(diff);
234
235                     for (int k = 0; k < 32; k++)
236                     {
237                         if (norm.L[k] != diff.L[k])
238                             total_diff_right++;
239                         if (norm.R[k] != diff.R[k])
240                             total_diff_left++;
241                     }
242
243                     // see if there is a 1 - 1 difference.
244                     // if so add 1 to count
245                    if (total_diff_right == 1 && total_diff_left == 1)
246                        good_differential_count++;
247                 }
248                 // Output count.
249                 cout <<  good_differential_count << endl;
250             }
251         }
252     }
```