COMPLEXITY ANALYSIS AND HARDWARE IMPLEMENTATION OF EXTENSIBLE MODULO ADDITION FOR LIGHTWEIGHT BLOCK CIPHER IN INTERNET OF THINGS (IOT)

By

Sheraz Raza Siddique

Bachelor of Electronics Engineering

Sir Syed University of Engineering and Technology, Pakistan 1999

Master of Computer Science

University of Karachi, Pakistan, 2001

A project

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Engineering

In the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2017

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this project. This is a true copy of the project, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this project to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this project by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my project may be made electronically available to the public.

Project:	Complexity analysis and hardware implementation of extensible modulo addition for lightweight block cipher in Internet of Things (IoT)
Degree:	Master of Engineering
Year of convocation:	2017
Name:	Sheraz Raza Siddique
Program:	Electrical and Computer Engineering
University:	Ryerson University

ABSTRACT

This project presents complexity analysis and hardware implementation of extensible modulo addition [15] encryption algorithm on a 32-bit lightweight FPGA based block cipher called INFLEX, which is designed for the internet of things (IoT) environment, supporting 64-bits key. It is designed for constrained hardware resources yet providing a highly secure scalable configuration for the variety of applications. This characteristic is obtained by the use of generalized Feistel structure combined with an improved block inflation feature. INFLEX follows a typical ARX (Add, Rotate, XOR) round function with a distinguished feature of block expansion and collapse as per user selected control string, which makes INFLEX act as a tweakable Cipher. We have shown comparison of INFLEX algorithm robustness and immunity against linear and differential attacks and demonstrated that it outperforms one of the benchmark block Ciphers Speck32/64 proposed by national security agency (NSA).

ACKNOWLEDGEMENT

As a graduate student in the electrical and computer engineering department at Ryerson University, I had quite a prolific and conversant experience during my master of engineering program.

First of all I would like to thank Professor Dr. Reza Sedaghat for all the motivation and encouragement behind this project. I had an opportunity to learn under his mentorship the subject of VLSI, which spurred us to pursue further on the subject and experience practical implementation in the real life project. I was one of the lucky students to get to work with Professor Reza on an exciting project. He introduced us to the world of cryptology and encouraged us to work on the development of a light weight cipher.

The exciting project would not have been materialized without the great help and guidance of Prathap (Patrick) Siddavaatam. Patrick ensured that we were always on track and guided us through the ebb and flow of the route to success. He always made himself available during his busy schedule, I am highly indebted to him for his help.

I would also like to thank my fellow students whose names are not mentioned here but they, apart from helping, kept life interesting, happy and colourful.

Finally, I would like to thank my family for their support, patience and encouragement, without their help and understanding I would not have achieved what this project is all about.

TABLE OF CONTENTS

List of Tables

List of Figures

Nomenclature

$\boxplus$	Addition Modulo 2^n
$\boxminus$	Subtraction Modulo 2^n
$\oplus$	Exclusive OR operation
$x <<< y$	left rotation of *x* by *y* bits
$x >>> y$	right rotation of *x* by *y* bits
Adversary	A malicious entity or attacker of cryptosystem
ARX	Add, rotate, Xor
Ciphertext	Encrypted output of a cipher
Confusion	Obscures the relationship between the Plaintext and Ciphertext.
Cryptography/Cryptology	is the practice and study of techniques for secure communication in the presence of adversaries
Deflation $\mathfrak{D}$.	The operation of extracting original bits from the expanded bits of a data block.
Diffusion	Dissipates the redundancy of the Plaintext by spreading it out over the Ciphertext.
Feistel	a symmetric structure used in the construction of block ciphers
G.E	Gate equivalent
Inflation $\mathcal{I}$	expanding message block bits by a user selected factor
IV	Initialization vector
KI	Input control bit string
KO	Output control bit string
LFSR	Linear feedback shift register
m	User defined control parameter
NSA	National Security Agency
Plaintext	Unencrypted input to a cipher
SPN	Substitution permutation network
Spoofing:	a person's or program's pretence as another entity

1. Introduction

1.1 Overview

With the rapid evolution of internet of things (IoT), where the number of connected devices to IoT is expected to reach over 50 billion by 2020 [1] from current 7 billion connected devices. The security industry is seeing a paradigm shift of not only managing identity and access management (IAM) of people or financial transactions but also managing hundreds of thousands of devices that may be connected to a network. With the adoption of advance encryption standard (AES) in 2001, AES became the preferred choice for any block cipher application and the need for new block cipher had greatly reduced. However for the constrained environment such as RFID tags or sensors network AES becomes over exorbitant. Rapid increase in the number of IoT devices gave an unprecedented challenge to the industry and security community to stay abreast with the development of smart, efficient and compact architecture of security algorithms in conformity with the compact architecture of IoT devices. The usage of cryptography has increased phenomenally in the last decades, as the use of the internet has become rampant. The elementary goal of cryptography is to enable two parties to communicate confidentially over a communication channel. This means that any adversary should not be able to intercept the message in its original form also known as Plaintext. The simplest definition of cryptography is the performance of encryption on Plaintext and decryption on the coded or Ciphertext. Where encryption means the transformation of the Plaintext into the Ciphertext and decryption is the reverse of encryption where the Ciphertext is used as input into transformation to recover the Plaintext.

Encryption is a part of our daily life although it is mostly invisible. It is used to prevent eavesdropping on communications over text, voice and video through any electronic communication channel, securing network connections, making e-commerce and e-banking possible and generally securing information from any unauthorized interception. Through the history of civilization, new encryption functions have been constructed as old ones have been broken.

In this thesis we introduce a new block cipher INFLEX, which has been carefully designed on Feistel network structure with a goal of performance and power consumption improvement when compared with similar type of block ciphers. Inspired by Speck cipher [13], INFLEX inherits basic architecture of Speck with a modified round function, and introduces a secondary

level of diffusion by a user selected scalable complexity level. Scalable complexity selection makes INFLEX a tweakable cipher, where cipher can be tuned as per the required complexity level. The improved round function is intended to craft a new cipher with improved security and complexity yet not compromising performance and silicon area, this shall be proved in subsequent sections of this thesis that along with security and complexity increase, performance and silicon area efficiency also increased or at least kept similar, except for application requiring extra high security. We will describe our proposed block cipher, after a brief background of ciphers, INFLEX is presented in details, followed by the cryptanalysis and the thesis is concluded with conclusion and future work remarks.

1.2 Motivation

Encryption is essential in the IoT environment as well as many other RFID authentication and Wireless Sensor Network (WSN) applications. Numerous encryption frameworks exist which are mostly based on software algorithms. However, most of the modern radio transceiver chip used in home automation, wireless alarm and security system and automatic meter reading (AMR) are equipped with an integrated hardware encryption engine. There exist several hardware encryption engines customized for a specific application but fewer are scalable and flexible for a variety of applications.

1.3 Our Contribution

In this project we have utilized the adaptive security framework with extensible computational complexity for cipher system, proposed by Siddavaatam et. al [15] to design and implement a lightweight block cipher intended for IoT and WSNs. Based on the proposed extensible computational complexity concept of the cipher, we have enhanced the typical ARX structure of the cipher round function based on a Feistel structure, implemented the message block expansion as per user selected control string KI to introduce a secondary level of diffusion, which contributes to increased security and makes the cipher design tweakable, customizable as per security requirements of the application. We have simulated a hardware implementation of the proposed design on Altera Cyclone IV family of FPGA using VHDL and presented results in terms of complexity computation, throughput calculations, gate equivalent estimation and provided a comparison with Speck32/64 (a 32-bit block cipher designed by NSA) in terms of linear and differential cryptanalysis (a practice adopted by the security community to analyze any cipher design).

1.4 Cryptography Algorithms and Keys

Cipher

In Cryptography Cipher is an algorithm for performing encryption or decryption. A series of well-defined steps that can be followed as a procedure. In common parlance Cipher is synonym with Code as they both are set of steps that encrypt a message, however the concept are distinct in cryptography.

Codes generally substitute different length strings of characters in the output, while cipher generally substitute same numbers of characters as are input. There are exceptions and some cipher systems may use slightly more, or fewer, characters when output versus the number that were input. When using a cipher the original information is known as *Plaintext*, and the encrypted form as *Ciphertext*. The Ciphertext message contains all the information of the Plaintext message, but is not in a format readable by a human or computer without a decrypt to it.

The operation of a cipher usually depends on a piece of auxiliary information called *Key*, a Key must be selected before using a cipher to encrypt a message, without the knowledge of the Key, it should be extremely difficult, if not impossible to decrypt the Ciphertext into a readable plaintext. The range of possible values of the key is called the *keyspace*. Both the encryption and decryption operations use this key (i.e. they are dependent on the key and this fact is denoted by the k subscript), so the encryption and decryption function now becomes:

$E_k(P) = C$

$D_k(C) = P$

These functions have the property

$D_k(E_k(P)) = P$

Some functions use a different encryption key K_1 and a different decryption key K_2. So that:

$E_{k1}(P) = C$

$D_{k2}(C) = P$

$D_{k2}(E_{k1}(P)) = P$

1.5 Public Key Algorithms

Public-key algorithms (also called asymmetric algorithms) are such that the key used for encryption is different from the key used for decryption. Further the decryption key cannot be calculated in a reasonable time from the encryption key. Such algorithms are call "*public-key*" because the encryption key can be made public. The encryption key is often called the public key, and the decryption key is often called the private key.

1.6 Symmetric Key Algorithms

Symmetric key algorithms are designed to be very fast and have a large number of possible keys, they are quickest and most commonly used type of encryption. In most symmetric algorithms, the encryption key and the decryption key are the same. These algorithms, also called secret-key algorithms, single-key algorithms, or one-key algorithms, require that the sender and receiver agree on a key before they can communicate securely. The security of a symmetric algorithm rests in the key; divulging the key means that anyone could encrypt and decrypt messages. As long as the communication needs to remain secret, the key must remain secret. There are many well-known symmetric key algorithms e.g. DES, RC2, RC4, IDEA etc.

1.7 Key Length

Symmetric Key Length

The security of a symmetric cryptosystem is a function of two things, 1) the strength of the algorithm and 2) the length of the key. The former is more important but the latter is easier to demonstrate. Assuming perfect strength of algorithm, which means there is no better way to break the cryptosystem other than trying every possible key through a brute-force attack. To launch this attack, a cryptanalyst needs a small amount of Ciphertext and the corresponding Plaintext generally 64 bits. If the key is 8 bits long there are 2^8, or 256 possible keys. Therefore it will take 256 attempts to find the correct key, with a 50% probability of finding the key after half of the attempts. If the key is 64 bits long then a supercomputer trying a million keys a second will take 585,000 years to find the correct key among the 2^{64} possible keys. Therefore before inventing a cryptosystem with a 64 or 256 bit key, the strength of the algorithm must be taken care of such that there is no better way to break it than with a brute-force attack. Despite this, the security of cryptosystem should rest in the key and not in the details of the algorithm. Assume that a cryptanalyst has access to all details of the algorithm, access to Ciphertext and Plaintext, he can mount a Ciphertext only attack, a Plaintext attack or even a chosen-Plaintext

attack, and if the cryptosystem can remain secure even with all that knowledge in adversary's hands the system is secure.

Public-Key Key Length

It is easy to multiply two number to get a product but hard to factor the product and recover the two large prime. Public-key cryptography uses this idea to make a trap-door one-way function. Public-key consists in several big integer values and a private key consists in also some integer values. The length of a key is the length in bits of the modulus. When a public key is said to have a length of 2048 bits, it really means that the modulus value lies between 2^{2048} and 2^{2047}. Since the public and private key of a given pair share the same modulus, they also have by definition the same length. To determine how long a key should be, requires to look at both the intended security and lifetime of the key and the current state-of-the-art factoring. Today you need a 2048-bit number to get the level of security you got from 1024-bit number 10 years back. If you want your keys to remain secure for 20 years, 2048 bit is likely too short.

1.8 Security of Algorithms

Different algorithms offer different degrees of security, it depends on how hard they are to break. If the cost required to break an algorithm is more than the value of the encrypted data, then algorithm is considered safe.

The algorithm breaking can be classified as:

1. **Total break.** A cryptanalyst finds the key *K* such that $D_K(C) = P$.
2. **Global deduction.** A cryptanalyst finds an alternate algorithm to decrypt the Ciphertext such that $D_K(C)$ without knowing *K*.
3. **Instance (or local) deduction.** A cryptanalyst finds the Plaintext of an encrypted Ciphertext.
4. **Information deduction.** A cryptanalyst gains some information about the key or Plaintext. This information could be a few bits of the key or the Plaintext and so forth.

An algorithm is unconditionally secure if no matter how much Ciphertext a cryptanalyst has, there is not enough information to recover the Plaintext. For this classification only a one-time pad is unbreakable given infinite resources. All other cryptosystems are breakable in a Ciphertext-only attack, simply by trying all possible key(s) one by one and checking if Plaintext is meaningful. This is called *brute-force* attack.

An algorithm is considered *computationally secure* (computationally infeasible to break) if it cannot be broken with available resources.

Complexity of an attack can be measured in different ways:

1. **Data complexity.** The amount of data needed as input to the attack.
2. **Processing Complexity.** The time needed to perform the attack.
3. **Storage requirements.** The amount of memory needed for the attack.

If an algorithm has a processing complexity of 2^{128}, then 2^{128} operation are required to break the algorithm, these operations may be complex and infeasible to perform in reasonable time.

Cryptographically Secure Pseudo-Random sequence

A sequence generator is pseudo-random if it has following property:

1. It looks random. This means that is passes all statistical tests of randomness that we can find.

Cryptographic applications demand much more of a pseudo-random-sequence generation than do most other applications. Cryptographic randomness does not mean just statistical randomness. For a sequence to be *cryptographically secure pseudo-random,* it must also have the following property:

2. It is unpredictable. It must be computationally infeasible to predict what the next random bit will be, given complete knowledge of the algorithm or hardware generating the sequence and all of the previously generated bits in the stream.

Cryptographically secure pseudo-random sequences should not be compressible unless the key is known. The key is generally the seed used to set the initial state of the generator.

A sequence generator is *real random* if it has this additional property:

3. It cannot be reliably reproduced. If the sequence generator is run twice with the exact same input, it should generate two completely unrelated random sequence.

The output of a generator satisfying these three properties will be sufficient enough for a one-time pad key generation and any other cryptographic applications that require a truly random sequence generator.

Time and cost estimates for brute-force attack

A brute-force attack is typically a known-Plaintext attack, it requires a small amount of Ciphertext and corresponding Plaintext. Two parameters determine the speed of a brute-force attack, a) the number of keys to be tested and the speed of each test. DES has a 56-bit key, it has 2^{56} possible keys. Assuming that we only have to try 25% of the possibilities i.e 4.5×10^9 keys, with a dual processor 3.2 Ghz which can process 4,000,000 keys per second, it would take 142 years, and with a 64-bit key for brute-forcing only 25% of key-space it would take 36,558 years. Therefore a 64 bit key seems safe from a worst case brute-force attack using current available technology.

How long should a Key Be?

There is no single answer to this, it depends on the situation, and how much security is required depends on how much data is worth, how long does it need to be secure and what are the resources that adversaries have got. Security requirement may be specified in these terms:

The Key length must be such that there is a probability of no more than 1 in 2^{32} that an attacker with $100 million to spend could break the system within one year, even assuming technology advances at a rate of 30 percent per annum over the period.

Public-Key Cryptography versus Symmetric Cryptography

The number and length of messages are far greater with public-key algorithms than with symmetric algorithms. Symmetric algorithms were concluded more efficient than the public-key algorithms by some researchers. Although true but this analysis over looked some significant security benefits of public-key cryptography.

Symmetric cryptography is best for encrypting data. It is faster and is not prone to chosen-Ciphertext attacks. Public-key cryptography can do things that symmetric cryptography can't, it is best for key management.

1.9 Application of Encryption

Encrypting Communications Channels

This encryption can take place at any layer in the Open Systems Interconnect (OSI) communications model. It takes place either at the lowest layers (one and two) or at the higher layers. If it takes place at the lowest layers, it is called *link-by-link encryption*, everything going through a particular data link is encrypted. If it takes place at higher layers, it is called *end-to*

end encryption, the data are encrypted selectively and stay encrypted until they are decrypted by the intended recipient.

Link-by-link Encryption

The easiest place to add encryption is at the physical layer, which is called link-by-link encryption. The interfaces to the physical layer are standardized and it is easy to connect hardware encryption devices at this point. The encryption device encrypt all data passing through it, including data, routing information and protocol information. Link-by-link encryption is very effective, as everything is encrypted, an adversary cannot get any information about the structure of data passing through. The adversary is not only denied access to information, but also access to the knowledge of where and how much information is flowing. Key management is also not complex, only the endpoints on the line need a common key and they change their key independently from the rest of the network.

The biggest challenge with encryption at the physical layer is that each physical link in the network needs to be encrypted, leaving any link unencrypted can jeopardize the security of the entire network. If the network size is large then the cost can easily become very high for this type of encryption.

Pros:

Easier operation, it can be made transparent to user, since everything is encrypted before sending over the link.

Only one set of key per link is required.

Provides traffic-flow security, since routing information is also encrypted.

Encryption is online.

Cons:

Data is exposed in the intermediate nodes.

End-to-End Encryption

Another approach is to introduce encryption device equipment between the network layer and the transport layer. The encryption equipment must understand the data according to protocols up to layer three and encrypt only the transport data, which is then combined with the unencrypted routing information and sent to lower layers for transmission. This approach

avoids encryption/decryption at the physical layer. In end-to-end encryption the routing information about the data is not encrypted and a cryptanalyst may learn who is communicating with whom and for how long without ever knowing the content of that communication. Key management is also more complicated, since individual users must ensure they have common keys.

Pros:

Higher secrecy level.

Cons:

Requires a more complex key-management system.

Traffic analysis is possible since routing information is not encrypted.

Encryption is offline.

Encrypting Data for Storage

In communication channels messages in transit have no essential value. If recipient does not receive a particular messages the sender can always resend it, however this is not true for data encrypted for storage. If data cannot be decrypted, you cannot undo encryption, it is lost for ever. Unlike encryption for communication where key is only required once for decryption when data is received, data encrypted for storage may sit for years before it is decrypted. A key used for data storage encryption might be needed for years, and hence must be stored securely for years.

Hardware Encryption versus Software Encryption

Until very recently, all encryption products were in the form of specialized hardware. Although software encryption is becoming more prevalent, hardware is still embodiment of choice for high security applications. There are several reasons for this:

- The first is speed, since encryption algorithm have complex operation which are hardware resources intensive and run better on specialized hardware than standard hardware computers.
- Dedicating computer resources for complex algorithms is inefficient, moving encryption to another processor makes the whole system faster.

- An encryption algorithm running on a standard computer has no physical protection and is subject to superstitious modification of algorithm by adversary without anyone ever realizing it. Hardware encryption device can be secured in temper proof enclosures.
- Hardware based encryption devices are easier to install.

Software based Encryption

Any encryption algorithm can be implemented in software, the disadvantage are in speed, cost and ease of modification/manipulation.

Compression, Encoding and Encryption

Using a data compression algorithm together with an encryption algorithm has following advantages:

Encryption is time intensive, compressed data before encryption speeds up the entire process.

Cryptanalysts exploit redundancies in the plaintext, compressing data before encryption reduces these redundancies.

2. Mathematical Interlude

2.1 Cryptographic Algorithms – Mathematical Background

The computational complexities of algorithms are often measured by *time complexity* T and *space complexity* S. Both T and S are commonly expressed as functions of *n,* where *n* is the size of input. Generally algorithms are classified according to their time or space complexities. An algorithm is *constant* if its complexity is independent of *n*: O(1). An algorithm is linear, if its time complexity is O(*n*). Algorithms can also be quadratic, cubic and so on. All such algorithms are polynomial, their time complexity is O(n^m), where *m* is a constant. Algorithms having polynomial time complexities are called as *polynomial-time* algorithms.

Algorithms with complexities in the form of O($t^{f(n)}$) where *t* is a constant greater than 1 and f*(n)* is some polynomial function of n, are called *exponential* algorithms. A subset of exponential algorithms whose complexities are O($c^{f(n)}$) where *c* is a constant and *f(n)* is greater than constant but less than linear are called *super-polynomial* algorithms.

As *n* grows, the time complexity of an algorithm can make enormous difference in whether algorithm is practical. Assuming the unit of "time" for computers in microseconds, the computer can complete a constant algorithm in microseconds, a linear algorithm in seconds, a quadratic algorithm in 11.6 days and 32,000 years to complete a cubic algorithm if $n = 10^6$. Performing the exponential algorithm is pointless no matter how much we extrapolate computing power.

2.2 Complexity of Problems

Problems that can be solved with polynomial-time algorithms are called *tractable,* because they can be solved in a reasonable amount of time for a reasonable size input. Problems that cannot be solved in polynomial time are called *intractable,* because calculating their solution in reasonable time becomes infeasible. Intractable problems are sometimes just called hard.

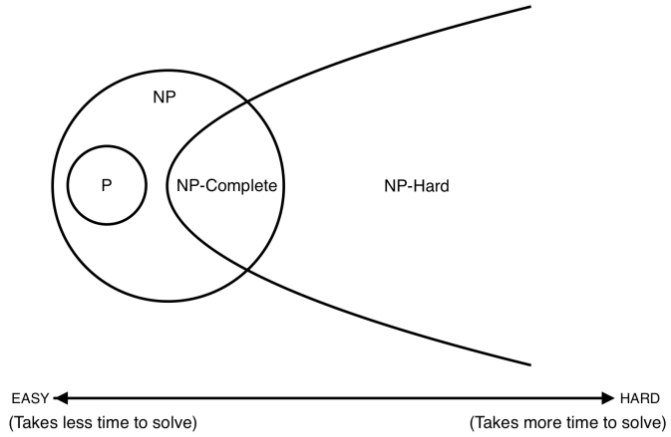


Figure 1. Complexity of problems

The Class P consists of all problems that can be solved in polynomial time. The Class NP consists of all problems that can be solved in polynomial time only on a nondeterministic Turing machine {a variant of a normal Turing machine (a Turing machine is a finite state machine with an infinite read-write memory) that can makes guesses.

As far as cryptography is concerned with NP, many symmetric algorithms and all public-key algorithms can be cracked in nondeterministic polynomial time. Given a Ciphetext *C*, the cryptanalyst simply guesses a Plaintext *P* and a key *k* and in polynomial time runs the encryption algorithm on input *P* and *k* to check if the result is equal to *C*. This is important because it puts an upper bound on the complexity of cryptanalysis for these algorithms.

2.3 Modular Arithmetic:

The set of integers from 0 to n-1 form what is called a complete set of residues modulo *n*. which means that for every integer *a*, its residue modulo *n* is some number from 0 to *n-1*.

$a \equiv b(mod\ n)$ if $a = b+kn$ for some integer *k*. if *a* is non-negative and *b* is between 0 and *n*, then *b* is the remainder of *a* when divided by *n*. *b* is sometimes called as residue of *a*, modulo *n*. Sometimes *a* is called congruent (the triple equals sign) to *b*, modulo *n*.

Modular arithmetic is just like normal arithmetic, it is commutative, associative and distributive.

$(a + b) mod\ n = \{(a\ mod\ n) + (b\ mod\ n)\}\ mod\ n$

$(a - b) mod\ n = \{(a\ mod\ n) - (b\ mod\ n)\}\ mod\ n$

$(a * b) mod\ n = \{(a\ mod\ n) * (b\ mod\ n)\}\ mod\ n$

$\{a * (b + c)\}\ mod\ n = [\{(a * b)\ mod\ n\} + \{(a * c) mod\ n\}]\ mod\ n$

Cryptography makes lots of use of the computation *mod n*, because calculating discrete logarithms and square roots *mod n* can be hard problems. Modular arithmetic is also easier to work with on computers. For a *k*-bit modulus, *n*, the intermediate result of any addition, subtraction or multiplication will not be more than *2k* bits long. So we can perform exponentiation in modular arithmetic without generating huge intermediate results.

For example if we want to calculate 2^8 mod n, we would not use the naïve approach and perform seven multiplication and one huge modular reduction:

$(a * a * a * a * a * a * a * a)\ mod\ n$

Instead, perform three smaller multiplications and three smaller modular reductions:

$((a^2\ mod\ n)^2\ mod\ n)^2\ mod\ n$

And with the same methodology

$a^{16}\ mod\ n = (((a^2\ mod\ n)^2\ mod\ n)^2\ mod\ n)^2\ mod\ n$

Computing $a^x mod\ n$, where x is not a power of 2 is slightly harder. Binary notation expresses x as sum power of 2 i.e 25 is 11001 in binary so $25 = 2^4 + 2^3 + 2^0$.

$$a^{25} mod\ n = (a * a^{24}) mod\ n = (a * a^8 * a^{16}) mod\ n$$
$$= (a * ((a^2)^2)^2 * (((a^2)^2)^2)^2)\ mod\ n = ((((a^2 * a)^2)^2)^2 * a) mod\ n$$

With careful storing of intermediate results, we only need six multiplications

$$\left(\left(\left(\left(\left(\left((a^2 mod\ n) * a\right) mod\ n\right)^2\ mod\ n\right)^2\ mod n\ n\right)^2\ mod\ n\right) * a\right)\ mod\ n$$

This is called *addition chaining,* or the binary square and multiply method, it uses a simple and obvious chain based on binary representation.

This technique reduces the operation to, on the average, 1.5*k operations, if *k* is the length of the number *x* in bits. Finding the calculation with fewest operations is a hard problem, but it is not too hard to get the number of operations down to 1.1*k or better, as *k* grows.

Inverse Modulo a Number

The multiplicative inverse of 3 is 1/3, because 3*1/3 = 1. In the modulo arithmetic the problem is more complicated

$a^{-1} \equiv x(mod\ n)$

The modular inverse problem is a lot more difficult to solve. Sometimes it has a solution and sometimes not. For example the inverse of 5 modulo 14 is 3. On the other hand there is no inverse for 2 modulo 14.

In the modular arithmetic we do not have a division operation. However, we find inverse as under:

$a^{-1} \equiv x(mod\ n)$ or $a * \ a^{-1} \equiv 1(mod\ n)\ or\ (a * \ a^{-1})\ mod\ n \ \equiv 1$

Only the numbers coprime to x (numbers that share no prime factors with x) have a modular inverse (mod x)

A *naïve method* of finding a modular inverse of *a (mod x)* is:

- Calculate a * b (mod x) for b values 0 through x-1
- The modular inverse of a mod x is the b value that makes a * b (mod x) = 1

Example a = 5, x = 14

$5 * 0 \ \equiv 0\ (mod\ 14)$

$5 * 1 \ \equiv 5\ (mod\ 14)$

$5 * 2 \ \equiv 10\ (mod\ 14)$

$5 * 3 \ \equiv 15 \ \equiv 1\ (mod\ 14)$

So the inverse of 5 modulo 14 is 3

This method of finding inverse modulo seems slow, there are other faster method of finding inverse modulo, one of them is called *extended Euclidean algorithm.*

Modulo 2 arithmetic is performed digit by digit on binary numbers, each digit is considered independently from other. Numbers are not carried or borrowed. Addition is performed using and exclusive OR (Xor) operation on the corresponding binary digits for each operand.

Addition of two binary numbers, X and Y shown below:

(X) 101101100

(Y) <u>001011010 ⊞</u>

(Z) 1001 10110

2.4 Probability Theory

Events are represented by sets (and set algebra), and probability is a measure on the sets, taking values between zero (for the null set) and one (for the universal set), that satisfies the following equations:

$P(A \cup B) = P(A) + P(B) - P(A \cap B)$ ----------(1)

where $P(A)$ and $P(B)$ are the probabilities of events A and B; $P(A \cup B)$ is the probability of the union (either A or B, denoted, $\cup$); and $P(A \cap B)$ is the probability of the conjunction, or

intersection of events (both A and B, denoted, $\cap$). If the intersection is the null set, probability is additive across mutually exclusive events.

The probability of an event, A, and its complement, A' (where A' is the complement of A, or not- A), sum to one, because an event and its complement are defined to be mutually exclusive and exhaustive; hence,

$P(A) = 1 - P(A'))$. ---------- (2)

If A' is the null set, it is impossible and A is a certainty, so $P(A')$ = 0 and $P(A)$ = 1.

The probability of the conjunction of events can be written as follows:

$P(A \cap B) = P(A|B)P(B) = P(B|A)P(A)$ ---------- (3)

where $P(A|B)$ and $P(B|A)$ are the conditional probabilities of A given B, and of B given A, respectively. The probability of a simple event, A, can also be expressed as follows:

$P(A) = P(A \cap B) + P(A \cap B')$ ---------- (4)

because A can either occur with B or without B (there are no other ways), and the intersection of B and B' is empty, by definition.

For two constituent events, A and B, one could ask twenty simple probability questions, including 4 probabilities of the events and their complements [$P(A)$, $P(A')$, $P(B)$, $P(B')$], 4 conjunctions [$P(A \cap B)$, $P(A \cap B')$, $P(A' \cap B)$, $P(A' \cap B')$], 4 unions [$P(A \cup B)$, $P(A \cup B')$, $P(A' \cup B)$, $P(A' \cup B')$], and 8 conditionals [$P(A|B)$, $P(A|B')$, $P(A'|B)$, $P(A'|B')$, $P(B|A)$, $P(B'|A)$, $P(B|A')$, $P(B'|A')$].

However, among these 20 probabilities, there are only 3 degrees of freedom, because once three values are known [for example, $P(A)$, $P(B)$, and $P(A \cap B)$], the remaining 17 can be calculated from the equations. Furthermore, even these three values are constrained, because $P(A \cap B)$ must be less than or equal to $P(A)$, less than or equal to $P(B)$, and greater than or equal to $P(A)$ + $P(B) - 1$.

For example, if $P(A)$ is .7 and $P(B)$ is .6, then $P(A')$ = .3 and $P(B')$ = .4, by Equation (2). From Equations (3), $P(A \cap B)$ must be less than or equal to .6. Additionally, $P(A \cap B)$ must also be greater than or equal to .3 [by Equation 4, $P(A \cap B)$ + $P(A \cap B')$ = .7 and $P(A \cap B')$ + $P(A' \cap B')$ = .4; therefore, $P(A \cap B)$ = .3 + $P(A' \cap B')$, which is greater than or equal to .3 because $P(A' \cap B')$ > 0]. This constraint also implies that $P(A|B)$ > .5, since $P(A|B)$ =

$(A \cap B)/P(B)$. Suppose $P(A \cap B)$ = .4; from Equation (4), it follows that $P(A \cap B')$ = .3, $P(A' \cap B)$ = .2, and $P(A' \cap B')$ = .1. Equation (1) can then be used to calculate all of the unions, and Equations (3) can be used to calculate all of the conditionals.

Bias

The bias $\epsilon(X)$ of a binary random variable X is defined by $\epsilon(X) = \Pr[X = 0] - \frac{1}{2}$.

$\approx \frac{1}{\epsilon^2(X)}$ samples are required to estimate X [22].

Let X and Y are the input to a nonlinear function such that $Y = f(X)$

We consider the bias of linear combinations of the form

$a.X \oplus b.Y = (\bigoplus_i a_i X_i) \oplus (\bigoplus_i b_i Y_i)$

Example: for a 4 bit vector of X and Y:

$X_2 \oplus X_3 = Y_1 \oplus Y_3 \oplus Y_4$

If the expression hold for 12 out of 16 cases (from 0000 to 1111) then the bias is

$$= \frac{12}{16} - \frac{1}{2} = 0.25$$

3. Ciphers in General

3.1 Cipher Types and Modes

Cipher can be categorized in to several ways:

1. By whether they work on block of symbols (*Block Ciphers*) or continuous stream of symbols (*Stream Ciphers*)
2. By whether the same key is used for both encryption and decryption (*Symmetric Key Algorithm*) or if a different key is used for encryption and decryption (*Asymmetric Key Algorithm)*

There are two basic types of symmetric algorithms: block cipher and stream ciphers:

3.2 Stream Cipher

Stream ciphers convert Plaintext to Ciphertext one bit at a time. A Stream cipher can be symmetric or asymmetric key cipher, but symmetric key stream cipher are widely used. In a stream Cipher, each Plaintext digit is encrypted one at a time with the corresponding digit of the keystream to give a digit of the Ciphertext stream. Since encryption of each digit is dependent on the current state of the cipher, it is also known as state cipher. In practice, a digit is typically a bit and the combining operation is an exclusive-or (XOR).

The pseudorandom keystream is typically generated serially from a random seed value using digital shift registers as $k_1, k_2, k_3, \ldots, k_n$. This keystream, also called as a running key, is Xored with a stream of Plaintext bits, $p_1, p_2, p_3, \ldots, p_n$to produce the stream of Ciphertext bits.

$$c_n = p_n \oplus k_n$$

At the decryption end, the Ciphetext bits are Xored with an identical keystream to recover the Plaintext bits.

$$p_n = c_n \oplus k_n$$

Since

$$p_n \oplus k_n \oplus k_n = p_n$$

The seed value serves as the cryptographic key for decrypting the Ciphertext stream. Stream ciphers represent a different approach to symmetric encryption from block ciphers. Block ciphers process Plaintext in relatively large blocks (e.g., n ≥ 64 bits). The same function is used to encrypt successive blocks; thus (pure) block ciphers are memoryless. In contrast, stream

ciphers process Plaintext in blocks as small as a single bit, and the encryption function may vary as Plaintext is processed; thus stream ciphers are said to have memory.

3.3 Modes of Stream Cipher

(i) **The One-Time Pad:** A vernam Cipher is asymmetrical stream cipher in which the Plaintext is combined with a random or pseudorandom stream of data ("the Keystream") of the same length to generate Ciphertext using the Boolean Xor function which is represented by $\oplus$ and the truth table is as under:

Input		Output
X	Y	Z=X $\oplus$ Y
-	-	+
-	+	+
+	-	+
+	+	-

Table 1. Xor truth table

Other names for this function are Not Equal (NEQ), *modulo* 2 addition (without carry) and modulo 2 subtraction (without borrow).
The Cipher is reciprocal i.e. identical keystream is used to do both encipher and decipher.

Plaintext $\oplus$ Key = Ciphertext
Ciphertext $\oplus$ Key = Plaintext

One-time pad has a perfect security but there are few problems with it, since the key bits must be random and can never be used again and the length of the random key must be equal to the length of the message. A one-time pad might be good for a few short messages but it will not work for high speed communications channels, since the keys need to be random and both the receiver and sender should have the same keys, key storage, transmission and synchronization become challenging for high speed communication, therefore one-time pad is suitable for ultra-secure low-bandwidth channels.

(ii) **Synchronous Stream Cipher**

A synchronous stream cipher is the one in which keystream is generated independently of the Plaintext message and the Ciphertext message.

The encryption process of synchronous stream cipher can be describe by the equation:

$$\sigma_i = f(\sigma_i, k),$$
$$z_i = g(\sigma_i, k),$$
$$c_i = h(z_i, m_i)$$

Where σ_o is the initial state and may be determined by the key k, f is the next-state function, g is the state function which produces the keystream z_i, and h is the output function which combines the keystream and Plaintext m_i to produce Ciphertext c_i.

The encryption and decryption process are shown in the Figure 2 below:

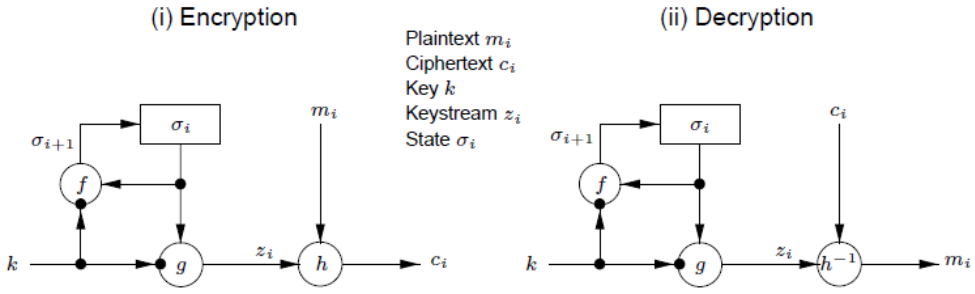


Figure 2. Encryption/decryption process of synchronous stream cipher

Properties of Synchronous Stream Cipher:

a) Synchronization Requirement: Both sender and receiver must be synchronized, using the same key and operating at the same state within that key.

b) No error propagation: A Ciphertext digit which is modified during transmission does not affect the decryption of other Ciphertext digits.

c) Active attacks: As a result of first property the modification of Ciphertext digit by an active adversary causes immediate loss of synchronization and therefore could possibly be detected by the decryptor. As a result of second property an active adversary might possibly be able to make changes to selected Ciphertext digits and obtain the effect on Plaintext. However

additional mechanism must be used to provide data origin authentication and data integrity guarantees.

(iii) **Self-Synchronizing Stream Ciphers**

A *self-synchronizing* or *asynchronous* stream cipher is one in which the key-stream is generated as a function of the key and a fixed number of previous Ciphertext digits.

The encryption function of a self-synchronizing stream cipher can be described by the equations:

$$\sigma_i = (C_{i-t}, C_{i-t+1}, \dots\dots, C_{i-1}),$$
$$z_i = g(\sigma_i, k),$$
$$c_i = h(z_i, m_i)$$

Where $\sigma_0 = (C_{i-t}, C_{i-t+1}, \dots\dots, C_{i-1})$ is the initial state, k is the key, g is the function which produces the keystream z_i, and h is the output function which combines the keystream and Plaintext m_i to produce Ciphertext c_i. The encryption and decryption process is shown in the Figure 3 below.

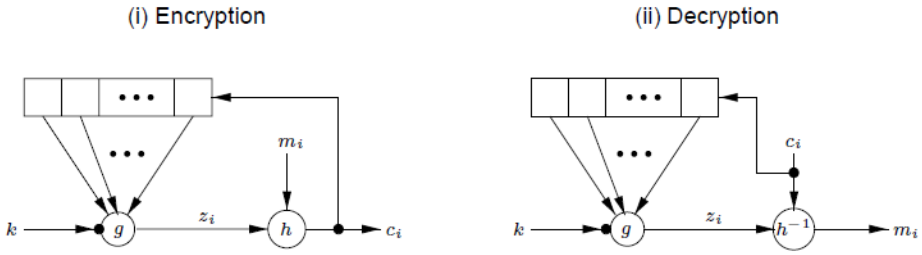


Figure 3. Encryption/decryption process of self-synchronizing stream cipher

Properties of Self-Synchronizing Stream Ciphers

a) Self-synchronization: Since decryption mapping depends only on a fixed number of preceding Ciphertext characters, such ciphers are capable of re-establishing proper decryption automatically after loss of synchronization in case Ciphertext digits are modified.

b) Limited error propagation: If a single Ciphertext digit is modified or even deleted during transmission, then decryption of Ciphertext up to that time may be incorrect, after which correct decryption resumes.

c) Active attacks: Second property implies that any modification of Ciphertext digits by an adversary causes several other Ciphertext digits to be decrypted incorrectly, thereby

improving the likelihood of being detected by the decryptor. As a consequence of first property it is more difficult to detect modification of Ciphertext digits by an adversary.

d) Diffusion of Plaintext statistics: Since Plaintext digit influences the entire following Ciphertext, the statistical properties of the Plaintext are dispersed through the Ciphertext. Self-synchronizing stream ciphers may be more resistant than synchronous stream ciphers against attacks based on Plaintext redundancy.

3.4 Block Cipher

A Block Cipher is a deterministic algorithm operating on fixed length group of bits called blocks. A Block Cipher function maps *n*-bit Plaintext blocks to *n*-bits Ciphertext block. *n* is called the block length. The function is parameterized by a *k*-bit key *K*, taking values from a subset $\mathfrak{k}$ (the key space) of the set of all *k*-bit vectors V_k. It is generally assumed that the key is chosen at random. Use of Plaintext block and Ciphertext block of equal size avoids data expansion.

An *n*-bit cipher block is a function $E : V_n \times K \rightarrow V_n$, such that for each key $K \in \mathfrak{k}$, $E(P,K)$ is an invertible mapping (the encryption function of K) from V_n to V_n. Written $E_K(P)$. The inverse mapping is the decryption function, denoted as $D_K(C)$. $C=E_K(P)$ denotes that Ciphertext C results from encrypting Plaintext P under K.

Whereas Block Cipher generally process Plaintext in relatively large blocks (e.g. $n \geq 64$), stream cipher typically process small unit. This distinction however is not definitive.

A (true) random Cipher is an n-bit block cipher implementing all $2^n!$ Bijections on 2^n elements. Each of the $2^n!$ Keys specifies one such permutation.

The most general block cipher implements every possible substitution. To represent the key of such an *n*-bit (true) random block cipher would require $\lg(2^n!) \approx (n-1.44)2^n$ bits. This excessive bit size makes (true) random Cipher impractical.

3.5 Modes of Block Cipher

Electronic Codebook Mode

Electronic codebook (ECB) is the most obvious way to use a block cipher. A block of Plaintext encrypts into a block of Ciphertext. Since the same block of Plaintext always encrypts to the same block of Ciphertext, it is possible to create a code book of Plaintexts and corresponding Ciphertexts. However if the block size is 64 bits, the codebook will have 2^{64} entries, which is

too large to compute and store, where every key will have a different codebook. This is the easiest mode to work with, since block can be encrypted randomly and not linearly as a file.

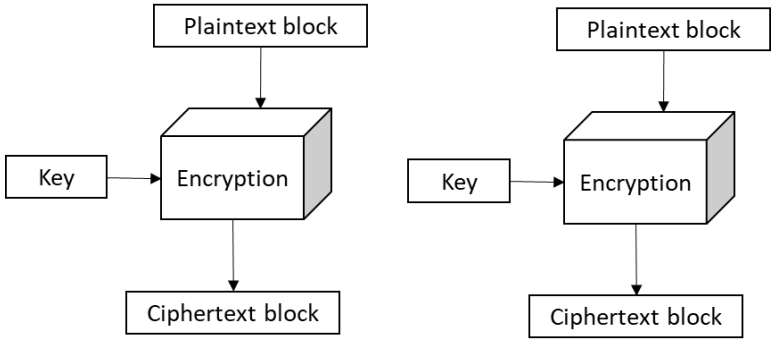


Figure 4. Electronic codebook mode of block cipher

The problem with ECB is that if a cryptanalyst has the Plaintext and Ciphertext for several messages, he can start to compile a code book without even knowing the key, and if there are several redundant messages he can easily decrypt Ciphertext to Plaintext whenever it appears next.

On the plus side, there is no risk encrypting multiple messages with the same key, each block can be looked at as a separate message encrypted with the same key.

Padding

Most messages don't divide neatly into 64-bit (or other size) encryption blocks; there is usually a short block at the end. ECB requires 64-bit blocks. Padding is the way to deal with this problem. Pad the last block with some regular pattern of zeros or ones, alternating ones and zeros to make it a complete block.

Block Replay

A more serious problem with ECB mode is that an adversary could modify encrypted messages without knowing the key, or even the algorithm, in such a way as to fool the intended recipient.

Cipher Block Chaining Mode

Chaining adds feedback mechanism to a block cipher. The results of the encryption of previous blocks are fed back into the encryption of the current block, resulting in each block to modify the encryption of the next block. Each Cipherblock is not only dependent on the Plaintext block that generated it but also on all the previous Plaintext block.

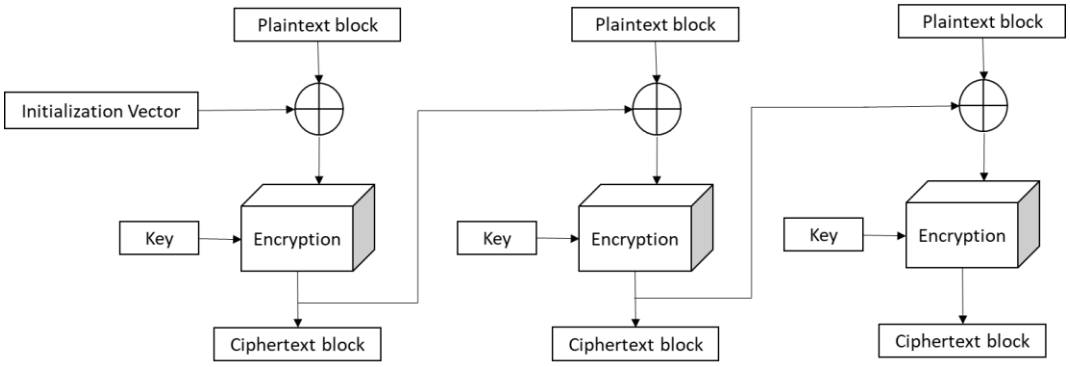


Figure 5. Block chaining mode of block cipher

In cipher block chaining (CBC) mode, the Plaintext is Xored with the previous Ciphertext block before it is encrypted. After a Plaintext block is encrypted to a Ciphertext block the resulting Ciphertext block is also stored in a feedback register to become the next input for the encrypting routine and it continues so on.

Decryption is done normally on a Ciphertext block which is saved in the feedback register. After the next block is decrypted, it is Xored with the results of the feedback register, then the next Ciphertext block is stored in the feedback register and so on.

Mathematically it is represented as:

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

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

Initialization Vector

CBC mode forces identical Plaintext blocks to encrypt to different Ciphertext blocks only when some previous Plaintext block is different, two identical messages will encrypt to the same Ciphertext, giving a cryptanalyst some useful information. This can be prevented by encrypting some random data as the first block. This block of random data is called as the *Initialization vector*. The initialization vector has no meaning, it is just there to make each message unique, and at the decryption end it is just used to fill the feedback register and otherwise ignored. With the addition of the initialization vector identical Plaintext messages encrypt to different Ciphertext messages.

Padding

Padding works just like ECB mode, but in some applications the Ciphertext has to be exactly the same size as the Plaintext.

Error Propagation

CBC mode can be characterized as *feedback* of the Ciphertext at the encryption end and the *feedforward* of the Ciphertext at the decryption end. This has implication on the error propagation, a single bit error in a Plaintext block will affect that Ciphetext block and all subsequent Ciphertext blocks. Ciphertext error are common. They can result from a noisy communication or a storage medium malfunction. This is not significant because decryption will reverse that affect, and the recovered Plaintext will have the same single bit error. Blocks after the second are not affected by the error, so CBC mode is self-recovering. CBC is an example of block cipher being used in a self-synchronizing manner at only the block level.

While CBC mode recovers quickly from bit errors, it cannot recover from synchronization errors. If a bit is lost or added from the Ciphertext block, then all subsequent blocks are shifted one bit out of position and decryption will generate garbage indefinitely.

Security Problems

CBC structure is prone to some potential security problems, since Ciphertext block affect the following block in a simple way, a cryptanalyst can add blocks at the end of the encrypted message without being detected or he can alter a Ciphertext block to introduce controlled changes in the following decrypted Plaintext blocks.

Cipher-Feedback Mode

Block ciphers can also be implemented as a self-synchronizing stream cipher, this is called a cipher-feedback (CFB) mode. With CBC mode, encryption cannot begin until a complete block of data is received. CBC mode cannot encrypt data in byte size. In CFB mode, data can be encrypted in units smaller than the block size, this property makes this block cipher classify as a stream cipher.

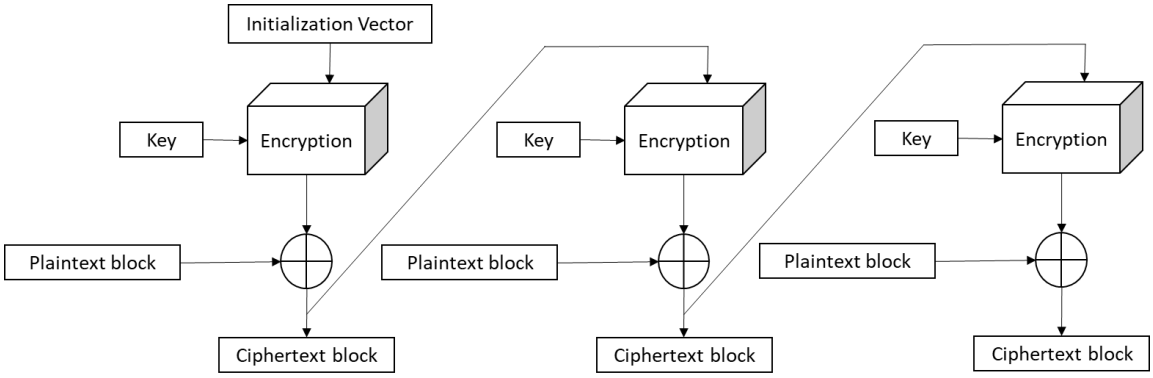


Figure 6. Cipher-feedback mode of block cipher

Output-Feedback Mode

Output-feedback mode (OFB) mode is a method of running a block cipher as a *synchronous stream cipher*. It is similar to the CFB mode except that *n* bits of the previous output block are moved into the right-most positions of the queue. Decryption is reverse of the encryption process. This is called *n*-bits OFB.

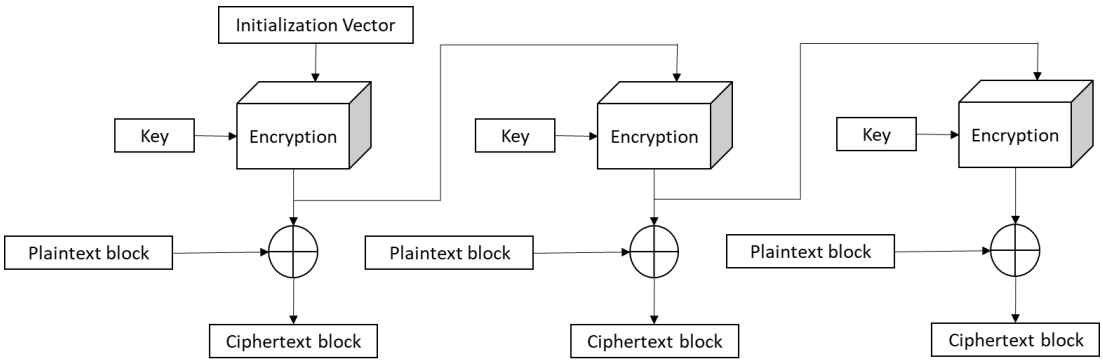


Figure 7. Output-feedback mode of block cipher

If *n* is the block size of the algorithm, then n-bit OFB is:

$C_i = P_i \oplus S_i; S_i = E_K(S_{i-1})$

$P_i = C_i \oplus S_1; S_i = E_K(S_{i-1})$

S_i is the sate, which is independent of either Plaintext or the Ciphertext. One useful feature of the OFB mode is that most of the work can occur offline, before the Plaintext message even received. When the message arrives, it can be XORed with the output of the algorithm to produce the Ciphertext.

Counter Mode

In the counter mode block cipher works similar to a stream cipher. As in the OFB mode, keystream bits are created regardless of content of encrypting data blocks. In this mode, subsequent values of an increasing counter are added to a nonce value (the nonce means a number that is unique: number used once) and the results are encrypted as usual. The nonce plays the same role as initialization vectors in the previous modes.

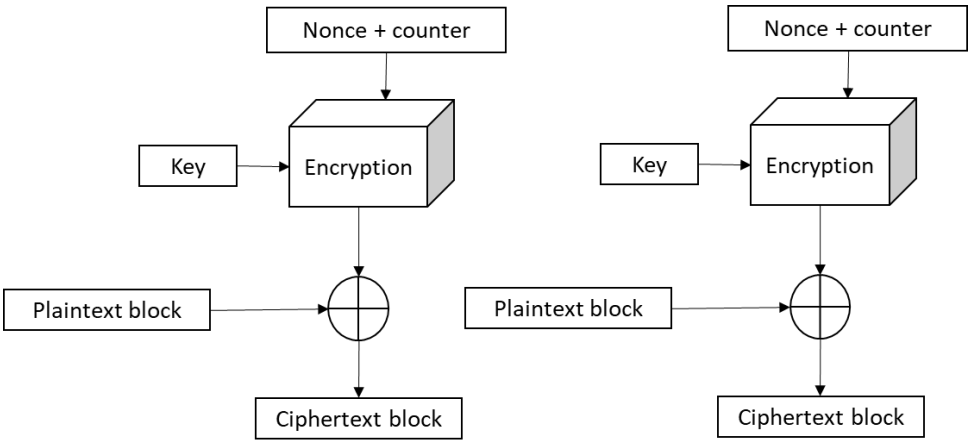


Figure 8. Counter mode of block cipher

3.6 Choosing a Cipher Mode

If simplicity and speed are the main concerns, Electronic Codebook (ECB) is the easiest and fastest mode to use a block cipher, it is also the weakest.

For normal Plaintext, use cipher block chaining (CBC), Cipher feedback (CFB) or output-feedback (OFB) mode. Which mode is chosen depends on the specific requirement. Table 2 below compares security and efficiency of various modes.

CBC is generally best for encrypting files. If the application is software-based CBC is almost always the best choice. Security is significant and while there are sometimes bit errors in stored data, there are almost never any synchronization errors.

ECB	CBC
Security	
- Plaintext patterns are not concealed	+ Plaintext patterns are concealed by Xoring with previous Ciphertext block.
- Input to the block cipher is not randomized. It is same as the Plaintext.	+ Input to block cipher is randomized by Xoring with the previous Ciphertext block.
+ More than one message can be encrypted with the same key	+ More than one message can be encrypted with the same key
- Plaintext is easy to manipulate, blocks can be removed, repeated or interchanged	+/- plaintext is somewhat difficult to manipulate, blocks can be removed from the beginning or end, bits of the first block can be changed, and repetition allows some controlled changes.
Efficiency	
+ Speed is same as the block cipher	+ Speed is same as the block cipher
- Ciphertext is up to one block longer than the Plaintext, due to padding	- Cipher text is up to one block longer than the Plaintext, not counting the initialization vector.
- No Preprocessing is possible	- No preprocessing is possible
+ processing can be done in parallel	+/- Encryption cannot be done in parallel. Decryption can be done in parallel and has a random access property.
Fault tolerance	
- A Ciphertext error affects one full block of Plaintext.	- A Ciphertext error affects one full block of Plaintext and the corresponding bit in the next block.
- Synchronization error cannot be recovered.	- Synchronization error cannot be recovered.
CFB	**OFB**
Security	
+ Plaintext patterns are concealed.	+ Plaintext patterns are concealed.
+ Input to the block cipher is randomized.	+ Input to the block cipher is randomized.

+ More than one message can be encrypted with the same key provided that a different initialization vector is used.	+ More than one message can be encrypted with the same key provided that a different initialization vector is used.
+/- Plaintext is somewhat difficult to manipulate, block can be removed from the beginning and the end of the message, bits of the first block can be changed and repetition allows some controlled changes	- Plaintext is very easy to manipulate, any changes in Ciphertext directly affect the Plaintext.
Efficiency	
+ speed is same as the block cipher	+ speed is same as the block cipher
- Ciphertext is the same size as the Plaintext, not counting the initialization vector.	Ciphertext is the same size as the Plaintext, not counting the initialization vector.
+/- Encryption cannot be done in parallel. Decryption can be done in parallel and has a random access property.	+ Processing is possible even before the message is seen.
- Some preprocessing is possible before a block is seen, the previous Ciphertext block can be encrypted	-/+ OFB processing is not parallelizable, counter processing is parallelizable.
+/- Encryption is not parallelizable, decryption is parallelizable and has random access property.	
Fault Tolerance	
- A Ciphertext error affects the corresponding bit of Plaintext and the next full block.	+ A Ciphertext error affects only the corresponding bit of Plaintext.
+ Synchronization errors of full block sizes are recoverable. 1-bit CFB can recover from the addition or loss of single bits.	- Synchronization error is unrecoverable.

Table 2. Cipher modes comparison

CFB is generally the mode of choice for encrypting stream of characters when each character has to be treated individually. OFB is often used in high speed synchronous system where error propagation is intolerable. OFB is also the mode of choice if preprocessing is required.

One of the four basic modes, ECB, CBC, OFB and CFB is suitable for almost any application. These modes are not overly complex and probably do not reduce the security of the system.

3.7 Block Ciphers versus Stream Ciphers

Although block and stream ciphers are very different, block ciphers can be implemented as stream ciphers and stream ciphers can be implemented as block ciphers. The main difference between two can be defined as:

"Block ciphers operate on data with a fixed transformation on large blocks of Plaintext data, stream ciphers operate with a time varying transformation on individual Plaintext bits".

Block ciphers seem to be more general (they can be used in any of the four modes) and stream ciphers seem to be easier to analyze mathematically. Stream ciphers that only encrypt and decrypt data one bit at a time are not really suitable for software implementation. Block ciphers can be easier to implement in software. On the other hand stream ciphers can be more suitable for hardware implementation. It makes more sense for a hardware encryption device on a digital communication channel to encrypt individual bits as they go by. On the other hand, it makes less sense for a software encryption device to encrypt each individual bit separately.

3.8 Why Block Ciphers

Academic research in block ciphers has progressed along a different course than research in stream ciphers. Block cipher papers have traditionally been concrete designs (with specific parameters and names) or breaks of those designs. Stream cipher papers are more often general design or analysis techniques, with general applications and examples. While stream-cipher cryptanalysis is at least as important as block cipher cryptanalysis, and in military circles more important, it is much harder to develop stream ciphers using existing academic papers.

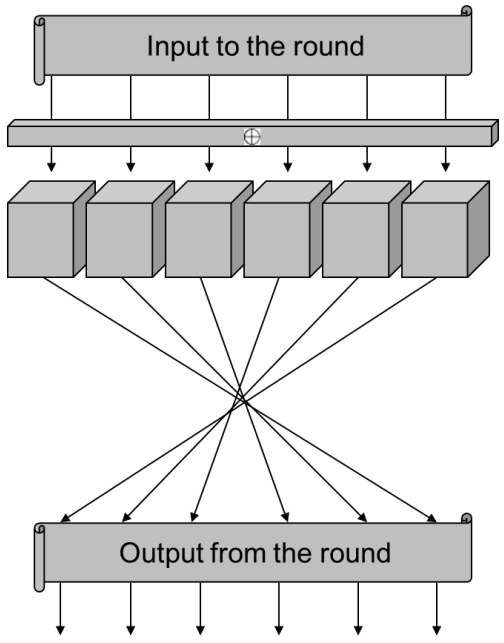
4. Structure of Block Ciphers

Simple encryption algorithms invented long before computers, are based on substitution and transposition of single Plaintext characters. The operations performed in modern encryption algorithms are usually similar but they affect single bits and bytes. Substitution ciphers replace each group of Plaintext letters with another predefined group. Transposition ciphers do not substitute any Plaintext characters but they change positions of the letters. The design model from which many different block ciphers are derived are usually of the following two types:

4.1 Substitution Permutation Networks (SPNs)

A substitution-permutation network (SPN), is a series of linked mathematical operations used in block cipher algorithms such as AES, PRESENT, SAFER, SHARK, and Square.

Such a network takes a block of the plaintext and the key as inputs, and applies several alternating "rounds" or "layers" of substitution boxes (S-boxes) and permutation boxes (P-boxes) to produce the Ciphertext block. The S-boxes and P-boxes transform (sub-)blocks of input bits into output bits. It is common for these transformations to be operations that are efficient to perform in hardware, such as exclusive or (Xor) and bitwise rotation. The key is introduced in each round, usually in the form of "round keys" derived from it.



First, the input is Xored with the round subkey

Second, the input is split into pieces (usually of one byte) and put through a substitution

Finally, the pieces are swapped around

And the output from this round becomes the input to the next round

Figure 9. Single round of SPN

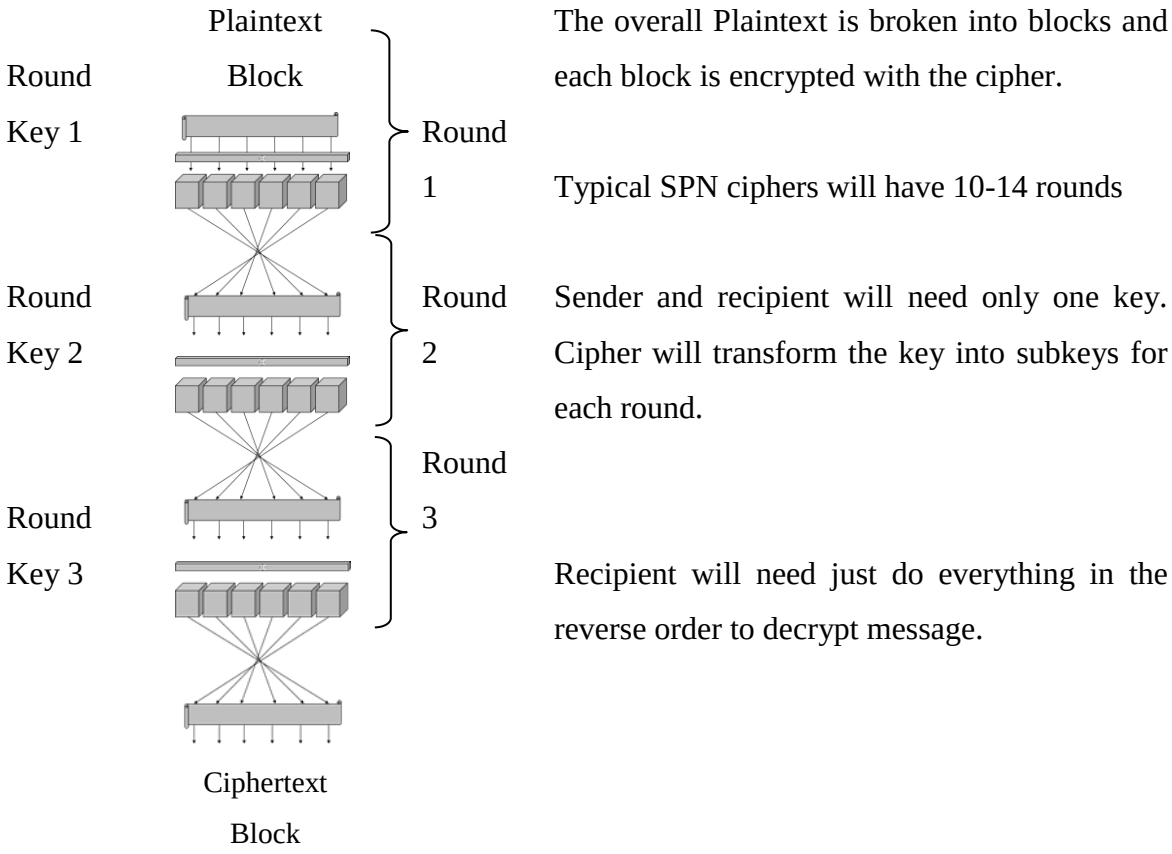


Figure 10. Multiple rounds of SPN

4.2 Feistel Structure

A Feistel cipher is a symmetric structure used in the construction of block ciphers, named after the German-born physicist and cryptographer *Horst Feistel* who did pioneering research while working for IBM (USA); it is also commonly known as a Feistel network. A large proportion of block ciphers use the scheme, including the Data Encryption Standard (DES). The Feistel structure has the advantage that encryption and decryption operations are very similar, even identical in some cases, requiring only a reversal of the key schedule. Therefore, the size of the code or circuitry required to implement such a cipher is nearly halved.

A Feistel network is an iterated cipher with an internal function called a round function.

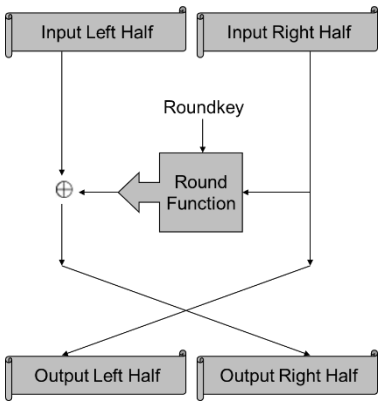


Figure 11. Single round of Feistel

The input to the round is divided into two halves.
The right half goes through a round function with the roundkey.
The output of the round function is Xored with the left half
The two halves switch sides to become the input to the next round

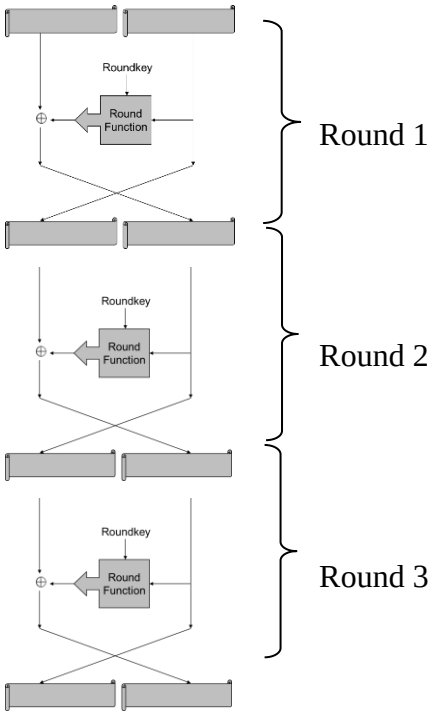


Figure 12. Multiple rounds of Feistel

Feistel ciphers need twice as many rounds as SPN ciphers because only half of the input is being encrypted each round

Works the same as SPN ciphers in terms of transforming one key into subkeys and splitting the Plaintext into blocks

To decrypt, the Ciphertext is sent through the same cipher and the roundkeys are used in reverse order

4.3 Data Encryption Standard (DES)

The Data Encryption Standard (DES), known as Data Encryption Algorithm (DEA) has been a worldwide standard for long time. Although it is showing signs of old age, it has held up remarkably well against years for cryptanalysis and is still secure against most powerful adversaries.

DES is a block cipher, it encrypts data in 64-bit blocks. A 64-bit block of Plaintext goes in one end of the algorithm and a 64-bit block of Ciphertext come out at the other end. DES is a symmetric algorithm i.e. the same algorithm and key are used for both encryption and decryption.

They key length is 56-bits. The key is usually expressed as a 64-bit number, but every eighth bit is used for parity checking. They key can be any 56-bit number and can be changes at any time.

At its simplest level, the algorithm is nothing more than a combination of two basic techniques, *confusion* and *diffusion*. The fundamental building block of DES is a single combination of these techniques on the text based on the key. This is known as a ***round.*** DES has 16 rounds, it applies the same combination of techniques on Plaintext block 16 times.

The algorithm uses only standard arithmetic and logical operations on numbers of 64-bits at most. The repetitive nature of the algorithm makes it ideal for use on a special-purpose chip.

DES operates on a 64-bit block of Plaintext. After an initial permutation, the block is broken into a left and right half, each of 32-bit. Then there are 16 rounds of identical operation, called function *f* where data are combined with the key. After sixteenth round, the right and left halves are joined, and a final permutation, which is inverse of the initial permutation, finishes off the algorithm.

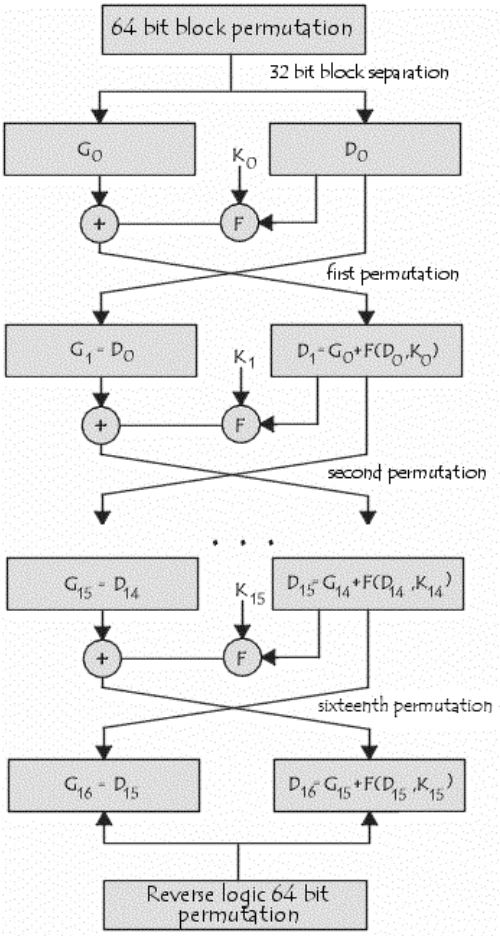


Figure 13. Structure of DES

As shown in the Figure 13. above, in each round the key bits are shifted and then 48 bits are selected from the 56 bits of the key. The right half of the data is expanded to 48 bits via an expansion permutation, combined with 48 bits of a shifted and permuted key via an Xor, sent through 8 S-boxes producing 32 new bit, and permuted again. These four operations make up Function *f*. The output of the Function *f* is then combined with the left half via another Xor. The result of these operations becomes the new right half; the old right half becomes the new left half. These operations are repeated 16 times, making 16 rounds of DES.

The Initial Permutation

The initial permutation occurs before round 1, it transposes the input block by shifting the bit positions, e.g the initial permutation moves bit 58 of the Plaintext to bit position 1, bit 50 to bit position 2, bit 42 to bit position 3, and so on.

The Key Transformation

Initially the 64-bit DES key is reduced to a 56-bit key by ignoring every eighth bit. These bits can be used as parity check to ensure the key is error free. After the 56-bit key is extracted, a different 48-bit subkey is generated for each of the 16 rounds of DES. These subkeys are generated by dividing the 56-bit key into two 28-bit halves. Then the haves are rotated left one or two bits depending on the round.

After being shifted, 48 out of the 56 bits are selected, because this operation permutes the order of the bits as well as selects a subset if bits, it is called a *compression permutation.*

The Expansion Permutation

This operation expands the right half of the data from 32 bits to 48 bits. Because this operation changes the operation of the bits as well as repeating certain bits, it is knows as an expansion permutation. This operation makes the right half the same size as the key for XOR operation and it provides a longer result that can be compressed during the substitution operation.

The S-Box Substitution

After the compressed key is Xored with the expanded block, the 48-bit result moves to a substitution operation. The substitutions are performed by eight substitution boxes or S-boxes. Each S-box has 6-bit input and a 4-bit output. The S-box substitution is the critical step in DES. The algorithm's other operations are linear and easy to analyze. The S-boxes are nonlinear and gives DES its security.

The result of this substitution phase is eight 4-bit blocks which are recombined into a single 32-bit block. This block moves to the next step, the P-Box permutation.

The P-Box Permutation

The 32-bit output of the S-box substitution is permuted according to a P-box. This permutation maps each input bit to an output position, where no bits are used twice and no bits are dropped. Finally the result of the P-box permutation is Xored with the left half of the initial 64-bit block, and then the left and right halves are switched and another round begins.

The Final Permutation

The final permutation is the inverse of the initial permutation, where exchanging the halves and shifting around the permutation yields exactly the same result. This is so that the algorithm can be used for both encryption and decryption.

4.4 Lightweight Block Ciphers

Lightweight block ciphers are suitable for constrained applications. For example devices with constrained resources such as RFID tags, wireless sensors network (WSNs) and sensor nodes often cannot support resource intensive cryptographic primitives such as SHA or AES or the public-key cryptosystem. To cater to the requirement of such resource constrained applications lightweight ciphers have been researched extensively in the recent years.

The fundamental principles and standards to design cryptosystems intended for such resource constrained applications are to some extent different from the design standards of the commonly used cryptographic systems for data and communication security.

For the design of a lightweight cryptosystem algorithm a designer is expected to maintain a balance of the three parameters: Security, cost (Gate equivalence) and performance.

Typically a basic RFID tag chip area needs 1,000 to 10,000 GE (Gate equivalent), therefore efficient, low cost and improved security lightweight ciphers have become an area of high demand for research and development.

Structure of these lightweight cipher as similar to traditional block ciphers can be classified into: substitution permutation networks and Feistel-type structure.

The criteria for lightweight block cipher is a balance between reliability, performance and cost.

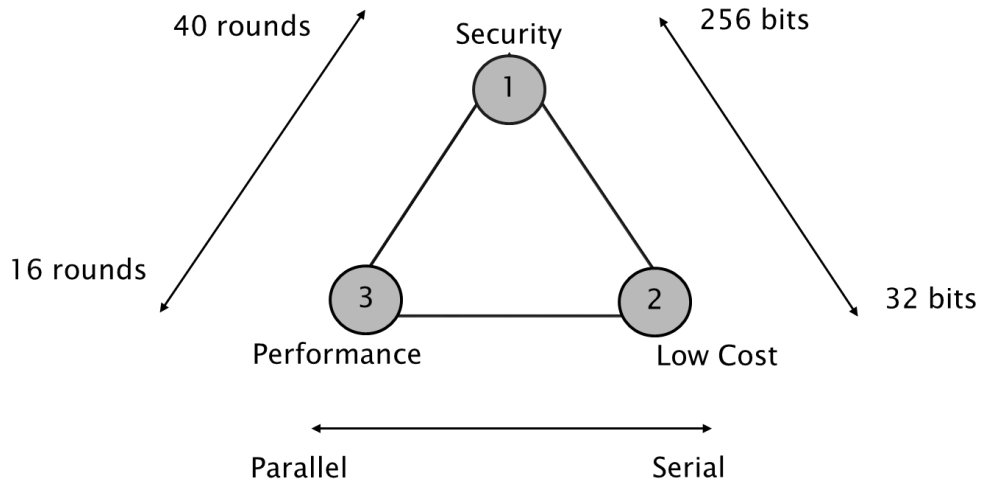


Figure 14. Interaction of reliability, performance and cost

The key size of a lightweight block cipher determines the ratio of the reliability and cost, the number of rounds of encryption provided ratio between reliability and performance and features of the hardware design are price and performance. Chip area is limited and power is also constrained.

5. Cryptanalysis

The whole point of cryptography is to keep the Plaintext (or the key, or both) secret from eavesdroppers (also called adversaries, attackers, interceptors, interlopers, intruders, opponents, or simply the enemy). Eavesdroppers are assumed to have complete access to the communications between the sender and receiver.

Cryptanalysis is the science of recovering the Plaintext of a message without access to the key. Successful cryptanalysis may recover the Plaintext or the key. It also may find weaknesses in a cryptosystem that eventually lead to the previous results.

An attempted cryptanalysis is called an *attack*. A fundamental assumption in cryptanalysis, first enunciated by the Dutchman A. Kerckhoffs in the nineteenth century, is that the secrecy must reside entirely in the key [21]. Kerckhoffs assumes that the cryptanalyst has complete details of the cryptographic algorithm and implementation. While real-world cryptanalysts don't always have such detailed information, it's a good assumption to make. If others can't break an algorithm, even with knowledge of how it works, then they certainly won't be able to break it without that knowledge.

There are many approaches to cipher security attack/cryptanalysis, some common are listed below:

- Differential cryptanalysis
- Linear cryptanalysis
- Extensions of differential cryptanalysis
  - Truncated, impossible, higher-order differential cryptanalysis, boomerang attacks
- Extensions of linear cryptanalysis
  - Multiple-approximation, zero-correlation linear cryptanalysis
- Differential-linear attacks
- Integral attacks
- Slide attacks
- Meet-in-the-middle attacks

The two most common and practised cryptanalysis approaches are:

5.1 Differential and Linear Cryptanalysis

Differential cryptanalysis and linear cryptanalysis are the two most important techniques in the analysis of symmetric-key cryptographic primitives. For block ciphers, differential cryptanalysis analyzes how input differences in the Plaintext propagates to output differences in the Ciphertext. Linear cryptanalysis studies probabilistic linear relations between Plaintext, Ciphertext and key. If a cipher behaves differently from a random cipher for differential or linear cryptanalysis, this can be used to build a distinguisher or even a key-recovery attack. For stream ciphers, differential cryptanalysis can be used in the context of a resynchronization attack. In one possible setting, the same data is encrypted several times with the same key, but using a different initial value. This is referred to as the standard (non-related-key) model, where the initial value is assumed to be under control of the attacker. An even stronger attack model is the related-key setting, where the same data is encrypted with different initial values and different keys. Not only the initial values, but also the differences between the keys are assumed to be under control of the attacker. Similar to differential cryptanalysis, linear cryptanalysis can also be used to attack stream ciphers in both the standard and related-key model. In the case of stream ciphers, linear cryptanalysis amounts to a known initial value attack instead of a chosen-initial value attack.

Resistance against linear and differential cryptanalysis is a standard design criterion for new ciphers [20]. For the block cipher AES, provable security against linear and differential cryptanalysis follows from the wide trail design strategy. In this work, we apply a similar strategy. After proving a reduced round attack probabilities on both differential and linear cryptanalysis, we use the basic linear and differential probability of the non-linear components to derive an upper bound for the probability of the best characteristic. We assume (as is commonly done) that the probability of the differential can accurately be estimated by the probability of the best characteristic and probability of linear can be best estimated by the probability of the closest linearized relation.

5.2 Differential cryptanalysis

Differential cryptanalysis looks specifically at Ciphertext pairs whose Plaintexts have particular differences. It analyzes the evolution of these differences as the Plaintexts propagate through the rounds of algorithm when they are encrypted with the same key.

The two Plaintext can be chosen at random, as long as they satisfy particular difference conditions, then using the differences in the resulting Ciphertext, assign different probabilities

to different keys. As more and more Ciphertext pairs are analyzed, one key will emerge as the most probable. This is the correct key.

Figure 15 below shows a DES round function, let's assume a pair of inputs X and X' that have the difference ΔX. The output Y and Y' are known and their difference ΔY is also known. Both the expansion permutation and the P-box are known. So ΔA and ΔC are known. B and B' are not known but their difference ΔB is known and equal to ΔA (when looking at the difference, the Xoring of K_i with A and A' cancels out). For any given ΔA, not all values of ΔC are equally likely. The combination of ΔA and ΔC suggest values for bits of A Xor K_i and A' Xor K_i. Since A and A' are known, this gives us information about K_i .

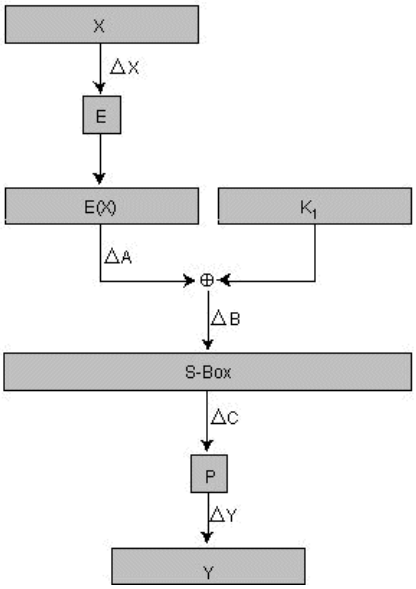


Figure 15. Differential propagation in DES

Certain differences in the Plaintext pairs have a high probability of causing certain differences in the resulting Ciphertext pairs. These are called *characteristics*. We can find these characteristics by generating a table where rows represent the possible Xors of two different sets of input bits, the column represent the possible output Xors, and the entries represent the number of times a particular output Xor occurs for a given input Xor. We can generate such table for each of DES's eight S-boxes.

To find the correct round key, simply collect enough guesses so that one subkey is suggested more often than all other. In effect the correct subkey will rise out of all the random alternatives.

The basic differential attack on *n*-round DES will recover the 48-bit subkey used in round *n,* and the remaining 8 key bits are obtained by brute-force guessing.

There are considerable problems with this attack, until you accumulate sufficient data you can't tell the correct subkey from all the noise, and the attack isn't practical, you have to use counters to assign different probabilities to 2^{48} possible subkeys.

Some researchers improved this attack by using a 13-round characteristic instead of 15-round characteristic on a 16-round DES. A shorter characteristic with a higher probability worked better.

The best attack against full 16-round DES requires 2^{47} chosen Plaintexts. This can be converted to known Plaintext attack, but that required 2^{55} known Plaintexts and 2^{37} DES operations are required during analysis.

Differential cryptanalysis works against DES and other similar algorithms with constant S-boxes. The attack is heavily dependent on the structure of the S-boxes.

DES's resistance can be improved by increasing the number of rounds. At 19 rounds or more differential cryptanalysis becomes impossible because it requires more than 2^{64} chosen Plaintext, since DES has a 64-bit block size, so it only has 2^{64} possible Plaintext blocks.

Here are some facts about differential attack. First, this attack is largely theoretical. The enormous time and data requirement to mount differential cryptanalytic attack makes it almost impossible to implement. To get the requisite data for this attack against a full DES, you have to encrypt a 1.5 megabits per second data stream of chosen plaintext for almost three years.

Second this is a chosen Plaintext attack, which can be converted to a known-Plaintext attack, but you have to go through all of the Plaintext-Ciphertext pairs looking for the useful ones.

5.3 Related Key Cryptanalysis

is similar to differential cryptanalysis, but it examines the difference between keys. The cryptanalyst chooses a relationship between a pair of keys, but does not know the keys themselves.

5.4 Linear Cryptanalysis

Linear cryptanalysis is another type of cryptanalytic attack, this attack uses linear approximations to describe the action of block cipher.

This means that if some of the Plaintext bits are Xored together, and then some of the Ciphertext bits are Xored together and then the results are Xored together, we will get a single bit that is the Xor of some of the key bits. This is a linear approximation and will hold some probability p. If $p \neq ½$, then this bias can be exploited. Use collected Plaintexts and associated Ciphertexts to guess the values of the key bits. The more data is available, the more reliable the guess. A good linear approximation for DES can be identified by finding a good 1-round linear approximations and joining them together. At the S-boxes, there are 6 input bits and 4 output bits. The input bits can be combined using Xor in 63 useful ways (2^6-1), and the output bits can be combined in 15 useful ways. Now for each S-box you can evaluate the probability that for a randomly chosen input, an input Xor combination equals some output Xor combination, if there is a combination with a high enough bias, then linear cryptanalysis may work.

There are two stages to applying linear cryptanalysis to a block cipher: (1) finding suitable linear approximations of the cipher, and (2) applying the known-Plaintext attack algorithm. An overview of how suitable linear approximations for a Feistel cipher may be found is as follows:

Step 1 Find linear equations which are good approximations to the nonlinear part of the cipher function. Take note of the probability with which the linear approximation holds.
Step 2 Extend the linear approximations to the round function, and thus formulate a linear equation for each approximation.
Step 3 Construct a linear approximation of the block cipher by compounding linear equations for the round function, making sure all intermediate unknown message terms are cancelled out. The probability of this cipher linear approximation can be calculated from the probabilities of the round approximations.
Step 4 Calculate the number of known Plaintexts required for the known Plaintext attack. This depends on the probability of the cipher approximation as well as the required degree of success.
If the linear approximations are unbiased, then they would hold for 32 of the 64 possible inputs, in case of DES the most biased S-box is S-box 5, the second input bit is equal to the XOR of all 4 output bits for only 12 inputs. This translates to a probability of 3/16 or a bias of 5/16 and is the most extreme bias in all the S-boxes.

Figure 16 below shows how to turn this into an attack against the DES round function.

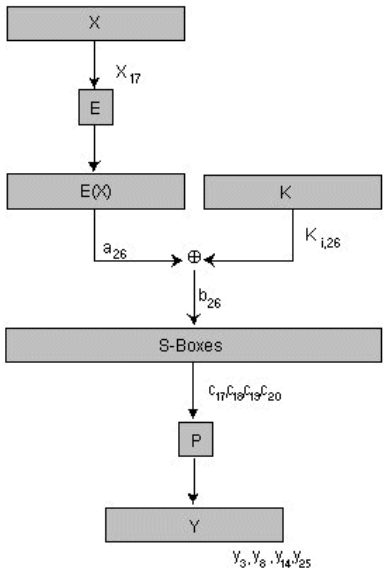


Figure 16. Linear attack on DES round

The input bit into S-box is b_{26}. The 4 output bits from S-box 5 are c_{17}, c_{18}, c_{19} and c_{20}. We can trace b_{26} backwards from the input to the S-box. The bit a_{26} is Xored with a bit from the subkey $K_{i,26}$ to obtain b_{26}. And bit X_{17} goes through the expansion permutation to become a_{26}. After the S-box, the 4 output bits go through the P-box to become 4 output bits of the round function Y3, Y8, Y14 and Y25. This means that with probability 1/2-5/16.

Linear approximations for different rounds can be joined in a manner similar to differential cryptanalysis. The basic attack is to use the best linear approximation for 16-round DES. It requires 2^{47} known Plaintext blocks and will result in 1 key bit, which is not very useful. If we interchange role of Plaintext and Ciphertext and use decryption as well as encryption, we can get 2 key bit, which is still not very useful.

This can be tweaked by using a 14-round linear approximation for rounds 2 through 15. Guess the 6 subkey bits relevant to S-box 5 for the first and last rounds (12 key bits in all). Effectively you are doing 2^{12} linear cryptanalyses in parallel and picking the correct one based on probabilities. This recovers the 12 bits plus the b_{26} and reversing Plaintext and Ciphertext another 13 bits, to get remaining 30 bits, use exhaustive search.

5.5 Cryptanalytic Attacks

An attempted cryptanalysis is called an attack. There are four general types of cryptanalytic attacks, each of them assumes that the cryptanalyst has complete knowledge of the encryption used:

1. **Ciphertext-only attack.** The cryptanalyst has the Ciphertext of several messages, which have been encrypted using the same encryption algorithm. The attacker's job is to recover the Plaintext from the Ciphertext, or better yet to obtain the key(s) used for encryption, in order to decrypt other messages encrypted with the same keys.

Given: $C_1 = E_k(P_1), C_2 = E_k(P_2), \ldots\ldots C_n = E_k(P_n)$

Deduce: Either $P_1, P_2, \ldots P_n$; k; or an algorithm to deduce P_{n+1} from $C_{n+1} = E_k(P_{n+1})$

2. **Known-Plaintext attack.** The Cryptanalyst has the access not only to the Ciphertext of several messages but also has Plaintext of those messages. Attacker's job is to find the key(s) used for encryption or an algorithm to decrypt any new messages encrypted from the same keys(s).

Given: $P_1, C_1 = E_k(P_1), C_2 = E_k(P_2), \ldots\ldots C_n = E_k(P_n)$

Deduce: Either k, or an algorithm to deduce P_{n+1} from $C_{n+1} = E_k(P_{n+1})$

3. **Chosen-Plaintext attack.** The cryptanalyst not only has access to the Ciphertext, associated Plaintext for several messages, but he also chooses the Plaintext that gets encrypted. This is more powerful than a known-Plaintext attack, because the attacker can choose specific Plaintext blocks to encrypt to yield more information about the key(s). Attacker's job is to deduce the key(s) used to encrypt the messages or an algorithm to decrypt any new messages encrypted with the same key(s).

Given: $P_1, C_1 = E_k(P_1), P_2, C_2 = E_k(P_2), \ldots\ldots C_n = E_k(P_n)$, *where the cryptanalyst gets to choose* $P_1, P_2, \ldots P_n$

Deduce: Either k, or an algorithm to deduce P_{n+1} from $C_{n+1} = E_k(P_{n+1})$

4. **Adaptive-chosen-Plaintext attack.** This is a special case of a chosen-Plaintext attack. Cryptanalyst not only gets to choose the Plaintext that is encrypted, but he can also modify his choice based on results of previous encryption. In a chosen-plaintext attack a cryptanalyst might just be able to choose one large block of Plaintext to be encrypted, whereas in an adaptive-chosen-Plaintext attack he can choose smaller block of Plaintext and then choose another based on the results of the previous and so forth.

The other three types of cryptanalytic attack are:

5. **Chosen-Ciphertext attack.** The Cryptanalyst can choose different Ciphertexts to be decrypted and gets access to the Plaintext. For examples the cryptanalyst has access to a tamper-poof box that does automatic decryption, his job is to deduce the key.

Given: C_1, $P_1 = D_k(C_1)$, C_2 , $P_2 = D_k(C_2), \ldots\ldots C_n$,$P_n = D_k(C_n)$

Deduce: k

This attack is primarily targeted to public-key algorithms but it is sometimes effective against symmetric algorithms as well.

6. **Chosen-key attack.** Cryptanalyst has some knowledge about the relationship between different keys, it does not mean that he can choose the key.

7. **Rubber-hose cryptanalysis**. In this attack the cryptanalyst threatens or blackmails or forcefully obtain the key from someone.

8. **Purchase-key attack**. Bribery is sometimes referred to as a purchase-key attack.

6. Cipher Inflex

6.1 The Block Cipher INFLEX

INFLEX block cipher presents, security analysis in terms of time complexity of solving encryption without knowledge of the key to the encryption algorithm which consists of modulo addition and unique tweakable feature on input block expansion [15], as well as hardware simulation using VHDL and cyclone iv family FPGA. INFLEX architecture is based on a Feistel network and consists of 20 rounds. Block size is 2n (32) bits and key supported is 64 bits. INFLEX 32/64 can be denoted as *2n/nl* (*n=16,l=4*) Round functions consists of:

- Inflate function $\mathcal{I}_f$
- Rotation $\ggg \lll$
- Modular addition and subtraction $\boxplus$ & $\boxminus$
- Deflate function $\mathfrak{D}_f$

INFLEX gets its fixed nonlinearity from modular addition operation, and scalable nonlinearity level from user selected string for the inflation function. The round function for INFLEX takes as an input a *64-bit* round key, the user selected 2^{m+1}*-bit* control string, together with 32-bits Plaintext word. The round function is the Feistel based map as under:

$$R_k(x,y) = ((\, s^{-\propto}\mathcal{I}^{2^m}x \boxplus \mathcal{I}^{2^m}y)\oplus k, s^{\beta}\mathcal{I}^{2^m}y \oplus (\, s^{-\propto}\mathcal{I}^{2^m}x \boxplus \mathcal{I}^{2^m}y)\oplus k) \text{--------(1)}$$

Where x and y are two 16 bit words.

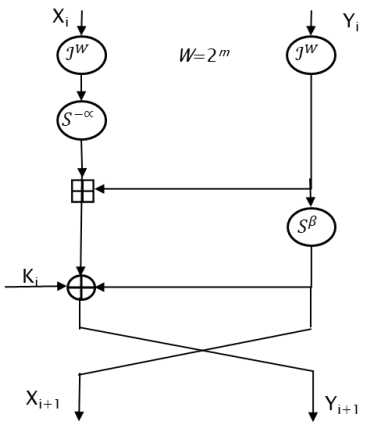


Figure 17. One round of INFLEX

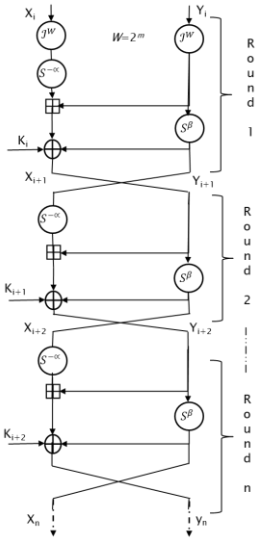


Figure 18. n rounds of INFLEX

INFLEX round function can be seen as combination of two Feistel-network maps with regards to two different types of additions as:

$(x, y) \longrightarrow (\mathcal{J}^{2^m} y, \big(S^{-\alpha} \mathcal{J}^{2^m} x \boxplus \mathcal{J}^{2^m} y \big) \oplus k)$; ---------- 2(a) and

$(x, y) \longrightarrow \mathcal{J}^{2^m} y, \big(S^{\beta} \mathcal{J}^{2^m} x \oplus \mathcal{J}^{2^m} y \big)$ ----------2(b)

The decomposition of the INFLEX round function can be seen in Figure 19

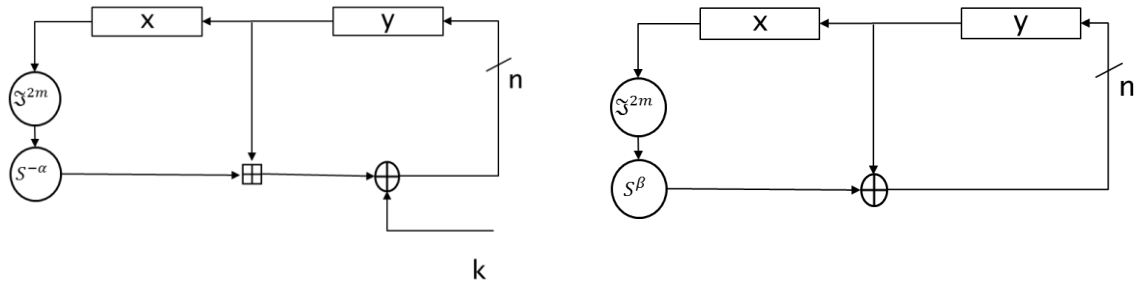


Figure 19. Decomposition of Round Function

Block size 2*n*	Key size	Word size *n*	Inflation *m*	KI String 2^{m+1}	Rotation α	Rotation β	Rounds T
32	64	16	0-4	4-32	7	2	20

Table 3. INFLEX specifications

6.2 Inflate Function $\mathcal{J}_f$

The inflate function expands each Plaintext single bit into a 2^m-bit string based on an 2^{m+1} bit control string *KI [15]*. The user selected parameter *m* determines extent of Plaintext expansion. The expansion routine runs n-bit/2^m times to expand the complete block. For a *16*-bit block size and m=2, the Plaintext block is broken down into 4 sub-blocks of 4 bits, each bit is expanded to 2^m=4 bits and the complete 16 bit block is expanded to a *64*-bit block. Expansion of the bit position is randomly selected through a 2^{m+1} length pseudorandom sequence generated by the linear feedback shift register (*LFSR*). The proposed expansion function is an arithmetic relationship that is easily scalable, the expanded block is 0-1 balanced and hence does not add any bias for cryptanalysis. The inflate function $\mathcal{J}_f$ can be represented as follows: for a given *n-bit* word input $X = \{x_k\}_{k=0}^{n} = \{x_{n-1}, \dots, x_1, x_0\}$ in $\mathbb{Z}$ with the control string input KI_X defined as $KI_X = \{KI_{x_{n-1}}, KI_{x_{n-2}}, \dots, KI_{x_1}, KI_{x_0} | KI_{x_k} \in \{0,1\}^m, 0 \leq k \leq n-1\}$. Each

input bit expansion can be represented as: $KI_{x_k,j} \triangleq \{KI_{x_k,2^m-1}, \ldots\ldots, KI_{x_k,1}, KI_{x_k,0}\}\ \forall\ KI_{x_k,j} \in \{0,1\}$ with $0 \leq k \leq n-1$, $0 \leq j \leq 2^m - 1$.

Let $\dot{X}$ be inflated input where $\dot{X} = \{\dot{x}_{k,j}\}_{k,j=0}^{k=n-1,\ j=2^m-1} = \{\dot{x}_{k,2^m-1}, \ldots\ldots, \dot{x}_{k,1}, \dot{x}_{k,0}\}$ and $\dot{x}_{k,j} \in \{0,1\}^{n-1}$ and KI_{x_k} are considered as decimal number in equation below:

$\dot{x}_k = \mathcal{I}_f(x_k, KI_{x_k})$

Where, $\mathcal{I}_f(x_k, KI_{x_k}) = \begin{cases} 2^{2^m} - 1 - 2^{KI_{x_k}}, & if\ x_k = 0 \\ 2^{KI_{x_k}}, & if\ x_k = 1 \end{cases}$

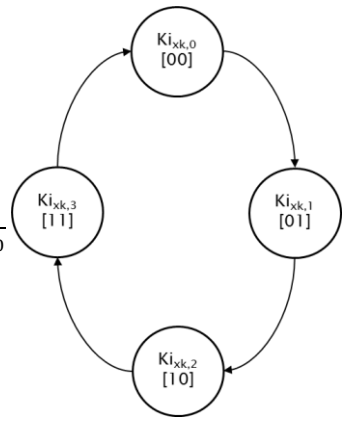
For n=4 and m=2 we have

$KI_{x_k} = \{KI_{x_3,1}, KI_{x_{3,0}}, KI_{x_2,1}, KI_{x_{2,0}}, KI_{x_1,1},$

$KI_{x_{1,0}}, KI_{x_0,1}, KI_{x_{0,0}}\}$

$(x_k = 0) \rightarrow x_{k,3}, x_{k,2}x_{k,1}, \overline{x_{k,0}}$

$(x_k = 1) \rightarrow \overline{x_{k,3}}, \overline{x_{k,2}}, \overline{x_{k,1}}, x_{k,0}$



$(x_k = 0) \rightarrow \overline{x_{k,3}}, x_{k,2}, x_{k,1}x_{k,0}$

$(x_k = 1) \rightarrow x_{k,3}, \overline{x_{k,2}}, \overline{x_{k,1}}, \overline{x_{k,0}}$

$(x_k = 0) \rightarrow x_{k,3}, \overline{x_{k,2}}, x_{k,1}x_{k,0}$

$(x_k = 1) \rightarrow \overline{x_{k,3}}, x_{k,2}, \overline{x_{k,1}}, \overline{x_{k,0}}$

$(x_k = 0) \rightarrow x_{k,3}, x_{k,2}, \overline{x_{k,1}}, x_{k,0}$

$(x_k = 1) \rightarrow \overline{x_{k,3}}, \overline{x_{k,2}}, x_{k,1}, \overline{x_{k,0}}$

Figure 20. Inflate algorithm for m=2, expands each bit by 2^m

6.3 Bitwise Rotation

Bit rotations of n-bit words by a fixed rotation amount r, are not by themselves very useful cryptographic transformations, because the groups are so small subgroups of the group of bijections over n-bit words which has $2^n!$ Elements. However, bit rotations are useful in

cryptography when combined with other common operations on n-bit words, such as other bitwise operations (And, Xor etc.) and addition modulo 2^n. Rotation combined with other operations allows to build arbitrary transformations of n-bit words. For example, by combining a sufficient number of rotations, and some fixed constants, any function over n-bit words can be constructed (thus including any bijection, thus any bit permutation). By contrast, when n>1, this cannot be achieved using any combination of AND, OR, XOR, NAND, NOR, and addition modulo 2^n (because none of these operations can make bit-1 on input influence bit-0 on output). Therefore, rotation is one of few ways to achieve diffusion from high to lower-order bits.

Bitwise shifts and especially rotations are widely used in cipher design because they provide good diffusion. INFLEX algorithm applies a bitwise left rotation $\ggg$ to the left half of message by $\alpha = 7$ bits and bitwise right rotation $\lll$ to the right half of message by $\beta = 2$ bits. This introduces the primary diffusion independent of the diffusion introduced by the inflation of input block. This ensure presence of diffusion even if there is no inflation to the input block.

6.4 Modular Addition

The n-bit Modular Addition of $Z = X \boxplus Y \ mod\ 2^n : (x_k, y_k) \mapsto z_k$ can be expressed as [15]

$z_k = x_k + y_k \quad if\ k = 0$

$z_k = x_k + y_k + c_{k-1} \quad if\ k = 1, carry\ bit\ c_k = x_k y_k$

$\vdots$

In general;

$$z_{k-1} = x_{k-1} + y_{k-1} + c_{k-2} \quad if\ 2 \leq k \leq n-1 \qquad 3(a)$$

$$\text{Where } c_k = x_k y_k + (x_k + y_k)(x_{k-1} y_{k-1}) + \sum_{q=0}^{k-2} x_q y_q \prod_{r=q+1}^{k} (x_r + y_r) \qquad 3(b)$$

The modular addition is denoted by $\boxplus$ sign in $\mathbb{GF}_2$ which is actually the logic Exclusive-OR operation. The carry bit can be described by equation 3(b). By analysing equation 3(a) we can conclude that Modular Addition is not completely nonlinear since the least significant bit (*lsb*) of the resultant always stays linear. The carry term dominate the algebraic degree, as the algebraic degree increases linearly with carry terms in 3(b).

The Modular Addition in the round function perform integer addition modulo 2^n is denoted by $x \boxplus y$ where $(x, y \in \mathbb{Z}_{2^n})$, however inflated inputs as a result of previous inflate operations are

$\grave{X} = \{\grave{x}_{n-1}, \dots, \grave{x}_1, \grave{x}_0\} \triangleq \{\grave{x}_{(n-1)(2^m-1)}, \dots . \grave{x}_{(n-1)(1)}, \grave{x}_{(n-1)(0)}, \dots ,$

$\grave{x}_{(1)(2^m-1)}, \dots, \grave{x}_{(1)(1)}, \grave{x}_{(1)(0)}, \dots, \grave{x}_{(0)(2^m-1)}, \dots , \grave{x}_{(0)(1)}, \grave{x}_{(0)(0)}\}$

And

$\grave{Y} = \{\grave{y}_{n-1}, \dots, \grave{y}_1, \grave{y}_0\}$

$\triangleq \{\grave{y}_{(n-1)(2^m-1)}, \dots . \grave{y}_{(n-1)(1)}, \grave{y}_{(n-1)(0)}, \dots , \grave{y}_{(1)(2^m-1)}, \dots, \grave{y}_{(1)(1)}, \grave{y}_{(1)(0)}, \dots, \grave{y}_{(0)(2^m-1)}$

$, \dots, \grave{y}_{(0)(1)}, \grave{y}_{(0)(0)}\}$

As a result of input block inflation the modulo addition has increased from 2^n to 2^{n*2^m} for some user selected $m \in \mathbb{Z}^*$. The output of the modular addition is given by

$\grave{Z} = \{\grave{z}_{n-1}, \dots, \grave{z}_1, \grave{z}_0\}$

$\triangleq \{\grave{z}_{(n-1)(2^m-1)}, \dots . \grave{z}_{(n-1)(1)}, \grave{z}_{(n-1)(0)}, \dots , \grave{z}_{(1)(2^m-1)}, \dots, \grave{z}_{(1)(1)}, \grave{z}_{(1)(0)}, \dots, \grave{z}_{(0)(2^m-1)},$

$\cdots, \grave{z}_{(0)(1)}, \grave{z}_{(0)(0)}\}$

We can drive equation 4 using equation 3(a) and 3(b)

$$\grave{z}_{kj} = \begin{cases} \grave{x}_{kj} + \grave{y}_{kj} & if\ k = 0,\ j = 0; \\ \grave{x}_{kj} + \grave{y}_{kj} + \grave{x}_{kj-1}\grave{y}_{kj-1} + (\grave{x}_{kj-1} + \grave{y}_{kj-1})(\grave{x}_{kj-1} + \grave{y}_{kj-1} + \grave{z}_{kj-1}) & for 0 < k \leq n-1; \\ & 0 < j \leq 2^m - 1 \end{cases} \quad (4)$$

6.5 Deflate function $\mathfrak{D}_f$

This function completes the last operation of the proposed cipher design. It is a compaction function that extracts original bits of block from the inflated bits block $\{\grave{z}_k \mapsto z_k\ \{0,1\}^{n2^m} \mapsto \{0,1\}^n\ \forall\ n, 2^m\ \in\ \mathbb{Z}^*\}$ based on an *n*m*-bit control string KO. Let $= \{KO_{n-1}, \dots\dots, KO_1, K_0 | KO_k\ \in \{0,1\}^m, 0 < k \leq n-1\}$. Thereby we have $Z = \{z_{n-1}, \dots\dots, z_1, z_0\} = \{\mathfrak{D}_f(\grave{z}_{n-1}, KO_{n-1}), \dots, \mathfrak{D}_f(\grave{z}_1, KO_1), \mathfrak{D}_f(\grave{z}_0, KO_0)\}$. Therefore expression for $\mathfrak{D}_f$ can be generalized as:

$z_k = \mathfrak{D}_f(\grave{z}_k, KO_k)$

Where, $\mathfrak{D}_f(\grave{z}_k, KO_k) = \{\sum_{p=0}^{2^m-1} \grave{z}_{kp}\ \prod_{k=0}^{m-1}(-1)^{\frac{p}{2^k}+1} KO_{pk}$

In the equation above, (-1) refers to the complement of KO_{pk}, summation refers to logic Exclusive OR operation, product refers logic AND operation.

For m = 2 bits and n =4 bits

$$z_k = \sum_{p=0}^{4-1} \dot{z}_{kp} \prod_{k=0}^{2-1} (-1)^{\frac{p}{2^k}+1} KO_{pk}$$

$$= \dot{z}_{k0}(-1)^1 KO_{k0}(-1)^1 KO_{k1} + \dot{z}_{k1}(-1)^2 KO_{k0}(-1)^1 KO_{k1} + \dot{z}_{k2}(-1)^3 KO_{k0}(-1)^2 KO_{k1} + \dot{z}_{k3}(-1)^4 KO_{k0}(-1)^2 KO_{k1}$$

$$Z_k = \dot{z}_{k0}\overline{KO_{k0}KO_{k1}} + \dot{z}_{k1}KO_{k0}\overline{KO_{k1}} + \dot{z}_{k2}\overline{KO_{k0}}KO_{k1} + \dot{z}_{k3}KO_{k0}KO_{k1}$$

6.6 Key Schedule

Each algorithm also requires a key schedule to turn a key into a sequence of round keys. The INFLEX key schedule use the round function to expand the initial l-word master key to generate the round key k_i. For a 64-bit key where $l = 4$. We can write (l_2, l_1, l_0, k_0) , where master key $l_i, k_0 \in GF(2)^n$. Sequences k_i and l_i are defined by:

$l_{i+3} = (k_i + S^{-\alpha} l_i) \oplus i$ and

$k_{i+1} = S^{\beta} \oplus l_{i+3}$

The value k_i is the i^{th} round key, for $0 \leq i < T$. The round counter i here which serves to eliminate slide properties.

6.7 Pseudo code for INFLEX key expansion and encryption

$(\alpha, \beta) = (7,2)$

$x, y = plaintext\ words$

$l[2], l[1], l[0], k[0] = key\ words$

---------- $key\ expansion$ ----------

$for\ i = 0\ to\ 18$

$l[i+3] \leftarrow (k[i] + S^{-\alpha} l[i]) \oplus i$

$k[i+1] \leftarrow S^{\beta} k[i] \oplus l[i+3]$

$end\ for$

---------- $encryption$ ----------

$for\ i = 0\ to\ 19$

$x \leftarrow (S^{-\alpha} \jmath^{2^m} x + \jmath^{2^m} y) \oplus k[i]$

$y \leftarrow S^{-\alpha} \jmath^{2^m} y \oplus x$

$end\ for$

7. Hardware implementation

7.1 Implementation Platform

We implemented INFLEX in VHDL using Altera Modelsim for simulation and Quartus prime compiler for design and synthesis for the selected device of cyclone iv family and typical operating condition of 1.2 Volts for the core voltage and 25 ^{o}C operating temperature.

We simulated encryption algorithm for 5 scalable complexity levels based on user selected KI string of 0 to 4 bits.

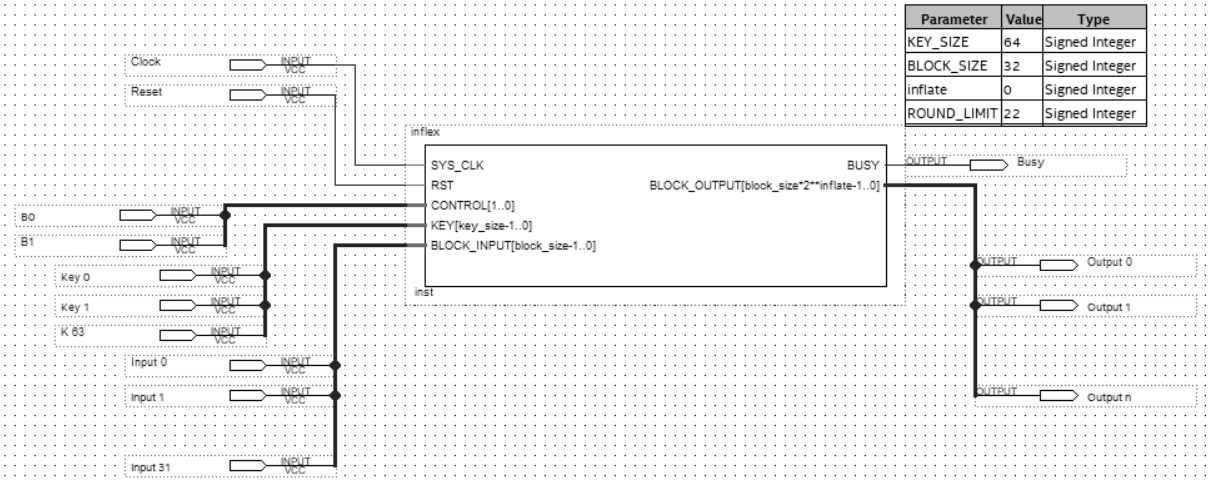


Figure 21. Block symbol diagram

7.2 INFLEX Inputs

INFLEX hardware has been designed to accept 32-bits input as a block of Plaintext and a 64-bits master key for generating the key schedule for all rounds of encryption. 32-bit block of data is split into two halves of 16 bits each, which goes through the round function of expansion modular addition and bitwise shift as depicted in Figure 17 above. Output of one round becomes input of the next round and undergoes same round function 20 times in 20 round, inducing diffusion and confusion round by round to generate Ciphertext output at the end of the 20th round. Clock, reset and control bit for INFLEX mode of key generation or encryption are also taken as input. Figure 22. Below shows simulation results of 32-bit Plaintext input with a 64-bit key input and 32-bit Cipher text output for an inflation factor m=0.

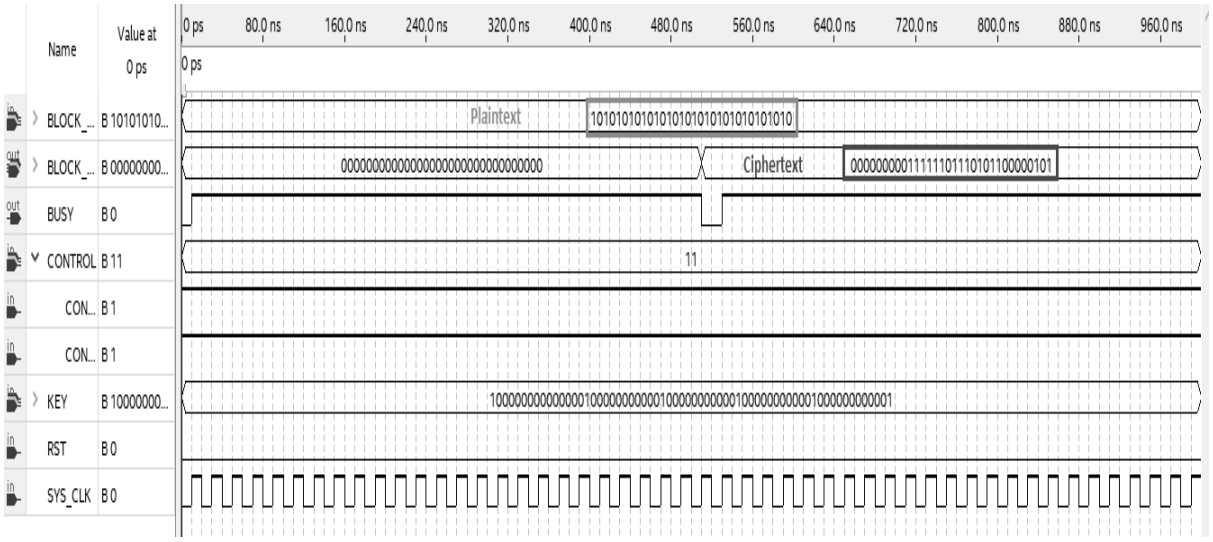


Figure 22. Encryption input/output of INFLEX

7.3 User requirement of inflation factor (m)

User requirement of inflation factor (m) which controls the time complexity level of solving the encryption algorithm of INFLEX without knowledge of key is taken at the hardware design stage and this factor dictates the output bits of the hardware. Figure 23 below shows an expanded output of 64-bit for a user selected inflation factor of m=1.

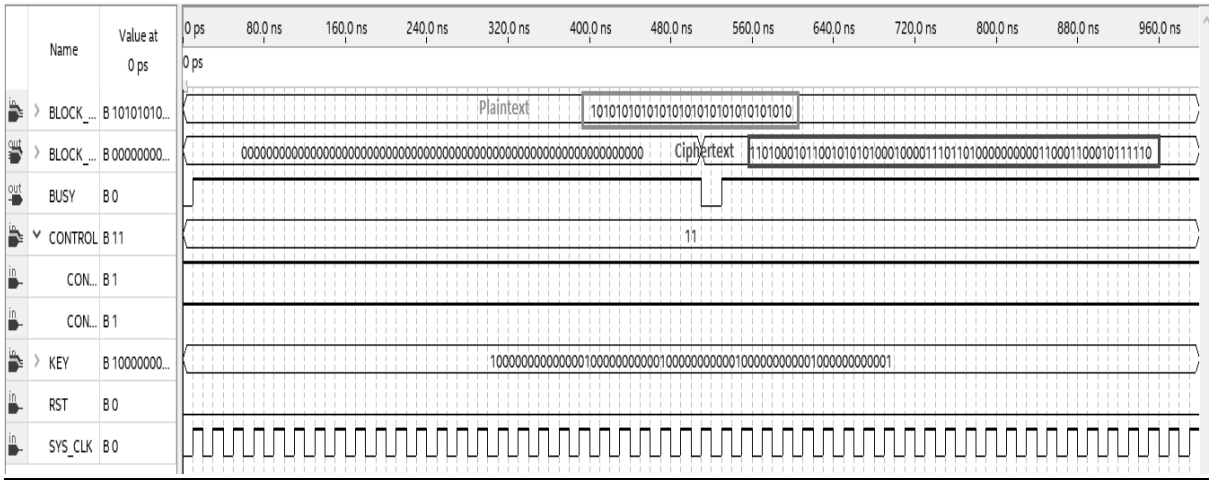


Figure 23. Encryption output with Inflation m=1

7.4 INFLEX Output

Size of the output bus depends on the user requirement of inflation factor (m). It is 32 bit for an inflation factor of zero (0) and it doubles to 64-bits for an inflation factor (m=1). This is fixed at the time of hardware design. Inflation factor (m) does not only dictates the times complexity of solving the encryption algorithm it also changes the hardware configuration and determines the fate equivalent of the hardware.

Table 4 below shows Gate Equivalent (G.E) for all selections of inflation factor (m). Resource utilization for all modules has been reported for all available inflation factor (m) values.

Module	**Gate Equivalent**	**Percentage**	**Inflation factor (m)**
Data state	268.8	31.1%	0
Counter	25.2	2.9%	
Control	8.4	1%	
Others	54.8	6.4%	
Key state	268.8	31.1%	
ARX	238	27.5%	
	864		
Data state	403	25.1%	1
Counter	29.4	1.84%	
Control	8.4	0.5%	
Others	185	11.5%	
Key state	537	33.4%	
ARX	445	27.67%	
	1607.8		
Data state	672	19.6%	2
Counter	33.6	0.98%	
Control	8.4	0.25%	
Others	449	13.1%	
Key state	1075.2	31.44%	
ARX	1181	34.54%	

	3419		
Data state	1209.6	16.9%	3
Counter	37.8	0.53%	
Control	8.4	0.12%	
Others	982	13.71%	
Key state	2150.4	30%	
ARX	2771	38.7%	
	7159		
Data state	2284.8	16.5%	4
Counter	42	0.3%	
Control	8.4	0.06%	
Others	2053.8	14.84%	
Key state	4300.8	31.1%	
ARX	5145	37.2%	
	13,834		

Table 4. Gate equivalent for different inflation scales

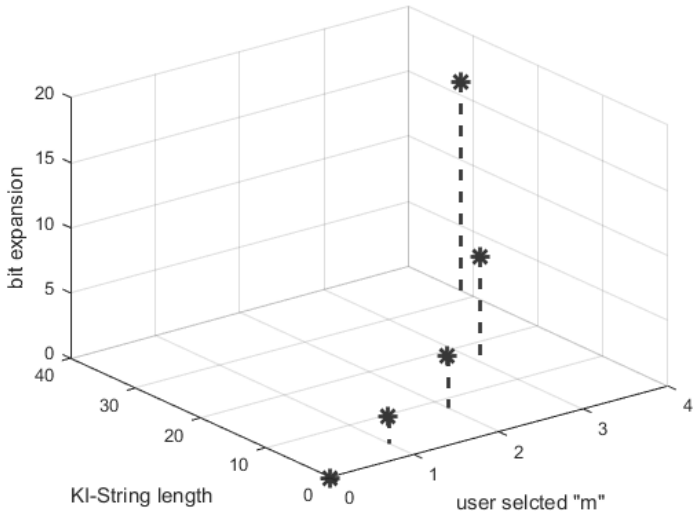


Figure 24. Bit expansion and KI string length for user selected inflate factor m

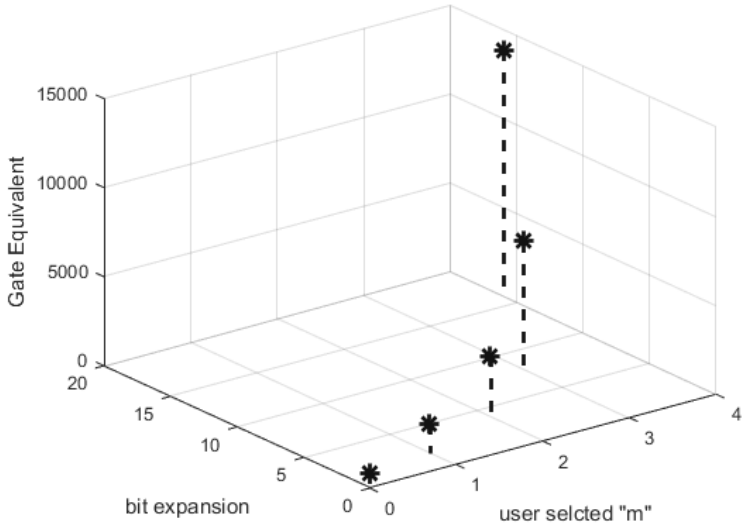


Figure 25. G.E and bit expansion for user selected inflate factor m

7.5 Data path for Plaintext to Ciphertext

Figure 26 below shows the data-path of an encryption-only INFLEX32/64, which performs one round in one clock cycle i.e. a 32-bit width data-path. The IFLEX hardware implementation is optimized and same hardware is used for multiple round of encryption algorithm.

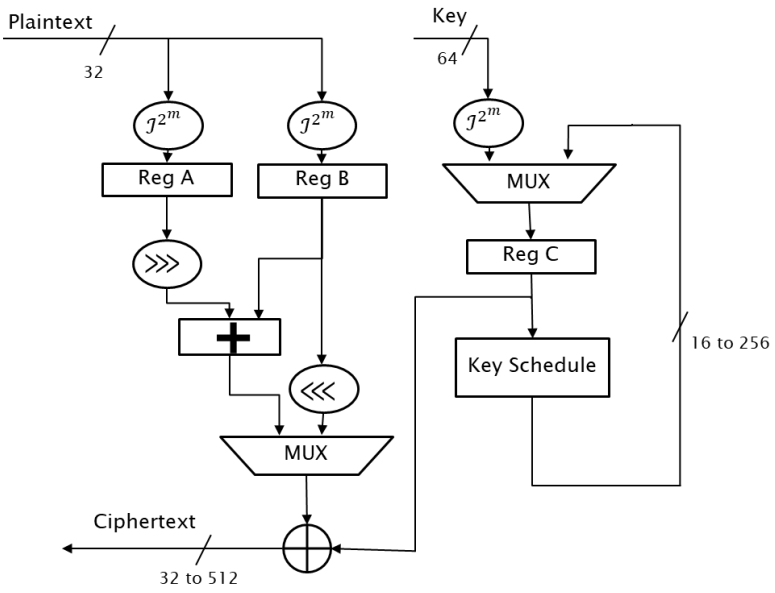


Figure 26. Data-path of INFLEX

7.6 Encrypt/Decrypt Symmetry

To enable compact joint implementations of the encryption and decryption algorithms, it's best to make encryption look like decryption. Most compact decryption can be accomplished by swapping Ciphertext words, reading round keys in reverse order, and then swapping the resulting Plaintext words.

INFLEX decryption requires deflate operation, modular subtraction, and the rotations are reversed, because its Feistel stepping performs all operations on one word.

Figure 27 below shows decryption by the same hardware where Ciphertext has now been taken as input and similar key is used to decrypt the Ciphertext back to the Plaintext.

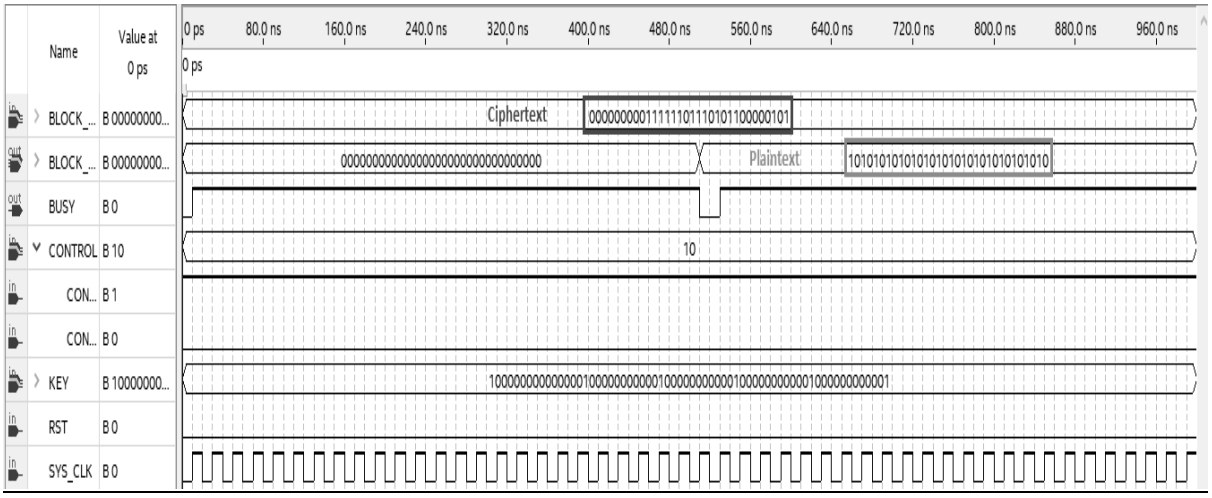


Figure 27. Decryption output of INFLEX

8. Performance Metrics

8.1 INFLEX Throughput

Our implementation requires 20 clock cycles to encrypt a 32-bit Plaintext with a 64-bit key. Ignoring the latency for data expansion and key schedule generation a throughput of 160kbps is achieved at 100kHz. Table 5 and Figure 28 below shows a comparison of INFLEX throughput with other state-of-art ciphers [5].

Cipher	**Key**	**Block**	**Cycles/block**	**Throughput (Kbps) at 100Khz**	**G.E**
INFLEX	64	32	20	160	864-13,834
Present	80	64	32	200	1570
AES	128	128	1032	12.4	3400
Hight	128	64	34	188.2	3048
mCrypton	96	64	13	492.3	2681
Camellia	128	128	20	640	11350
DES	56	64	144	44.4	2309
DESXL	184	64	144	44.4	2168

Table 5. INFLEX throughput and G.E comparison [5]

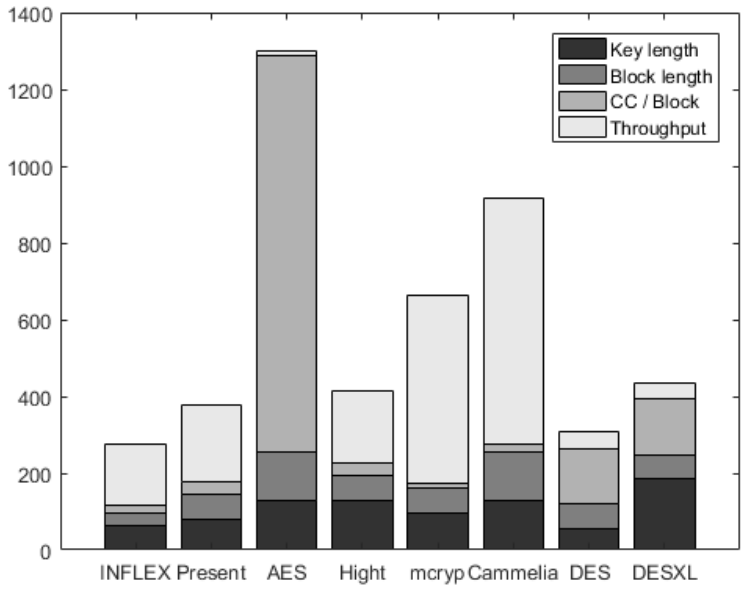


Figure 28. Comparison of throughput, and clock cycles per block [5]

8.2 Time complexity measure in terms of Linear and differential Cryptanalysis

INFLEX is comparable to Speck 32/64 variant of Simon and Speck family ciphers [13]. Since INFLEX is inspired by Speck and inherits round function with an improvement of bit expansion, we will utilize Speck cryptanalysis and add bit expansion complexity to demonstrate overall improvement in the algorithm complexity. Time complexity for attacking different round of INFLEX has been calculated both for differential and linear cryptanalysis and has been reported in Tables 8 & 9.

8.3 Linear Cryptanalysis

Linear cryptanalysis is a known-Plaintext attack which was introduced by Matsui as a theoretical attack on the Data Encryption Standard (DES) [16]. It exploits the correlation of linear approximations between input and output of a block cipher. Since INFLEX is an ARX cipher, which consists of bit expansion, modular addition, bit rotation and Xor operations, the basic step in establishing the linear approximation is to calculate the correlation of linear approximation for modulo addition.

We will first demonstrate linear approximation of Speck with only Modular Addition and then include nonlinearities of expansion function for the linear approximation of INFLEX. Linear approximation used here is a basic known-Plaintext attack, there have been improvements to this basic attack in an attempt to recover more bits of the key with less known plaintext, [17, 18] reported linear attack on 9 rounds with a correlation of 2^{-14}using linear mask tables.

Mounting Linear Attack on Speck

Nonlinearity in the round function of Speck is associated with the modular addition $\boxplus$ component only. The nonlinearity in the modular addition is due to the carry operation. Let,

$S = X \boxplus Y$ Which can be represented with the following equations:

$S[i] = X[i] \oplus Y[i] \oplus C[i]$;

$C[0] = 0$;

$C[i+1] = X[i]Y[i] \oplus X[i]C[i] \oplus Y[i]C[i]$.

Where S represents the sum and C is the carry, *i-th* bit of A is A[i] and the least significant bit (LSB) of X is X[0]. It can be noted that $S[0] = X[0] \oplus Y[0]$ being LSB is linear over $\mathbb{F}_2$. The non linearity in the modular addition only comes from the carry bit and this propagates upwards

only. The Speck encryption round function for the least-significant bit of the input block is linear:

$R_i(x_i, y_i) = ((\ x_i \boxplus y_i) \oplus k_i, y_i \oplus (x_i \boxplus y_i) \oplus k_i)$

$R_1(x_1, y_1) = ((\ x_1[0] \oplus y_1[0]) \oplus k_1[0], y_1[0] \oplus (x_1[0] \oplus\ y_1[0]) \oplus k_1[0])$

Similarly the decryption round function for the least-significant bit of the output block is also linear where the key is applied in reverse order:

$R_{22}(x_{22}, y_{22}) = ((\ x_{22}[0] \oplus y_{22}[0]) \oplus k_2[0], y_{22}[0] \oplus (x_{22}[0] \oplus\ y_{22}[0]) \oplus k_2[0])$

With these two linear equations of the LSB, we can recover two key bits (the LSB of K_1 and K_2) using known Plaintext by finding the linear expressions relating the LSBs of the Plaintext, Ciphertext and key.

Now in order to approximate the linearity of non-linear carry operation, we simulate a modular addition operation of X and Y.

X[i]	Y[i]	C[i]	C[i+1]
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 6. Modular addition and carry propagation

The Walsh transform reveals four linear approximations to the carry function, each holding with probability 6/8 = 0.750:

$C[i]$: $Prob = 6/8$,

$X[i]$: $Prob = 6/8$,

$Y[i]$: $Prob = 6/8$,

$X[i] \oplus Y[i] \oplus C[i] \oplus 1$: $Prob = 6/8$

This gives us two useful linear approximations for carry operation, $C[i] = x[i-1] and\ C[i] = y[i-1]$ both holding probability 6/8 or 0.75.

$x_2 = (x_1 \boxplus y_1) \oplus k$

$y_2 = x_2 \oplus y_1$

Above can be linearized as

$y_2[i] = x_1[i] \boxplus y_1[i] \oplus y_1[i] \oplus k_1[i]$

$y_2[i] = x_1[i] \oplus k_1[i-1]$

$k_1[i-1] = x_1[i] \oplus y_2[i]$

The above expression holds probability of 0.75 for 1 (i-1) bit of key for one round. We can use piling-up lemma described by Matsui [16] to obtain the probability of the cipher linear approximation from the probabilities of the one-round linear approximations. Equivalently, the probability of the cipher linear approximation may be calculated by which evaluates the probability that the exclusive-or of *n* independent random variables will equal zero.

Piling-up Lemma

$$P = \frac{1}{2} + 2^{n-1} \prod_{i=1}^{n} \left(pi - \frac{1}{2}\right)$$

For sixteen round attack the probability is $\frac{1}{2} + 2^{-17}$. According to lemma 2 of Matsui [16] $\left[p - \frac{1}{2}\right]^{-2}$ a total of $[0.500007629 - 0.5]^{-2}$=1.717x$10^{10} = 2^{34}$ known Plaintexts are required for a success rate of 97.7%. with an attack complexity of $2^{17}.2^{34} = 2^{51}$.

For fourteen round attack the probability is $\frac{1}{2} + 2^{-15}$. According to lemma 2 of Matsui [16] $\left[p - \frac{1}{2}\right]^{-2}$ a total of $[0.5000305 - 0.5]^{-2}$=1.0737x$10^{10} = 2^{30}$ known Plaintexts are required for a success rate of 97.7%. with an attack complexity of $2^{15}.2^{30} = 2^{45}$

For twelve round attack the probability is $\frac{1}{2} + 2^{-13}$. According to lemma 2 of Matsui [16] $\left[p - \frac{1}{2}\right]^{-2}$ a total of $[0.500122 - 0.5]^{-2}$=6.7108x$10^{8} = 2^{27}$ known Plaintexts are required for a success rate of 97.7%. with an attack complexity of $2^{13}.2^{27} = 2^{40}$

For ten round attack the probability is $\frac{1}{2} + 2^{-11}$. According to lemma 2 of Matsui [16] $\left[p - \frac{1}{2}\right]^{-2}$ a total of $[0.500488 - 0.5]^{-2}$=4.194x$10^{6} = 2^{22}$ known Plaintexts are required for a success rate of 97.7%. with an attack complexity of $2^{11}.2^{22} = 2^{33}$

For eight round attack the probability is $\frac{1}{2} + 2^{-9}$. According to lemma 2 of Matsui [16] $\left[p - \frac{1}{2}\right]^{-2}$ a total of $[0.50195 - 0.5]^{-2}$=2.62144x$10^5 = 2^{18}$ known Plaintexts are required for a success rate of 97.7%. with an attack complexity of $2^9 . 2^{18} = 2^{27}$

Extending Speck linear attack to INFLEX

INFLEX, in addition to modular addition nonlinear component has input inflation function, which has multiple hardware implementations, the simplest can be implemented through a lookup table with 2^{m+1} entries which randomly select the bit inflation pattern as per the Linear Feedback Shift Register generated sequence of 2^{m+1}. To expand *n*-bits to *2m*n*-bits. For m=4 the probability of selecting a pattern is 1/32 = 0.03125 and the probability of masking a bit in a 16 bit string is 1/16= 0.0625, which gives a combined probability of 1/0.00195. This probability of expansion will remain same for all round of cipher.

For m=1 the probability of selecting a pattern is 1/4 = 0.25 and the probability of masking a bit in a 2 bit string is 1/2= 0.5, which gives a combined probability of 1/0.125. This probability of expansion will remain same for all round of cipher.

With an inflation of m=4

By adding this probability to 16 round attack we get INFLEX linear attack probability for 16 rounds as 2^{-26} and a total of $[0.500000014 - 0.5]^{-2} = 4.5036x10^{15} = 2^{52}$ Plaintexts are required for a success rate of 97.7%.

For a 14 round attack we get INFLEX linear attack probability for as 2^{-24} and a total of $[0.500000059 - 0.5]^{-2} = 2.8147x10^{14} = 2^{48}$ Plaintexts are required for a success rate of 97.7%.

For a 12 round attack we get INFLEX linear attack probability for as 2^{-22} and a total of $[0.500000238 - 0.5]^{-2} = 1.7592x10^{13} = 2^{44}$ Plaintexts are required for a success rate of 97.7%.

For a 10 round attack we get INFLEX linear attack probability for as 2^{-20} and a total of $[0.500000953 - 0.5]^{-2} = 1.09951x10^{12} = 2^{40}$ Plaintexts are required for a success rate of 97.7%.

For an 8 round attack we get INFLEX linear attack probability for as 2^{-18} and a total of $[0.500003814 - 0.5]^{-2} = 6.8719x10^{10} = 2^{36}$ Plaintexts are required for a success rate of 97.7%.

For a 5 round attack we get INFLEX linear attack probability for as 2^{-15} and a total of $[0.500030517 - 0.5]^{-2} = 1.073x10^{9} = 2^{30}$ Plaintexts are required for a success rate of 97.7%.

With an inflation of m=3

By adding this probability to 16 round attack we get INFLEX linear attack probability for 16 rounds as 2^{-24} and a total of $[0.500000059 - 0.5]^{-2} = 2.815x10^{14} = 2^{48}$ Plaintexts are required for a success rate of 97.7%.

For a 14 round attack we get INFLEX linear attack probability for as 2^{-22} and a total of $[0.500000238 - 0.5]^{-2} = 1.7592x10^{13} = 2^{44}$ Plaintexts are required for a success rate of 97.7%.

For a 12 round attack we get INFLEX linear attack probability for as 2^{-20} and a total of $[0.500000953 - 0.5]^{-2} = 1.09951x10^{12} = 2^{40}$ Plaintexts are required for a success rate of 97.7%.

For a 10 round attack we get INFLEX linear attack probability for as 2^{-18} and a total of $[0.500003814 - 0.5]^{-2} = 6.8719x10^{10} = 2^{36}$ Plaintexts are required for a success rate of 97.7%.

For an 8 round attack we get INFLEX linear attack probability for as 2^{-16} and a total of $[0.500015258 - 0.5]^{-2} = 4.2954x10^{9} = 2^{32}$ Plaintexts are required for a success rate of 97.7%.

With an inflation of m=2

By adding this probability to 16 round attack we get INFLEX linear attack probability for 16 rounds as 2^{-22} and a total of $[0.5000000238 - 0.5]^{-2} = 1.7592x10^{13} = 2^{44}$ Plaintexts are required for a success rate of 97.7%.

For a 14 round attack we get INFLEX linear attack probability for as 2^{-20} and a total of $[0.500000953 - 0.5]^{-2} = 1.09951x10^{12} = 2^{40}$ Plaintexts are required for a success rate of 97.7%.

For a 12 round attack we get INFLEX linear attack probability for as 2^{-18} and a total of $[0.500003814 - 0.5]^{-2} = 6.8719x10^{10} = 2^{36}$ Plaintexts are required for a success rate of 97.7%.

For a 10 round attack we get INFLEX linear attack probability for as 2^{-16} and a total of $[0.500015258 - 0.5]^{-2} = 4.2954x10^{9} = 2^{32}$ Plaintexts are required for a success rate of 97.7%.

With an inflation of m=1

By adding this probability to 16 round attack we get INFLEX linear attack probability for 16 rounds as 2^{-20} and a total of $[0.500000953 - 0.5]^{-2} = 1.09951x10^{12} = 2^{40}$ Plaintexts are required for a success rate of 97.7%.

For a 14 round attack we get INFLEX linear attack probability for as 2^{-18} and a total of $[0.500003814 - 0.5]^{-2} = 6.8719x10^{10} = 2^{36}$ Plaintexts are required for a success rate of 97.7%.

For a 12 round attack we get INFLEX linear attack probability for as 2^{-16} and a total of $[0.500015258 - 0.5]^{-2} = 4.2949x10^{9} = 2^{32}$ Plaintexts are required for a success rate of 97.7%.

8.4 Linear Attack Complexity

With an inflation of m=4

For a 16 round attack, the complexity for encrypting 2^{52} linear relations pairs for each 2^{26} guesses of key bits and is $(2^{52}.2^{26}) \approx 2^{78}$.

For a 14 round attack, the complexity for encrypting 2^{48} linear relations pairs for each 2^{24} guesses of key bits and is $(2^{48}.2^{24}) \approx 2^{72}$.

For a 12 round attack, the complexity for encrypting 2^{44} linear relations pairs for each 2^{22} guesses of key bits and is $(2^{44}.2^{22}) \approx 2^{66}$.

For a 10 round attack, the complexity for encrypting 2^{40} linear relations pairs for each 2^{20} guesses of key bits and is $(2^{40}.2^{20}) \approx 2^{60}$.

For a 8 round attack, the complexity for encrypting 2^{36} linear relations pairs for each 2^{18} guesses of key bits and is $(2^{18}.2^{36}) \approx 2^{54}$.

For a 5 round attack, the complexity for encrypting 2^{30} linear relations pairs for each 2^{15} guesses of key bits and is $(2^{15}.2^{30}) \approx 2^{45}$.

Figure 29 compares time complexity of solving encryption algorithm of INFLEX at inflation factor m=4 with time complexity of solving encryption algorithm of Speck as calculated above.

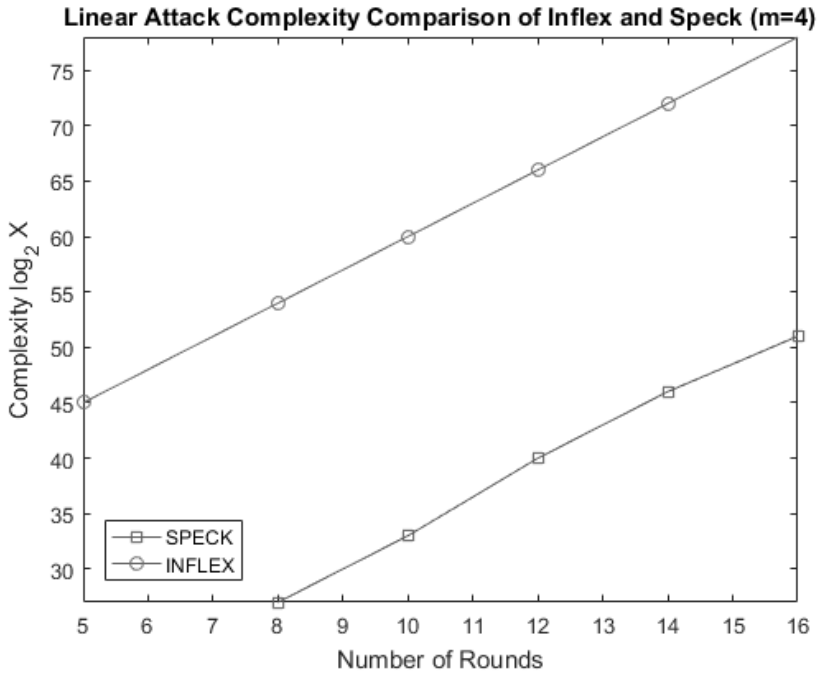


Figure 29. Time Complexity comparison with m=4 inflation

With an inflation of m=3

For a 16 round attack, the complexity for encrypting 2^{48} linear relations pairs for each 2^{24} guesses of key bits and is $(2^{48}.2^{24}) \approx 2^{72}$.

For a 14 round attack, the complexity for encrypting 2^{44} linear relations pairs for each 2^{22} guesses of key bits and is $(2^{44}.2^{22}) \approx 2^{66}$.

For a 12 round attack, the complexity for encrypting 2^{40} linear relations pairs for each 2^{20} guesses of key bits and is $(2^{40}.2^{20}) \approx 2^{60}$.

For a 10 round attack, the complexity for encrypting 2^{36} linear relations pairs for each 2^{18} guesses of key bits and is $(2^{18}.2^{36}) \approx 2^{54}$.

For a 8 round attack, the complexity for encrypting 2^{32} linear relations pairs for each 2^{16} guesses of key bits and is $(2^{16}.2^{32}) \approx 2^{48}$.

Figure 30 compares time complexity of solving encryption algorithm of INFLEX at inflation factor m=3 with time complexity of solving encryption algorithm of Speck as calculated above.

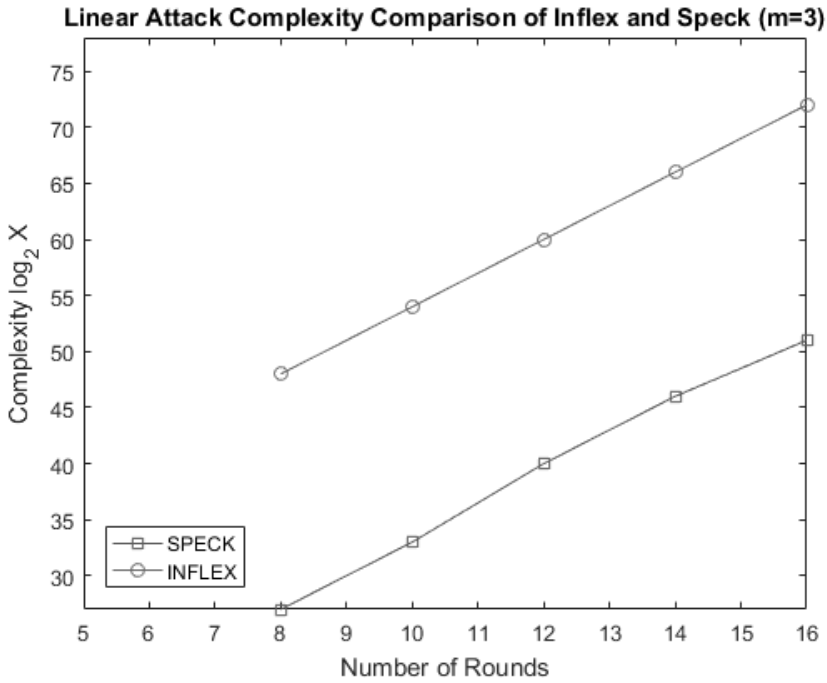


Figure 30. Time Complexity comparison with m=3 inflation

With an inflation of m=2

For a 16 round attack, the complexity for encrypting 2^{44} linear relations pairs for each 2^{22} guesses of key bits and is $(2^{44}.2^{22}) \approx 2^{66}$.

For a 14 round attack, the complexity for encrypting 2^{40} linear relations pairs for each 2^{20} guesses of key bits and is $(2^{40}.2^{20}) \approx 2^{60}$.

For a 12 round attack, the complexity for encrypting 2^{36} linear relations pairs for each 2^{18} guesses of key bits and is $(2^{18}.2^{36}) \approx 2^{54}$.

For a 10 round attack, the complexity for encrypting 2^{32} linear relations pairs for each 2^{16} guesses of key bits and is $(2^{16}.2^{32}) \approx 2^{48}$.

Figure 31 compares time complexity of solving encryption algorithm of INFLEX at inflation factor m=2 with time complexity of solving encryption algorithm of Speck as calculated above.

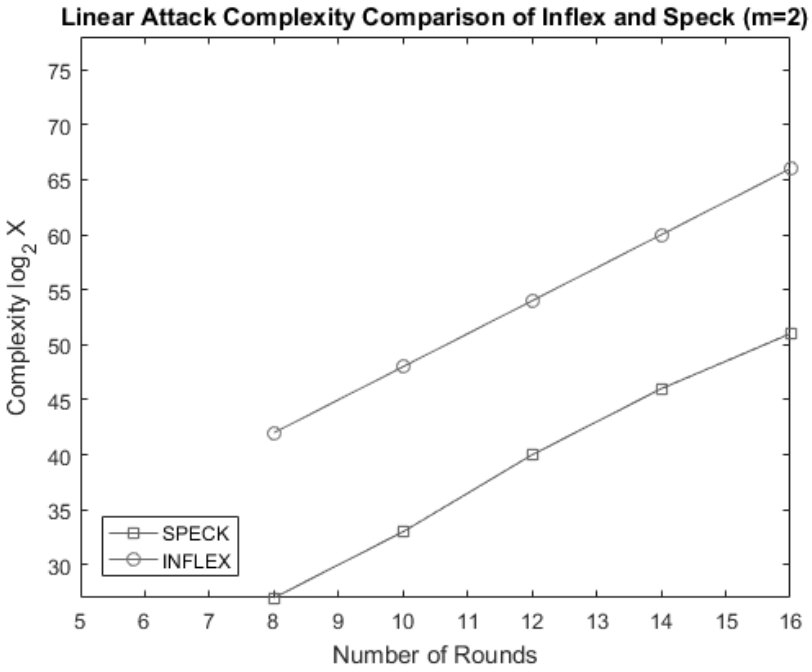


Figure 31. Time Complexity comparison with m=2 inflation

With an inflation of m=1

For a 16 round attack, the complexity for encrypting 2^{40} linear relations pairs for each 2^{20} guesses of key bits and is $(2^{40}.2^{20}) \approx 2^{60}$.

For a 14 round attack, the complexity for encrypting 2^{36} linear relations pairs for each 2^{18} guesses of key bits and is $(2^{36}.2^{18}) \approx 2^{54}$.

For a 12 round attack, the complexity for encrypting 2^{32} linear relations pairs for each 2^{16} guesses of key bits and is $(2^{32}.2^{16}) \approx 2^{48}$.

Figure 32 compares time complexity of solving encryption algorithm of INFLEX at inflation factor m=1 with time complexity of solving encryption algorithm of Speck as calculated above.

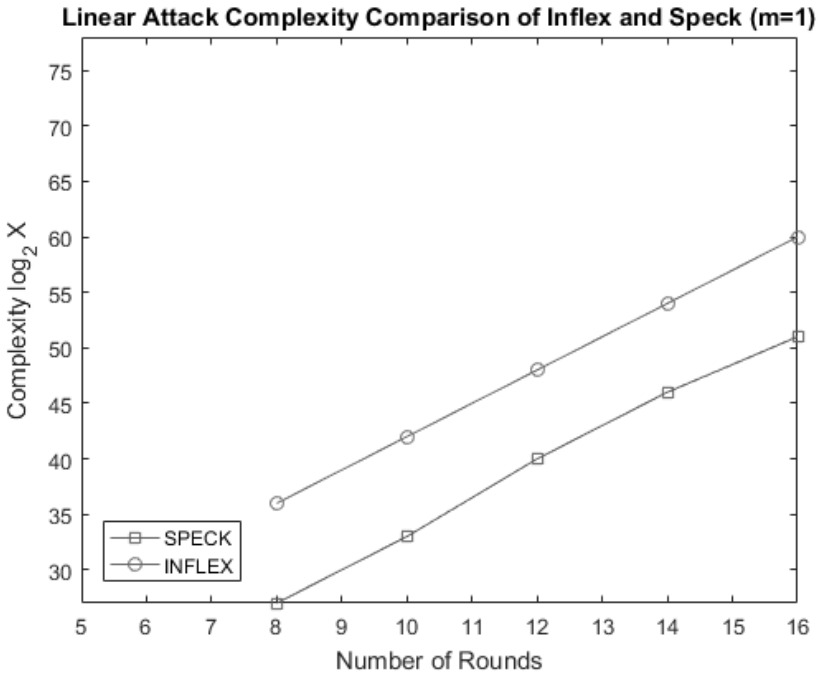


Figure 32. Time Complexity comparison with m=1 inflation

8.5 Differential Cryptanalysis

While nonlinearity of a cipher secures it against a linear attack, and the adversary needs to linearize nonlinear characteristic to the best approximation to launch a linear attack. A linearized equation of the form:

$C = P \oplus k \; and \; C' = P' \oplus k$

Is not useful for a differential attack, as the difference of above results in the following equation, $C \oplus C' = P \oplus P'$

And does not lead to any information about the key. For a nonlinear function of a cipher the difference between inputs (Plaintext) is not the same as the difference between outputs (Ciphertext) for all possible pair of input–output differences, as the difference in the corresponding Ciphertext may depend on the key and hence some useful information about the key may be extracted. This idea is the principal behind the differential cryptanalysis.

Mounting Differential Attack on Speck

Consider the nonlinear operation of the Speck round function

$F(x_1, y_1) = x_2 = (x_1 \boxplus y_1) \oplus k$

Let the difference of the input be denote by $\Delta x_1 = x_1 \oplus x_1' \; and \; \Delta y_1 = y_1 \oplus y_1'$ and the difference of their corresponding outputs $Y \; and \; Y' be \; \Delta Y = F(x_1, y_1) \oplus F(x_1', y_1')$ then by

selecting a random key 2^n (65,536 pairs in case of Speck 2n input block) differential pairs of input and output can be created. There are some difference which are obtained multiple times with specific difference of input (x and x'). These similar difference are grouped together to form the pairs holding same input difference and corresponding output difference.

A pair of input and output differences to a function forms a *characteristic* of the function. For example in case of an 8-bit input if the difference pair of input and output (01_H,$1C_H$) occurs 32 times, then the characteristic (01_H,$1C_H$) holds a probability of 32/128 = 0.250.

When a differential cryptanalysis is applied to an iterated cipher, the adversary observes the propagation of differences between the Plaintext to subsequent rounds and tries to obtain pair of differences which hold a high probability.

Based on the above concept of constructing *differential characteristic* of the round function, a difference distribution table (DDT) is to be created for Speck round function and differential pairs of high probabilities can be shortlisted for mounting a differential attack. Computing a full DDT for modular addition operation would require $4\ X\ 2^{3n}$ bytes of memory [19] and is therefore impractical even for *n>16*. A Biryukov [19] proposed a partial DDT with a threshold of probabilities to discard low probability differential pairs by using a brute force differential characteristic algorithm. The best found *differential characteristic* for Speck32 as calculated with a differential routine are listed in the Table 7 below:

r	Δ_L	Δ_R	$\log_2 p$
0	8054	A900	-0
1	0	A402	-3
2	A402	3408	-3
3	50C0	80E0	-8
4	181	203	-4
5	C	800	-5
6	2000	0	-3
7	40	40	-1
8	8040	8140	-1
9	40	542	-2
	$\sum_r \log_2 p_r$	-30	
	pDDT	2^{30}	

Table 7. Best differential characteristic for differential analysis of Speck

A Biryukov [19] extended the attack to 11 rounds by adding 1 round at the top and 1 round at the bottom of 9-round attack and used the 9 round differential trail with the probability of 2^{-30}. By encrypting 2^{30} pair of Plaintext such that ($\Delta_L^1 = 8054, \Delta_R^1 = A9000$) then it is expected to produce $2^{30} X \frac{1}{2^{30}} = 1$ pair of Plaintext satisfying the $\Delta^2 and\ \Delta^{10}$ and $2^{30} X \frac{1}{2^{28}} = 4$ pairs of Plaintext satisfying the $\Delta^2\ and\ \Delta^9$.

It was observed that last 7 bits after the modular addition at round-10 are always 100 0000. This implies that Δ^{10} is of the form of **** **** *100 0000. Where * represents an unknown bit. Hence 2^{30} pairs of Plaintext/Ciphertext can be filtered for the output difference of 7 bits at Δ^{10}. Which reduces the number of Plaintext/Ciphertext to $2^{30-7} = 2^{23}$. This filtering enables guessing 16 bits of K^{10} and 11 bits of K^9 including 1 carry bit of modular addition. For the remaining 64-27=37 bits brute force exhaustive search can be used.

8.6 Differential Attack Complexity

The complexity for decrypting 2^{23} Ciphertext pairs for each 2^{28} guesses of key bits and a carry bit is $(2^{28}.2^{23}).\frac{1}{11} \approx 2^{47}$. However the 27 bit key guessing was done in 2^{18} count for the satisfaction of the differential characteristic $\Delta_L^9, \Delta_R^9 = (8040,8140)$ and for the remainder of 37 bits a 2^{37} brute force guess dominates the attack complexity with $2^{18}.2^{37} = 2^{55}$.

Extending Speck differential attack to INFLEX

INFLEX, in addition to modular addition nonlinear component has input inflation function. The probability of selecting a pattern is increased by 1/32*1/16 = 0.0625. This probability of expansion will remain same for all round of cipher. By adding this probability to 9 round attack we get INFLEX linear attack probability for 9 rounds as 2^{-34} whereas the requirement for 2^{34} Plaintext/Ciphertext differential pair exceed the entire code book of 2^{32} Plaintext/Ciphertext pairs. The Attack complexity increases to $2^{21}.2^{37} = 2^{58}$ for m=1 and for m=4 it increases to $2^{27}.2^{37} = 2^{64}$.

8.7 INFLEX Comparison with SPECK

With minimum inflation of m=1					
Cipher	Rounds		Data	Complexity	Attack
	Total	Attacked			
Speck	22	16	2^{34}	2^{51}	Linear
		14	2^{30}	2^{45}	
		12	2^{27}	2^{40}	
INFLEX	20	16	2^{40}	2^{60}	Linear
		14	2^{36}	2^{54}	
		12	2^{32}	2^{48}	
Speck	22	11	2^{30}	2^{55}	Differential
INFLEX	20	11	2^{33}	2^{58}	Differential

Table 8. Complexity comparison for m=1 inflation

With maximum inflation of m=4					
Cipher	Rounds		Data	Complexity	Attack
	Total	Attacked			
Speck	22	16	2^{34}	2^{51}	Linear
		14	2^{30}	2^{45}	
		12	2^{27}	2^{40}	
INFLEX	20	16	2^{52}	2^{78}	Linear
		14	2^{48}	2^{72}	
		12	2^{44}	2^{66}	
		10	2^{40}	2^{60}	
		8	2^{36}	2^{54}	
		5	2^{30}	2^{45}	
Speck	22	11	2^{30}	2^{55}	Differential
INFLEX	20	11	2^{39}	2^{64}	Differential

Table 9. Complexity comparison for m=4 inflation

9. Conclusion

9.1 Summary

In this project we have carried out complexity analysis and hardware implementation of extensible modulo addition [15] encryption algorithm on a 32-bit lightweight FPGA based block cipher called INFLEX. The primary design consideration has been an ultra-lightweight cipher that offers a scalable level of security comparable with distinguished lightweight block cipher of 32-bit block size and a 64-bit key.

INFLEX architecture characteristic primarily originates from the generalized Feistel-network with an ARX round function and highly-diffusive and scalable block expansion feature. We have presented a thorough security analysis, particularly for the linear and differential attacks and compared complexity result with Speck32/64 Cipher. Although the result reveal sufficient security of full-round INFLEX, it even outperform Speck32/64 on attack complexity, which encourages us to reduce number of rounds and increase execution time further, however, its security naturally needs to be studied further. Like any other novel proposals, we only recommend beta-version deployment of INFLEX and strongly encourage its analysis.

9.2 Future Work

Future work may involve a hardware implementation on FPGA, ASIC or SOC with compression techniques implemented on the output Ciphertext or serialization of the output in sub-blocks of 32-bits for expanded Ciphertext in order to further shrink hardware resources. It is anticipated that for the serialization of maximum security inflated block a maximum of 16 clock cycles will be added for one block encryption and the throughput could reduce to 88 Kbps at 100Khz from 160Kbp, this is still well higher than the minimum required throughput of lightweight block cipher. The future prospects are very bright and development of a marketable cipher product is highly recommended.

Appendix- A

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity inflex is
Generic(KEY_SIZE : integer := 64 ; BLOCK_SIZE : integer  := 32; inflate : integer := 1;
ROUND_LIMIT: integer := 22);

Port (SYS_CLK,RST : in std_logic; BUSY : out  std_logic; CONTROL : in
std_logic_vector(1 downto 0);
KEY : in  std_logic_vector (KEY_SIZE - 1 downto 0);
BLOCK_INPUT : in  std_logic_vector (BLOCK_SIZE - 1 downto 0);
BLOCK_OUTPUT : out  std_logic_vector (BLOCK_SIZE*(2**inflate)-1 downto 0));-- 255
for m=3; 127 for m=2; 63 for m=1; 31 for m=0;
end inflex;


architecture Behavioral of inflex is
------------------------------------------------------------
-- Cipher Constants
constant WORD_SIZE : integer range 0 to 32 := BLOCK_SIZE / 2;
constant KEY_WORDS_M : integer range 0 to 4 := KEY_SIZE / WORD_SIZE;
constant ALPHA_SHIFT : integer range 0 to 7 := 7;
constant BETA_SHIFT : integer range 0 to 3 := 2;
------------------------------------------------------------

-- inflate look up table
type ARRAY_inflateLUT is array(0 to (2*2**4)) of std_logic_vector( 2**4- 1 downto 0);
signal inflate_Lut: ARRAY_inflateLUT;
```

```
type ARRAY_inflateKeyLUT is array(0 to (2*2**3 - 1)) of std_logic_vector( 2**3 - 1
downto 0);
signal inflate_key_Lut: ARRAY_inflateKeyLUT;
-- Key Schedule Storage Array
type ARRAY_ROUNDxWORDSIZE is array(0 to (ROUND_LIMIT - 1)) of
std_logic_vector(WORD_SIZE*2**(inflate)-1 downto 0);
signal key_schedule: ARRAY_ROUNDxWORDSIZE;
signal round_key : std_logic_vector(WORD_SIZE*2**(inflate)-1 downto 0);
type ARRAY_PARTKEYxWORD is array (0 to KEY_WORDS_M-1) of
std_logic_vector(WORD_SIZE*2**(inflate)-1 downto 0);
signal key_l : ARRAY_PARTKEYxWORD;
signal key_feedback : ARRAY_PARTKEYxWORD;
signal key_gen_round_output : STD_LOGIC_VECTOR(BLOCK_SIZE*2**(inflate)-1
downto 0);

----------------------------------------------------
-- Fiestel Structure Signals
signal b_buf : STD_LOGIC_VECTOR(WORD_SIZE*2**inflate - 1 downto 0);
signal a_buf : STD_LOGIC_VECTOR(WORD_SIZE*2**inflate - 1 downto 0);
signal encryption_round_output : STD_LOGIC_VECTOR(BLOCK_SIZE*2**(inflate)-1
downto 0);
signal decryption_round_output : STD_LOGIC_VECTOR(BLOCK_SIZE*2**(inflate)-1
downto 0);
signal inflate_round_output : STD_LOGIC_VECTOR(BLOCK_SIZE*2**inflate - 1 downto
0);
signal inflate_key_round_output : STD_LOGIC_VECTOR(KEY_SIZE*2**(inflate)-1
downto 0);
----------------------------------------------------

----------------------------------------------------
-- State Machine Signals
type state is (Reset,Idle,Key_Schedule_Generation_Run,Key_Schedule_Generation_Finish,
Cipher_Start,Cipher_Run,Cipher_Finish_1,Cipher_Finish_2,Cipher_Latch);
signal pr_state,nx_state : state;
```

```
------------------------------------------------------

------------------------------------------------------
-- Round Counting Signals
signal round_count : integer range 0 to (ROUND_LIMIT - 1);
signal inv_round_count : integer range 0 to (ROUND_LIMIT - 1);
signal round_count_mux : integer range 0 to (ROUND_LIMIT - 1);
signal cipher_direction : std_logic;
------------------------------------------------------
--inflate

function  Inflate_Round (bl : std_logic_vector(BLOCK_SIZE - 1 downto 0) ) return
std_logic_vector is variable inflate_output : std_logic_vector(BLOCK_SIZE*2**inflate - 1
downto 0);
variable j0 : integer :=0;
variable j1 : integer :=0;
variable k : integer :=2**inflate;
variable inflatelutread : std_logic_vector(15 downto 0);



begin

if inflate < 1 then
inflate_output := bl;
else
inflate_LUT(0)   <= "1111111111111110";
inflate_LUT(1)   <= "1111111111111101";
inflate_LUT(2)   <= "1111111111111011";
inflate_LUT(3)   <= "1111111111110111";
inflate_LUT(4)   <= "1111111111101111";
inflate_LUT(5)   <= "1111111111011111";
inflate_LUT(6)   <= "1111111110111111";
inflate_LUT(7)   <= "1111111101111111";
```

```
inflate_LUT(8)   <= "1111111011111111";
inflate_LUT(9)   <= "1111110111111111";
inflate_LUT(10)  <= "1111101111111111";
inflate_LUT(11)  <= "1111011111111111";
inflate_LUT(12)  <= "1110111111111111";
inflate_LUT(13)  <= "1101111111111111";
inflate_LUT(14)  <= "1011111111111111";
inflate_LUT(15)  <= "0111111111111111";
inflate_LUT(16)  <= "0000000000000001";
inflate_LUT(17)  <= "0000000000000010";
inflate_LUT(18)  <= "0000000000000100";
inflate_LUT(19)  <= "0000000000001000";
inflate_LUT(20)  <= "0000000000010000";
inflate_LUT(21)  <= "0000000000100000";
inflate_LUT(22)  <= "0000000001000000";
inflate_LUT(23)  <= "0000000010000000";
inflate_LUT(24)  <= "0000000100000000";
inflate_LUT(25)  <= "0000001000000000";
inflate_LUT(26)  <= "0000010000000000";
inflate_LUT(27)  <= "0000100000000000";
inflate_LUT(28)  <= "0001000000000000";
  inflate_LUT(29)  <= "0010000000000000";
  inflate_LUT(30)  <= "0100000000000000";
  inflate_LUT(31)  <= "1000000000000000";

for i in 0 to (BLOCK_SIZE - 1) loop
if j0 > k-1 then
j0:=0;
end if;

if j1 > k-1 then
j1:=0;
end if;
if (bl(i) = '1')  then
```

```
inflatelutread := inflate_LUT(j1+16);
inflate_output := inflate_output (BLOCK_SIZE*2**(inflate)-1 downto k/2)   &
inflatelutread((k/2)-1 downto 0);
inflate_output := to_stdlogicvector(to_bitvector(inflate_output) sla k/2);
j1:=j1+1;
end if;
if (bl(i) = '0') then
inflatelutread := inflate_LUT(j0);
inflate_output := inflate_output (BLOCK_SIZE*2**(inflate)-1 downto k/2) &
inflatelutread((k/2)-1 downto 0);
inflate_output := to_stdlogicvector(to_bitvector(inflate_output) sla k/2);
end if;
end loop;
end if;
return inflate_output;
end Inflate_Round;
--inflate key
function  Inflate_Key_Round (bl : std_logic_vector(KEY_SIZE - 1 downto 0) ) return
std_logic_vector is
variable inflate_key_output : std_logic_vector(KEY_SIZE*2**(inflate)-1 downto 0);

variable j0 : integer :=0;
variable j1 : integer :=0;
variable k : integer :=2**(inflate)/2;
variable inflatekeylutread : std_logic_vector(7 downto 0);
begin
if inflate < 1 then
inflate_key_output := bl;
else
inflate_key_LUT(0) <= "11111110";
inflate_key_LUT(1) <= "11111101";
inflate_key_LUT(2) <= "11111011";
inflate_key_LUT(3) <= "11110111";
inflate_key_LUT(4) <= "11101111";
```

```
inflate_key_LUT(5) <= "11011111";
inflate_key_LUT(6) <= "10111111";
inflate_key_LUT(7) <= "01111111";
inflate_key_LUT(8)  <= "00000001";
inflate_key_LUT(9)  <= "00000010";
inflate_key_LUT(10) <= "00000100";
inflate_key_LUT(11) <= "00001000";
inflate_key_LUT(12) <= "00010000";
inflate_key_LUT(13) <= "00100000";
inflate_key_LUT(14) <= "01000000";
inflate_key_LUT(15) <= "10000000";
for i in 0 to (KEY_SIZE - 1) loop
if j0 > k-1 then
j0:=0;
end if;
if j1 > k-1 then
j1:=0;
end if;
if (bl(i) = '1')  then
--

inflatekeylutread := inflate_key_LUT(j1+8);
inflate_key_output := inflate_key_output(KEY_SIZE*2**(inflate)-1 downto k/2) &
inflatekeylutread((K/2)-1 downto 0);
inflate_key_output := to_stdlogicvector(to_bitvector(inflate_key_output) sla k/2);
j1:=j1+1;
end if;
if (bl(i) = '0') then
inflatekeylutread := inflate_key_LUT(j0);
inflate_key_output := inflate_key_output(KEY_SIZE*2**(inflate)-1 downto k/2) &
inflatekeylutread((K/2)-1 downto 0);
inflate_key_output := to_stdlogicvector(to_bitvector(inflate_key_output) sla k/2);
j0:=j0+1;
end if;
```

```
end loop;
end if;
return inflate_key_output;
end Inflate_Key_Round;
--end inflate key

--Encryption round
function  Encrypt_Round(b, a, key_i : std_logic_vector(WORD_SIZE*2**(inflate)-1 downto
0)) return std_logic_vector is
variable b_unsigned : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable a_unsigned : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable r_shift_alpha : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable l_shift_beta : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable adder: unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable key_xor : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable cross_xor : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable encrypt_output : std_logic_vector(BLOCK_SIZE*2**(inflate)-1 downto 0);
begin
b_unsigned := unsigned(b);
a_unsigned := unsigned(a);
r_shift_alpha := b_unsigned(ALPHA_SHIFT - 1 downto 0) &
b_unsigned(WORD_SIZE*2**(inflate)-1 downto ALPHA_SHIFT);
l_shift_beta := a_unsigned(WORD_SIZE*2**(inflate) - (BETA_SHIFT + 1) downto 0) &
a_unsigned((WORD_SIZE*2**(inflate) -1) downto (WORD_SIZE*2**(inflate) -
BETA_SHIFT));
adder := r_shift_alpha + a_unsigned;
key_xor := adder xor unsigned(key_i);
cross_xor := l_shift_beta xor key_xor;
encrypt_output := std_logic_vector(key_xor) & std_logic_vector(cross_xor);
return encrypt_output;
end Encrypt_Round;

--decryption round
```

```
function  Decrypt_Round(b, a, key_i : std_logic_vector(WORD_SIZE*2**(inflate)-1 downto
0)) return std_logic_vector is
variable b_unsigned : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable a_unsigned : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable l_shift_alpha : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable r_shift_beta : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable subtractor: unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable key_xor : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable cross_xor : unsigned(WORD_SIZE*2**(inflate)-1 downto 0);
variable decrypt_output : std_logic_vector(BLOCK_SIZE*2**(inflate)-1 downto 0);
begin
b_unsigned := unsigned(b);
a_unsigned := unsigned(a);
cross_xor := b_unsigned xor a_unsigned;
r_shift_beta := cross_xor(BETA_SHIFT - 1 downto 0) &
cross_xor(WORD_SIZE*2**(inflate) -1 downto BETA_SHIFT);
key_xor := b_unsigned xor unsigned(key_i);
subtractor := key_xor - r_shift_beta;
l_shift_alpha := subtractor(WORD_SIZE*2**(inflate) - (ALPHA_SHIFT + 1) downto 0) &
subtractor((WORD_SIZE*2**(inflate) -1) downto (WORD_SIZE*2**(inflate) -
ALPHA_SHIFT));

decrypt_output := std_logic_vector(l_shift_alpha) & std_logic_vector(r_shift_beta);
return decrypt_output;
end Decrypt_Round;
begin
------------------------------------------------------------------
-- State Machine Processes
------------------------------------------------------------------
State_Machine_Head : process (SYS_CLK) ----State Machine Master Control
begin
if (SYS_CLK'event and SYS_CLK='1') then
if (RST = '1') then
pr_state <= RESET;
```

```
else
pr_state <= nx_state;
end if;
end if;
end process; -- State_Machine_Head

State_Machine_Body : process (CONTROL, round_count, pr_state) ---State Machine State Definitions
begin
case pr_state is
when Reset =>  --Master Reset State
nx_state <= Idle;
when Idle =>
if (CONTROL = "01") then
nx_state <= Key_Schedule_Generation_Run;
elsif (CONTROL = "11" or CONTROL = "10") then
nx_state <= Cipher_Start;
else
nx_state <= Idle;
end if;
when Key_Schedule_Generation_Run =>
if (round_count = ROUND_LIMIT - 2) then
nx_state <= Key_Schedule_Generation_Finish;
else
nx_state <= Key_Schedule_Generation_Run;
end if;
when Key_Schedule_Generation_Finish =>
nx_state <= Idle;
when Cipher_Start =>
nx_state <= Cipher_Run;
when Cipher_Run =>
if (round_count = ROUND_LIMIT - 2) then
nx_state <= Cipher_Finish_1;
else
```

```
nx_state <= Cipher_Run;
end if;
when Cipher_Finish_1 =>
nx_state <= Cipher_Finish_2;
when Cipher_Finish_2 =>
nx_state <= Cipher_Latch;
when Cipher_Latch =>
nx_state <= Idle;
end case;
end process;
--------------------------------------------------------------------
-- END State Machine Processes
--------------------------------------------------------------------
--------------------------------------------------------------------
-- Register Processes
--------------------------------------------------------------------
Cipher_Direction_Flag : process(SYS_CLK)
begin
if SYS_CLK'event and SYS_CLK = '1' then
if (pr_state = Reset) then
cipher_direction <= '0';
elsif (pr_state = Idle) then
cipher_direction <= CONTROL(0);
end if ;
end if;
end process;
Busy_Flag_Generator : process(SYS_CLK)
begin
if SYS_CLK'event and SYS_CLK = '1' then
if (pr_state = Reset or (pr_state = Idle and CONTROL /= "00")) then
BUSY <= '1';
elsif ((pr_state = Idle and CONTROL = "00") or pr_state = Cipher_Latch or pr_state =
Key_Schedule_Generation_Finish) then
BUSY <= '0';
```

```
end if;
end if;
end process ; -- Busy_Flag_Generator
Key_Schedule_Generator : process(SYS_CLK)
begin
if SYS_CLK'event and SYS_CLK = '1' then
if (pr_state = Idle) then
Init_Gen_Regs : for i in 0 to (KEY_WORDS_M -1) loop
key_l(i) <= inflate_key_round_output(((i + 1) * WORD_SIZE*2**inflate) - 1 downto (i *
WORD_SIZE*2**inflate));
end loop ; -- Update_Gen_Regs
elsif (pr_state = Key_Schedule_Generation_Run or pr_state =
Key_Schedule_Generation_Finish) then
for i in 0 to (KEY_WORDS_M - 1) loop
key_l(i) <= key_feedback(i);
end loop;
end if;
end if;
end process ; -- Key_Schedule_Generator
Main_Cipher_Process : process(SYS_CLK)
begin
if SYS_CLK'event and SYS_CLK = '1' then
-- Load for Encryption/Decryption
if (pr_state = Idle) then
if (CONTROL(1) = '1') then
a_buf <= inflate_round_output(WORD_SIZE*2**(inflate)-1 downto 0);
b_buf <= inflate_round_output(BLOCK_SIZE*2**(inflate)-1 downto
WORD_SIZE*2**(inflate));
end if;
-- Run Cipher Engine
elsif (pr_state = Cipher_Run  or pr_state = Cipher_Finish_1 or pr_state = Cipher_Finish_2)
then
if (cipher_direction = '1') then  -- Encryption
a_buf <= encryption_round_output(WORD_SIZE*2**(inflate)-1 downto 0);
```

```
b_buf <= encryption_round_output(BLOCK_SIZE*2**(inflate)-1 downto
WORD_SIZE*2**(inflate));
else  -- Decryption
a_buf <= decryption_round_output(WORD_SIZE*2**(inflate)-1 downto 0);
b_buf <= decryption_round_output(BLOCK_SIZE*2**(inflate)-1 downto
WORD_SIZE*2**(inflate));
end if;
end if;
end if;
end process ;
Output_Buffer : process(SYS_CLK)
begin
if SYS_CLK'event and SYS_CLK = '1' then
if (pr_state = Cipher_Latch) then
BLOCK_OUTPUT <= b_buf & a_buf;
end if;
end if;
end process ; -- Output_Buffer
---------------------------------------------------------------------
-- END Register Processes
---------------------------------------------------------------------
---------------------------------------------------------------------
-- RAM Processes
---------------------------------------------------------------------
Key_Schedule_Array: process (SYS_CLK)
begin
if (SYS_CLK'event and SYS_CLK = '1') then
round_key <= key_schedule(round_count_mux);
if (pr_state = Key_Schedule_Generation_Run or pr_state =
Key_Schedule_Generation_Finish) then
key_schedule(round_count) <= key_l(0);
end if;
end if;
end process;
```

```
--------------------------------------------------------------------
-- End RAM Processes
--------------------------------------------------------------------
--------------------------------------------------------------------
-- Counter Processes
--------------------------------------------------------------------
Round_Counter : process(SYS_CLK)
begin
if (SYS_CLK'event and SYS_CLK = '1') then
if (pr_state = Reset) then
round_count <= 0;
inv_round_count <= 0;
elsif (pr_state = Idle) then
round_count <= 0;
inv_round_count <= ROUND_LIMIT - 1;
elsif (pr_state = Cipher_Start or pr_state = Cipher_Run or pr_state =
Key_Schedule_Generation_Run) then
round_count <= round_count + 1;
inv_round_count <= inv_round_count - 1;
end if ;
end if ;
end process;
--------------------------------------------------------------------
-- END Counter Processes
--------------------------------------------------------------------
--------------------------------------------------------------------
-- Async Signals
--------------------------------------------------------------------
round_count_mux <= round_count when cipher_direction = '1' else inv_round_count;
key_gen_round_output <= Encrypt_Round(key_l(1), key_l(0),
std_logic_vector(to_unsigned(round_count, WORD_SIZE*2**(inflate))));
inflate_round_output <= Inflate_Round(BLOCK_INPUT);
inflate_key_round_output <= Inflate_Key_Round(key);
encryption_round_output <= Encrypt_Round(b_buf, a_buf, round_key);
```

```
decryption_round_output <= Decrypt_Round(b_buf, a_buf, round_key);
key_feedback(0) <= key_gen_round_output(WORD_SIZE*2**(inflate)-1 downto 0);
key_feedback(KEY_WORDS_M - 1) <=
key_gen_round_output(BLOCK_SIZE*2**(inflate)-1 downto WORD_SIZE*2**(inflate));
Keys_3 : if (KEY_WORDS_M = 3) generate
begin
key_feedback(1) <= key_l(2);
end generate;
Keys_4 : if (KEY_WORDS_M = 4) generate
begin
key_feedback(1) <= key_l(2);
key_feedback(2) <= key_l(3);
end generate;
end Behavioral;
```

References

[1] Dinu, Daniel et al. “Triathlon of Lightweight Block Ciphers for the Internet of Things.” *IACR Cryptology ePrint Archive* 2015.

[2] Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.”PRESENT: An Ultra-Lightweight Block Cipher.” *In: Paillier, P., Verbauwhede, I. (eds.) CHES. Lecture Notes in Computer Science*, vol. 4727, pp. 450–466. Springer (2007).

[3] Gong, Z., Nikova, S., Law, Y.W.: “KLEIN: A New Family of Lightweight Block Ciphers.” *In: Juels, A., Paar, C. (eds.) RFIDSec. Lecture Notes in Computer Science*, vol. 7055, pp. 1–18. Springer (2011)

[4] Christophe Cannière, Orr Dunkelman, and Miroslav Knežević. “KATAN and KTANTAN A Family of Small and Efficient Hardware-Oriented Block Ciphers.” *In Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '09)*, Christophe Clavier and Kris Gaj (Eds.). Springer-Verlag, Berlin, Heidelberg, 2009, pp. 272-288.

[5] Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.J.B. “The LED Block Cipher”. In *Cryptographic Hardware and Embedded Systems -- CHES 2011: 13th International Workshop, Nara, Japan*, September 28 -- October 1, 2011. Pp 326-341

[6] Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J., Chee, S. “HIGHT: A New Block Cipher Suitable for Low-Resource Device”. *In: Goubin, L., Matsui, M.(eds.) CHES. Lecture Notes in Computer Science*, vol. 4249, pp. 46–59. Springer 2006.

[7] Knudsen, L. R., Leander, G., Poschmann, A., & Robshaw, M. J. “PRINTcipher: A Block Cipher for IC-Printing”. *In CHES*, Vol. 6225, 2010, pp. 16-32.

[8] Leander, G., Paar, C., Poschmann, A., & Schramm, K. “New lightweight DES variants”. In *International Workshop on Fast Software Encryption* Springer, Berlin, Heidelberg. 2007, pp. 196-210

[9] Wu, Wenling, and Lei Zhang. "LBlock: a lightweight block cipher." In *Applied Cryptography and Network Security*, pp. 327-344., 2011.

[10] Shibutani, Kyoji, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. "Piccolo: An ultra-lightweight blockcipher." *In CHES*, vol. 6917, 2011, pp. 342-357.

[11] Mace, Franois, Franois-Xavier Standaert, and Jean-Jacques Quisquater. "ASIC implementations of the block cipher sea for constrained applications." In *Proceedings of the Third International Conference on RFID Security-RFIDSec*, vol. 2007, pp. 103-114.

[12] Suzaki, T., Minematsu, K., Morioka, S. and Kobayashi, E., “TWINE: A Lightweight Block Cipher for Multiple Platforms”. In *Selected Areas in Cryptography* Vol. 7707, pp. 339-354, August 2012.

[13] Beaulieu, R., Treatman-Clark, S., Shors, D., Weeks, B., Smith, J. and Wingers, L., “The SIMON and SPECK lightweight block ciphers”. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE* (pp. 1-6).

[14] Virgil Gligor D. "Light-Weight Cryptography–How Light is Light." In *Keynote presentation at the Information Security Summer School, Florida State University. Available for download at http://www. sait. fsu. edu/conferences/2005/is3/resources/slides/gligorv-cryptolite. ppt*. 2005, pp 1-29

[15] Partap Siddavaatam, Reza Sedaghat and M. H. Cheng, "An adaptive security framework with extensible computational complexity for cipher systems," in *11th International Conference for Internet Technology and Secured Transactions (ICITST), Barcelona,* 2016, pp.133-140

[16] Matsui, M.: “Linear Cryptoanalysis Method for DES Cipher”. In: *Helleseth, T. (ed.) EUROCRYPT. Lecture Notes in Computer Science*, Springer vol. 765, pp. 386–397. 1993.

[17] Liu, Yu, Kai Fu, Wei Wang, Ling Sun, and Meiqin Wang. "Linear cryptanalysis of reduced-round SPECK." *Information Processing Letters* 116, no. 3, pp 259-266, 2016.

[18] Yao, Yuan, Bin Zhang, and Wenling Wu. "Automatic search for linear trails of the SPECK family." *International Information Security Conference*. Springer, Cham, 2015., pp.158–176.

[19] Alex Biryukov, Arnab Roy, and Vesselin Velichkov. "Differential analysis of block ciphers SIMON and SPECK." *International Workshop on Fast Software Encryption*. Springer, Berlin, Heidelberg, 2014, pp 546-570.

[20] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. "Differential and Linear Cryptanalysis Using Mixed-Integer Linear Programming." *Inscrypt* 7537, 2011, pp 57-76.

[21] David Kahn, *The Codebreakers: The Story of Secret Writing*, New York: Macmillan Publishing Co., 1967. Pp 147

[22] Douglas Wikstrom, *Foundations of Cryptography*, lecture notes, KTH Royal Institute of Technology. 2016, pp 1-25