Applications of Machine Learning in Cryptology

Scientific Analysis Group

DRDO

Metcalfe House Complex

Delhi - 110 054

Under the supervision of

Girish Mishra
Scientist 'E', SAG

Submitted By:

Aayush Jain
F-417/17

Applications of Machine Learning in Cryptology

Aayush Jain

Cluster Innovation Centre

University of Delhi

CERTIFICATE

I hereby certify that Mr. Aayush Jain, Roll No. F-417/17 of Cluster Innovation Centre has undergone industrial training at our organization. He worked on "Applications of Machine Learning in Cryptology" project during the training under my supervision.

During his tenure with us we found him sincere and hard working.

Signature of the Supervisor

Girish Mishra
Scientist 'E'
SAG, DRDO

Acknowledgements

I feel honored in expressing my profound sense of gratitude and indebtedness towards my guide Dr. Girish Mishra, Scientist 'E' Scientific Analysis Group, DRDO, Delhi for the confidence shown in me by giving me an opportunity to work on a new idea and explore under his guidance.

The pragmatic and invaluable advice of my guide kept me going through the critical phases of my project. I am indebted to him for his insightful and encouraging words that have been driving force of my project. His sincere guidance and industrious attitude inspired me to reach beyond limits.

My sincere thanks to all my friends for moral support. I would like to express my deep sense of gratitude to Defense Research and Development Organization, Metcalf House for giving me an opportunity to work in this organization.

I thank once again to all who inspired & helped me during the project to make the venture a success.

Delhi

Aayush Jain
F-417/17

Abstract

Cipher is a plain text encrypted using a specific key. Cryptology is the study of those cipher text. Various encryption methods are available for encrypting a plain text such as AES, **PRESENT, FeW** etc.

In this summer internship I have compared different rounds of PRESENT and FeW lightweight cipher on the basis of randomness (generated after each round) using neural network toolbox of MATLAB. The inherited biases present in FeW were exploited using the said toolbox.

Furthermore, a method to create an image out of given text on which the **DFFT** (discreate fast Fourier transform) is applied to get the **log transformed** images. Above method is applied on various types of texts and random data.

A MATLAB function is also described to give the spectrum image of the text and random rata. Lastly, **DWT** (Discreate Wavelet Transform) was implemented to see the different types of decomposition of an image.

Table of Contents

List of Figures

1 Introduction

1.1 Introduction About the Organization - Defense Research & Development Organization(DRDO)

Defense Research & Development Organization (DRDO) works under Department of Defense Research and Development of Ministry of Defense. DRDO dedicatedly working towards enhancing self-reliance in Defense Systems and undertakes design & development leading to production of world class weapon systems and equipment in accordance with the expressed needs and the qualitative requirements laid down by the three services.

DRDO is working in various areas of military technology which include aeronautics, armaments, combat vehicles, electronics, instrumentation engineering systems, missiles, materials, naval systems, advanced computing, simulation and life sciences. DRDO while striving to meet the Cutting-edge weapons technology requirements provides ample spinoff benefits to the society at large thereby contributing to the nation building.

1.2 Scientific Analysis Group (SAG)

1.2.1 Historical Background

Scientific Analysis Group was established in 1963 for evolving new scientific methods for design and analysis of communication systems. It was in Central Secretariat Complex and consists of 12 scientists. This group was placed under direct control of Chief Controller (R&D) in 1973 and become a full-fledged Directorate in R&D Headquarters. In 1976, SAG started undertaking R&D projects on mathematical, communication and speech analysis. Due to the increasing responsibilities of SAG, the manpower component of SAG underwent expansion and additional accommodation was allotted to SAG at Metcalfe House Complex. SAG was further entrusted with R&D work in the field of electronics. Work related to evaluating communication equipment to be introduced in Services was taken up during 1980. The manpower structure was again revised and the electronic facilities were enhanced. Presently SAG is housed in two independent buildings with a total manpower of 190 scientists /technical and other staff.

1.2.2 Areas of Work

1. Evaluation and Grading of Crypto systems
2. Cryptanalysis of Symmetric and Asymmetric Crypto Systems
3. Traffic Analysis
4. Development of Tools and Techniques for Information Security and Network Security

1.2.3 Achievements

Development of many Software Packages based on advanced mathematical techniques such as:

1. Boolean function
2. Binary sequence
3. Algebraic concepts - Universal algebra and finite fields
4. Artificial Intelligence & ANN
5. Development of Test-bed for Evaluation of Crypto-products
6. Genetic Algorithms
7. Security Evaluation of Networks
8. Side Channel Attacks
9. Design and Analysis of Communication Protocols.

1.2.4 Products

SAG is the lab working on the projects and other R&D activities directly related to use by Services. Some of the electronic systems are in use by other Govt Agencies.

1.3 Basic Introduction About the Project

The need to keep information secret, especially in communications, is obvious in many circumstances such as military, diplomatic, and business affairs. Therefore, a way of secret communication was developed, by hiding the existence of a message, is known as steganography. This way of secret communication has its limits as the message is revealed if it is found. Hence, in parallel with the development of steganography, there was the evolution of Cryptology. The aim of Cryptology is not to hide the existence of a message, but rather to hide its meaning.

Cryptologic research has historically been done by governments and kept secret. Only the last decades there has been a widespread open research in cryptology. Basics of Cryptology is discussed in chapter 2.

One notable product of the research was AES or Advanced Encryption Standard. It is arguably the most widely used block cipher in the world but today's world is filled up with small electronic gadgets (credit cards etc.). These small pieces of plastic have now intervened with our life style. We use these gadgets for personal use and hence want them to be good enough to keep our private information safe. Ciphers can do the trick but they require high end CPU's for the purpose. To fit all those bulky units (CPU, RAM, PCB etc.) into a thin sheet and at the same time make it sturdy enough for daily use is practically impossible.

To counter this problem Lightweight ciphers were introduced. They require less resources but are strong enough to keep our information safe for quite a time. One such cipher is PRESENT. It is an ultra-lightweight block cipher, developed by the Orange Labs (France), Ruhr University Bochum (Germany) and the Technical University of Denmark in 2007. PRESENT is designed by Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. The algorithm is notable for its compact size (about 2.5 times smaller than AES). Another example of a lightweight cipher is FeW which uses a mix of Feistel and generalized Feistel structures to achieve high

efficiency in software-based department. It was developed by M. Kumar, S. K. Pal and A. Panigrahi. I will discuss about PRESENT and FeW in Chapter 3.

Another thing to discuss is Machine Learning. It is a field of computer science that gives computers the ability to learn without being explicitly programmed. Evolved from the study of pattern recognition and computational learning theory in artificial intelligence, machine learning explores the study and construction of algorithms that can learn from and make predictions on data. The example of machine learning used in this report is Neural Network. They are computing systems inspired by the biological neural networks learn to do tasks by considering examples, generally without task-specific programming. In this report, comparison between the outputs of different rounds of PRESENT using Neural Network Toolbox of MATLAB is discussed in Chapter 4.

Next things discussed in this report are the Discrete Fourier Transformation and Discrete Wavelet Transformation. These will be discussed in Chapter 5 along with some functions written in MATLAB. Finally, in chapter 6, using the Neural Network Toolbox of MATLAB, I present a method to exploit the inherited biases present in intermediate rounds and final cipher text of FeW. The report is concluded in chapter 7.

2 Basics of Cryptology

2.1 Cryptology

Cryptology is a method of using advanced mathematical principles in storing and transmitting data in a particular form so that only those whom it is intended can read and process it. Encryption is a key concept in Cryptology, it is a process whereby a message is encoded in a format that cannot be read or understood by an eavesdropper. The information a sender wishes to transmit to a receiver is the plaintext, while the unreadable text that results from encrypting the plaintext is the cipher text. A plain text from a user can be encrypted to a ciphertext, then send through a communication channel and no eavesdropper can interfere with the plain text. When it reaches the receiver end, the ciphertext is decrypted to the original plain text.

Encryption is accomplished by combining the plaintext with the key to yield the cipher text.

$$C = E_k (M)$$

Where C is ciphertext, E is encryption algorithm and M is plaintext.

Decryption is the inverse process, where the ciphertext is converted back to plaintext.

$$M = D_k (C)$$

Where D is decryption Algorithm.

2.2 Classic Encryption Techniques

2.2.1 Caesar Cipher

The earliest known use of a substitution cipher, and the simplest, was by Julius Caesar. The Caesar cipher involves replacing each letter of the alphabet with the letter standing three places further down the alphabet.

For example,

Plain: meet me after the toga party

Cipher: PHHW PH DIWHU WKH WRJD SDUWB

If it is known that a given ciphertext is a Caesar cipher, then a brute-force cryptanalysis is easily performed: Simply try all the 25 possible keys.

2.2.2 Play-fair Cipher

The best-known multiple-letter encryption cipher is the Playfair, which treats bigrams in the plaintext as single units and translates these units into ciphertext bigrams. The Playfair algorithm is based on the use of a 5 x 5 matrix of letters constructed using a keyword.

Despite this level of confidence in its security, the Playfair cipher is relatively easy to break because it still leaves much of the structure of the plaintext language intact. A few hundred letters of ciphertext are generally sufficient.

2.2.3 Hill Cipher

Hill cipher was developed by the mathematician Lester Hill in 1929. The encryption algorithm takes m successive plaintext letters and substitutes for them m ciphertext letters. The substitution is determined by m linear equations in which each character is assigned a numerical value (a = 0, b = 1 ... z = 25).

Although the Hill cipher is strong against a ciphertext-only attack, it is easily broken with a known plaintext attack.

2.2.4 One – Time Pad

An Army Signal Corp officer, Joseph Mauborgne, proposed a cipher that yields the ultimate in security. Mauborgne suggested using a random key that is as long as the message, so that the key need not be repeated. In addition, the key is to be used to encrypt and decrypt a single message, and then is discarded. Each new message requires a new key of the same length as the new message. Such a scheme, known as a one-time pad, is unbreakable. It produces random output that bears no statistical relationship to the plaintext. Because the ciphertext contains no information whatsoever about the plaintext, there is simply no way to break the code.

Plaintext $\oplus$ Key = Ciphertext

Consider the ciphertext,

ANKYODKYUREPFJBYOJDSPLREYIUNOFDOIUERFPLUYTS

Two different decryptions using two different keys:

Ciphertext: ANKYODKYUREPFJBYOJDSPLREYIUNOFDOIUERFPLUYTS

Key: pxlmvmsydofuyrvzwctnlebnecvgdupahfzzlmnyih

Plaintext: mr mustard with the candlestick in the hall

Ciphertext: ANKYODKYUREPFJBYOJDSPLREYIUNOFDOIUERFPLUYTS

Key: mfugpmiydgaxgoufhkllmhsqdqogtewbqfgyovuhwt

Plaintext: miss scarlet with the knife in the library

Suppose that a cryptanalyst had managed to find these two keys. Two plausible plaintexts are produced. How will the cryptanalyst decide which is the correct decryption?

In fact, given any plaintext of equal length to the ciphertext, there is a key that produces that plaintext. Therefore, if you did an exhaustive search of all possible keys, you would end up with many legible plaintexts, with no way of knowing which the intended plaintext was. Therefore, the code is unbreakable.

The security of the one-time pad is entirely due to the randomness of the key. If the stream of characters that constitute the key is truly random, then the stream of characters that constitute the ciphertext will be truly random. Thus, there are no patterns or regularities that a cryptanalyst can use to attack the ciphertext.

The one-time pad offers complete security but, in practice, has two fundamental difficulties:

1. There is the practical problem of making large quantities of random keys. Any heavily used system might require millions of random characters on a regular basis. Supplying truly random characters in this volume is a significant task.
2. Even more daunting is the problem of key distribution and protection. For every message to be sent, a key of equal length is needed by both sender and receiver. Thus, a mammoth key distribution problem exists.

Because of these difficulties, the one-time pad is of limited utility, and is useful primarily for low bandwidth channels requiring very high security.

2.3 Symmetric and Asymmetric Ciphers

2.3.1 Symmetric ciphers

This is the simplest kind of encryption that involves only one secret key to cipher and decipher information. Symmetrical encryption is an old and best-known technique. It uses a secret key that can either be a number, a word or a string of random letters. It is a blended with the plain text of a message to change the content in a particular way. The sender and the recipient should know the secret key that is used to encrypt and decrypt all the messages. Blowfish, AES, RC4, DES, RC5, and RC6 are examples of symmetric encryption. The most widely used symmetric algorithm is AES-128, AES-192, and AES-256.

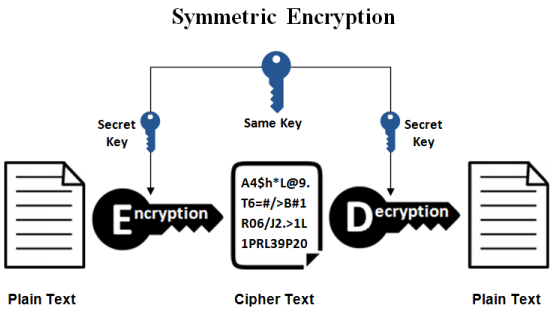


FIGURE 2.1. Symmetric Encryption

Source: https://www.ssl2buy.com/wiki/wp-content/uploads/2015/12/Symmetric-Encryption.png

2.3.2 Asymmetric ciphers

Asymmetrical encryption is also known as public key Cryptology, which is a relatively new method, compared to symmetric encryption. Asymmetric encryption uses two keys to encrypt a plain text. Secret keys are exchanged over the Internet or a large network. It ensures that malicious persons do not misuse the keys. It is important to note that anyone with a secret key can decrypt the message and this is why asymmetrical encryption uses two related keys to boosting security. A public key is made freely available to anyone who might want to send you a message. The second private key is kept a secret so that you can only know.

A message that is encrypted using a public key can only be decrypted using a private key, while also, a message encrypted using a private key can be decrypted using a public key. Security of the public key is not required because it is publicly available and can be passed over the internet.

Asymmetric encryption is mostly used in day-to-day communication channels, especially over the Internet. Popular asymmetric key encryption algorithm includes ElGamal, RSA, DSA, Elliptic curve techniques, PKCS.

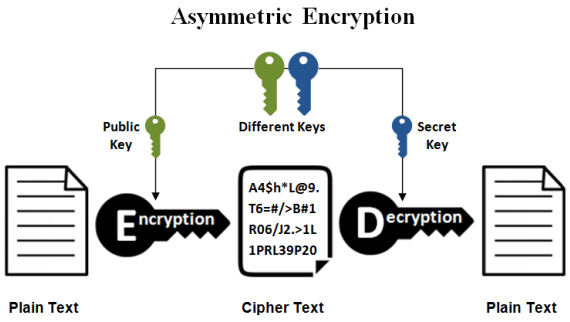


FIGURE 2.2. Asymmetric Encryption

Source: https://www.ssl2buy.com/wiki/wp-content/uploads/2015/12/Asymmetric-Encryption.png

2.4 Block Ciphers and Stream Ciphers

Symmetric ciphers can be further classified into stream ciphers and block ciphers.

2.4.1 Block Cipher

A block cipher is one in which a block of plaintext is treated as a whole and used to produce a ciphertext block of equal length. Typically, a block size of 64 or 128 bits is used. A block cipher is an encryption algorithm that encrypts a fixed size of n-bits of data - known as a block - at one time. The usual sizes of each block are 64 bits, 128 bits, and 256 bits. So, for example, a 64-bit block cipher will take in 64 bits of plaintext and encrypt it into 64 bits of ciphertext. In cases where bits of plaintext are shorter than the block size, padding schemes are called into play. Majority of the symmetric ciphers used today are actually block ciphers.

Examples:

1. DES - DES, which stands for Data Encryption Standard, used to be the most popular block cipher in the world and was used in several industries. It's still popular today, but only because it's usually included in historical discussions of encryption algorithms. The DES algorithm became a standard in the US in 1977. However, it's already been proven to be vulnerable to brute force attacks and other cryptanalytic methods. DES is a 64-bit cipher that works with a 64-bit key. Actually, 8 of the 64 bits in the key are parity bits, so the key size is technically 56 bits long.
2. 3DES - As its name implies, 3DES is a cipher based on DES. It's practically DES that's run three times. Each DES operation can use a different key, with each key being 56 bits long. Like DES, 3DES has a block size of 64 bits. Although 3DES is many times stronger than DES, it is also much slower (about 3x slower). Because many organizations found 3DES to be too slow for many applications, it never became the ultimate successor of DES. That distinction is reserved for the next cipher in our list - AES.
3. AES - A US Federal Government standard since 2002, AES or Advanced Encryption Standard is arguably the most widely used block cipher in the world. It has a block size of 128 bits and supports three possible key sizes - 128, 192, and 256 bits. The longer the key size, the stronger is the encryption. However, longer keys also result in longer processes of encryption.

2.4.2 Stream Cipher

A stream cipher is a cipher that encrypts a digital data stream one bit or one byte at a time. Examples of classical stream ciphers are the auto keyed Vigenère cipher and the Vernam cipher. It uses an infinite stream of pseudorandom bits as the key. For a stream cipher implementation to remain secure, its pseudorandom generator should be unpredictable and the key should never be reused. Stream ciphers are designed to approximate an idealized cipher, known as the One-Time Pad.

3 Lightweight Block Cipher

3.1 Introduction

With the establishment of the AES the need for new block ciphers has been greatly reduced; for almost all block cipher applications the AES is an excellent and preferred choice. However, despite recent implementation advances, the AES is not suitable for extremely constrained environments such as RFID tags and sensor networks.

3.2 PRESENT lightweight block cipher

PRESENT is an ultra-lightweight block cipher. Its block size is 64 bits and the key size can be 80 bits or 128 bits. The non-linear layer is based on a single 4-bit S-box which was designed with hardware optimizations in mind. PRESENT is intended to be used in situations where low-power consumption and high chip efficiency is desired. The International Organization for Standardization and the International Electrotechnical Commission included PRESENT in the new international standard for lightweight cryptographic methods.

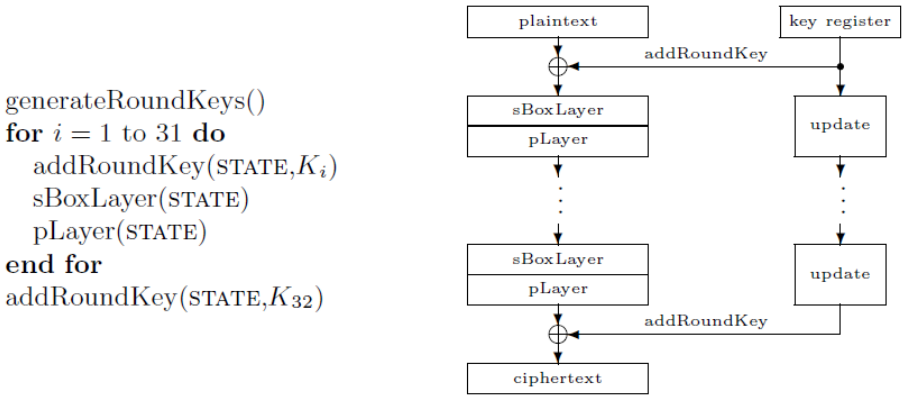


FIGURE 3.1. A top-level algorithmic description of present.

Source: (Bogdanov et al., 2007)

3.3 Implementation of PRESENT

Each of the 31 rounds consists of an xor operation to introduce a round key Ki for 1 ≤ i ≤ 32, where K32 is used for post-whitening, a linear bitwise permutation and a non-linear substitution layer. The non-linear layer

uses a single 4-bit S-box S which is applied 16 times in parallel in each round. The cipher is described in pseudo-code in Figure 3.1, and each stage is now specified in turn.

addRoundKey

Given round key $K_i = K^i_{63}\ldots K^i_0$ for $1 \le i \le 32$ and current STATE $b_{63}\ldots b_0$, addRoundKey consists of the operation for $1 \le j \le 63$,

$$b_j \rightarrow b_j \oplus k^i_j$$

sBoxlayer

The S-box used in PRESENT is a 4-bit to 4-bit S-box $S: F_2^4 \rightarrow F_2^4$. The action of this box in hexadecimal notation is given by Table 1.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 1. sBoxLayer for PRESENT

For sBoxLayer the current STATE $b_{63}\ldots b_0$ is considered as sixteen 4-bit words $w_{15}\ldots w_0$ where $w_i = b_{4*i+3} \parallel b_{4*i+2} \parallel b_{4*i+1} \parallel b_{4*i}$ for $0 \le i \le 15$ and the output nibble $S[w_i]$ provides the updated state values in the obvious way.

PLayer

The bit permutation used in PRESENT is given by Table 2. Bit I of STATE is moved to bit position P(i).

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P(i)	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P(i)	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
P(i)	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
P(i)	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Table 2. PLayer

The key schedule

PRESENT can take keys of either 80 or 128 bits. However, we focus on the version with 80-bit keys. The user-supplied key is stored in a key register K and represented as $k_{79}k_{78}\ldots k_0$. At round I the 64-bit round key $K_i = k_{63}k_{62}\ldots k_0$ consists of the 64 leftmost bits of the current contents of register K. Thus, at round i we have that:

$$K_i = k_{63}k_{62}\ldots k_0 = k_{79}k_{78}\ldots k_{16}$$

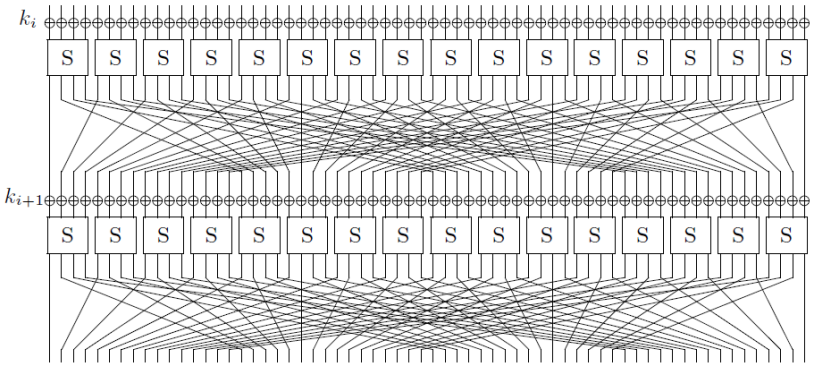


FIGURE 3.2. The S/P network for PRESENT

After extracting the round key K_i, the key register $K_i = k_{79}k_{78}\ldots k_0$ is updated as follows.

1. $[k_{79}k_{78}\ldots k_1k_0] = [k_{18}k_{17}\ldots k_{20}k_{19}]$
2. $[k_{79}k_{78}k_{77}k_{76}] = S[k_{79}k_{78}k_{77}k_{76}]$
3. $[k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus$ round_counter

Thus, the key register is rotated by 61-bit positions to the left, the left-most four bits are passed through the PRESENT S-box, and the round_counter value i is exclusive-ored with bits $k_{19}k_{18}k_{17}k_{16}k_{15}$ of K with the least significant bit of round_counter on the right.

C++ implementation of PRESENT is given in Appendix 1.

3.4 FeW: A Lightweight Cipher

FeW is an Lightweight block cipher which consist of 32 rounds. It generates a cipher text of size 64 bits using the input plain text of same size. FeW uses two options for the size of Master key MK: 80 bits and 128 bits. Based on two key sizes, name of the two versions of FeW, the first version with 80-bit key size as FeW-80 and the second version with 128-bit key size as FeW-128. A round of FeW is shown in Figure 3.3.

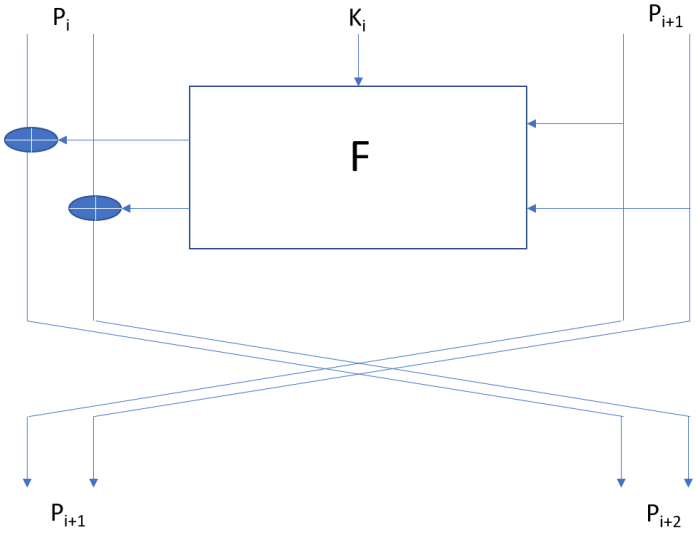


FIGURE 3.3: Single round illustration of FeW

3.5 Implementation of FeW

The FeW lightweight ciphers divide the input plaintext into two equal halves, P0 and P1. Each of these two halves is then passed through round functions, swap function and finally concatenated to form single 64-bit cipher text.

Round functions F

Round functions F is applied on 32-bit word P_{i+1} which is then xor with P_i to get P_{i+2}. Here i is in range, $0 \leq i \leq 31$.

$P_{i+2} \leftarrow P_i \oplus F(P_{i+1}, K_i)$ Where K_i is the i^{th} round key.

It uses two different weight functions WF_1 and WF_2. Weight function WF_1 consists of application of S-box given in Table 3, four times in parallel as non-linear operation and cyclic shifts and xor operation as linear mixing operation L_1. WF_2 on the other hand, consists of application of S-box 4 times in parallel and apply cyclic shifts on V and xor these with V to get Z as output of WF_2 as linear mixing operation L_2 which is different from L_1. Round function is illustrated in Figure 3.4.

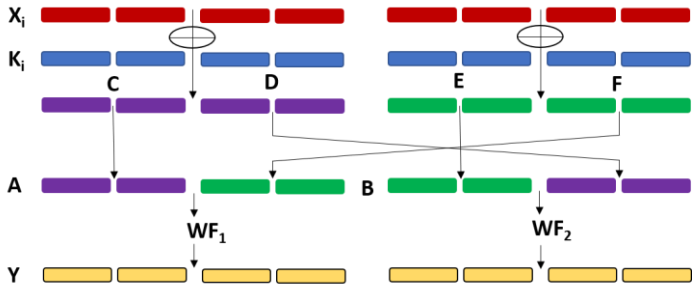


FIGURE 3.4: Illustration of round functions F

Swap function

Output of last round is swapped:

$$(P_{32}P_{32}) \leftarrow (P_{32}, P_{33})$$

Concatenation

32-bits outputs are concatenated to form 64-bit cipher text.

$$(C_0, C_1) \leftarrow (P_{33}, P_{32})$$

$$C_m \leftarrow C_0 || C_1$$

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	2	E	F	5	C	1	9	A	B	4	6	8	0	7	3	D

Table 3. S-box for FeW

Key scheduling for 80 Bits

FeW stores 80-bits master key in register named MK ($MK=k_{79}k_{78}\ldots k_0$). Round subkey RK_0 is obtained by extracting leftmost 16-bits from MK. Subsequent round keys are obtained using following method:

1. For i < 64, update MK in following steps:
   a. MK$\lll$13
   b. Update bits using S-Box in the following way:
      i. $[K_0K_1K_2K_3] \leftarrow S[K_0K_1K_2K_3]$
      ii. $[K_{64}K_{65}K_{66}K_{67}] \leftarrow S[K_{64}K_{65}K_{66}K_{67}]$
      iii. $[K_{76}K_{77}K_{78}K_{79}] \leftarrow S[K_{76}K_{77}K_{78}K_{79}]$
   c. $[K_{68}K_{69}K_{70}K_{71}K_{72}K_{73}K_{74}K_{75}] \leftarrow [K_{68}K_{69}K_{70}K_{71}K_{72}K_{73}K_{74}K_{75}] \oplus [i]_2$
2. Increment i by 1 and extract leftmost 16 bits of current contents of MK as round subkey RKi.

4 Neural Networks

4.1 Introduction

Artificial neural networks (ANNs) are computing systems inspired by the biological neural networks learn to do tasks by considering examples, generally without task-specific programming. They have found most use in applications difficult to express in a traditional computer algorithm using rule-based programming.

An ANN is based on a collection of connected units called artificial neurons. Each connection between neurons can transmit a signal to another neuron. The receiving neuron can process the signal(s) and then signal downstream neurons connected to it. Neurons may have state, generally represented by real numbers, typically between 0 and 1.

Typically, neurons are organized in layers (hidden layers). Different layers may perform different kinds of transformations on their inputs. Signals travel from the first (input), to the last (output) layer, possibly after traversing the layers multiple times.

The original goal of the neural network approach was to solve problems in the same way that a human brain would. Over time, attention focused on matching specific mental abilities. Wide variety of tasks are available for neural networks such as computer vision, speech recognition etc.

4.2 Neural Network Toolbox

Neural Network Toolbox of MATLAB provides algorithms, pretrained models, and apps to create, train, visualize, and simulate both shallow and deep neural networks. We can perform classification, regression, clustering, dimensionality reduction, time-series forecasting, and dynamic system modeling and control.

4.2.1 Deep Learning

Deep learning is part of a broader family of machine learning methods based on learning data representations, as opposed to task-specific algorithms. Learning can be supervised, partially supervised or unsupervised.

Deep learning architectures such as deep neural networks have been applied to fields including computer vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation and bioinformatics where they produced results comparable to and in some cases superior to human experts.

Deep learning networks include convolutional neural networks (ConvNets, CNNs), directed acyclic graph (DAG) network topologies, and autoencoders for image classification, regression, and feature learning. For time-series classification and prediction, the toolbox provides long short-term memory (LSTM) deep learning networks. We can visualize intermediate layers and activations, modify network architecture, and monitor training progress.

4.2.2 Capabilities

For small training sets, we can quickly apply deep learning by performing transfer learning with pretrained deep network models.

To speed up training on large data sets, we can distribute computations and data across multicore processors and GPUs on the desktop.

4.3 Implementation

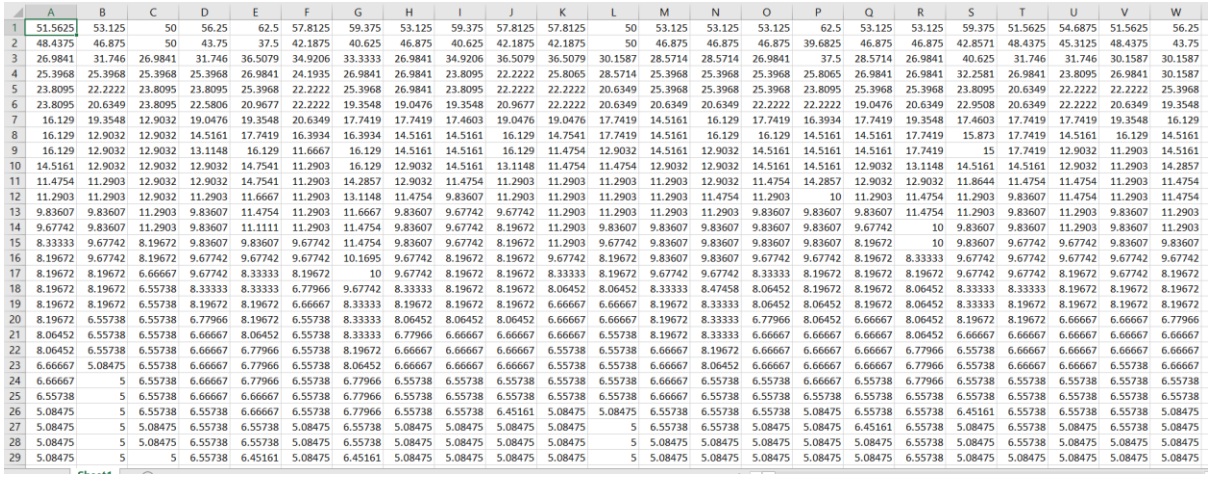
Setting up a neural network for pattern recognition on MATLAB is very easy. There is an application called "Neural-Network Pattern Recognition" which is used to set up a neural network for our feature set.

We need a list of features in our sample data and the target class to get the neural network running.

4.3.1 Features

Features are the special traits which differentiate the data in our sample. Each class has some specific features. In this report,

1. 100, 80-bit random keys were generated and the plain text with all reset bits (0000 0000 0000 0000) was encrypted with all the keys.
2. The cipher texts generated after each round were stored in separate files.
3. Ciphers corresponding to first round were selected one at a time and a given bit pattern was searched along a sliding window.
4. 126 bit-patterns were searched for each cipher.
5. The percentages of these patterns arranged in descending order to form an array were called features.
6. A matrix of 126 rows and 100 columns was thus made using these percentages.
7. Above procedure was repeated for each round.
8. All the matrices were then concatenated into a single 126 x 3100 matrix.
9. This is called feature list (fig 4.1).



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
1	51.5625	53.125	50	56.25	62.5	57.8125	59.375	53.125	59.375	57.8125	57.8125	50	53.125	53.125	53.125	62.5	53.125	53.125	59.375	51.5625	54.6875	51.5625	56.25
2	48.4375	46.875	50	43.75	37.5	42.1875	40.625	46.875	40.625	42.1875	42.1875	50	46.875	46.875	46.875	39.6825	46.875	46.875	42.8571	48.4375	45.3125	48.4375	43.75
3	26.9841	31.746	26.9841	31.746	36.5079	34.9206	33.3333	26.9841	34.9206	36.5079	36.5079	30.1587	28.5714	28.5714	26.9841	37.5	28.5714	26.9841	40.625	31.746	31.746	30.1587	30.1587
4	25.3968	25.3968	25.3968	25.3968	26.9841	24.1935	26.9841	26.9841	23.8095	22.2222	25.8065	28.5714	25.3968	25.3968	25.3968	25.8065	26.9841	26.9841	32.2581	26.9841	23.8095	26.9841	30.1587
5	23.8095	22.2222	23.8095	23.8095	25.3968	22.2222	25.3968	26.9841	23.8095	22.2222	22.2222	20.6349	25.3968	25.3968	25.3968	23.8095	25.3968	25.3968	23.8095	20.6349	22.2222	22.2222	25.3968
6	23.8095	20.6349	23.8095	22.5806	20.9677	22.2222	19.3548	19.0476	19.3548	20.9677	22.2222	20.6349	20.6349	20.6349	22.2222	22.2222	19.0476	20.6349	22.9508	20.6349	22.2222	20.6349	19.3548
7	16.129	19.3548	12.9032	19.0476	19.3548	20.6349	17.7419	17.7419	17.4603	19.0476	19.0476	17.7419	14.5161	16.129	17.7419	16.3934	17.7419	19.3548	17.4603	17.7419	17.7419	19.3548	16.129
8	16.129	12.9032	12.9032	14.5161	17.7419	16.3934	16.3934	14.5161	14.5161	16.129	14.7541	17.7419	14.5161	16.129	16.129	14.5161	14.5161	17.7419	15.873	17.7419	14.5161	16.129	14.5161
9	16.129	12.9032	12.9032	13.1148	16.129	11.6667	16.129	14.5161	14.5161	16.129	11.4754	12.9032	14.5161	12.9032	14.5161	14.5161	14.5161	17.7419	15	17.7419	12.9032	11.2903	14.5161
10	14.5161	12.9032	12.9032	12.9032	14.7541	11.2903	16.129	12.9032	14.5161	13.1148	11.4754	11.4754	12.9032	12.9032	14.5161	14.5161	12.9032	13.1148	14.5161	14.5161	12.9032	11.2903	14.2857
11	11.4754	11.2903	12.9032	12.9032	14.7541	11.2903	14.2857	12.9032	11.4754	11.2903	11.2903	11.2903	11.2903	12.9032	11.4754	14.2857	12.9032	12.9032	11.8644	11.4754	11.4754	11.2903	11.4754
12	11.2903	11.2903	12.9032	11.2903	11.6667	11.2903	13.1148	11.4754	9.83607	11.2903	11.2903	11.2903	11.2903	11.4754	11.2903	10	11.2903	11.4754	11.2903	9.83607	11.4754	11.2903	11.4754
13	9.83607	9.83607	11.2903	9.83607	11.4754	11.2903	11.6667	9.83607	9.67742	9.67742	11.2903	11.2903	11.2903	11.2903	9.83607	9.83607	9.83607	11.4754	11.2903	9.83607	11.2903	9.83607	11.2903
14	9.67742	9.83607	11.2903	9.83607	11.1111	11.2903	11.4754	9.83607	9.67742	8.19672	11.2903	9.83607	9.83607	9.83607	9.83607	9.83607	9.67742	10	9.83607	9.83607	11.2903	9.83607	11.2903
15	8.33333	9.67742	8.19672	9.83607	9.83607	9.67742	11.4754	9.83607	9.67742	8.19672	11.2903	9.67742	9.83607	9.83607	9.83607	9.83607	8.19672	10	9.83607	9.67742	9.67742	9.83607	9.83607
16	8.19672	9.67742	8.19672	9.67742	9.67742	9.67742	10.1695	9.67742	8.19672	8.19672	9.67742	8.19672	9.83607	9.83607	9.67742	9.67742	8.19672	8.33333	9.67742	9.67742	9.67742	9.67742	9.67742
17	8.19672	8.19672	6.66667	9.67742	8.33333	8.19672	10	9.67742	8.19672	8.19672	8.33333	8.19672	9.67742	9.67742	8.33333	8.19672	8.19672	8.19672	9.67742	9.67742	8.19672	9.67742	8.19672
18	8.19672	8.19672	6.55738	8.33333	8.33333	6.77966	9.67742	8.33333	8.19672	8.19672	8.06452	8.06452	8.33333	8.47458	8.06452	8.19672	8.19672	8.06452	8.33333	8.33333	8.19672	8.19672	8.19672
19	8.19672	8.19672	6.55738	8.19672	8.19672	6.66667	8.33333	8.19672	8.19672	8.19672	6.66667	6.66667	8.19672	8.33333	8.06452	8.06452	8.19672	8.06452	8.33333	8.19672	8.19672	8.19672	8.19672
20	8.19672	6.55738	6.55738	6.77966	8.19672	6.55738	8.33333	8.06452	8.06452	8.06452	6.66667	6.66667	8.19672	8.33333	6.77966	8.06452	6.66667	8.06452	8.19672	8.19672	6.66667	6.66667	6.77966
21	8.06452	6.55738	6.55738	6.66667	8.06452	6.55738	8.33333	6.77966	6.66667	6.66667	6.66667	6.55738	8.19672	8.33333	6.66667	6.66667	6.66667	8.06452	6.66667	6.66667	6.66667	6.66667	6.66667
22	8.06452	6.55738	6.55738	6.66667	6.77966	6.55738	8.19672	6.66667	6.66667	6.66667	6.55738	6.55738	6.66667	8.19672	6.66667	6.66667	6.66667	6.77966	6.55738	6.66667	6.66667	6.66667	6.66667
23	6.66667	5.08475	6.55738	6.66667	6.77966	6.55738	8.06452	6.66667	6.66667	6.66667	6.55738	6.55738	6.66667	8.06452	6.66667	6.66667	6.66667	6.77966	6.55738	6.66667	6.66667	6.55738	6.66667
24	6.66667	5	6.55738	6.66667	6.77966	6.55738	6.77966	6.55738	6.55738	6.55738	6.55738	6.55738	6.66667	6.55738	6.55738	6.66667	6.55738	6.77966	6.55738	6.55738	6.55738	6.55738	6.55738
25	6.55738	5	6.55738	6.66667	6.66667	6.55738	6.77966	6.55738	6.55738	6.55738	6.55738	6.55738	6.66667	6.55738	6.55738	6.55738	6.55738	6.55738	6.55738	6.55738	6.55738	6.55738	6.55738
26	5.08475	5	6.55738	6.55738	6.66667	6.55738	6.77966	6.55738	6.55738	6.45161	5.08475	5.08475	6.55738	6.55738	6.55738	5.08475	6.55738	6.55738	6.45161	6.55738	6.55738	6.55738	5.08475
27	5.08475	5	5.08475	6.55738	6.55738	5.08475	6.55738	5.08475	5.08475	5.08475	5.08475	5	6.55738	6.55738	5.08475	5.08475	6.45161	6.55738	5.08475	6.55738	5.08475	6.55738	5.08475
28	5.08475	5	5.08475	6.55738	6.55738	5.08475	6.55738	5.08475	5.08475	5.08475	5.08475	5	5.08475	5.08475	5.08475	5.08475	5.08475	6.55738	5.08475	6.55738	5.08475	5.08475	5.08475
29	5.08475	5	5	6.55738	6.45161	5.08475	6.45161	5.08475	5.08475	5.08475	5.08475	5	5.08475	5.08475	5.08475	5.08475	5.08475	6.55738	5.08475	5.08475	5.08475	5.08475	5.08475

FIGURE 4.1 Feature List

After getting feature list, we set up a hidden layer of neurons to get a model which can describe the features of each class.

4.3.2 Target Class

Target Class is the generalization of sample data with similar characteristics. All the round 1 ciphers belong to one class. Same applies for other round ciphers.

For 31 rounds we need 31 classes.

1. A matrix with 31 rows and 3100 columns was created.
2. For the first 100 columns, the feature list has round 1 ciphers. So, the top row was filled with 1 and the rest with 0.
3. Similarly, for next 100 columns, the feature list has round 2 ciphers. So, the second row was filled with 1 and the rest with 0.
4. Same procedure was repeated for all columns.
5. This matrix is called Target class (fig. 4.2).

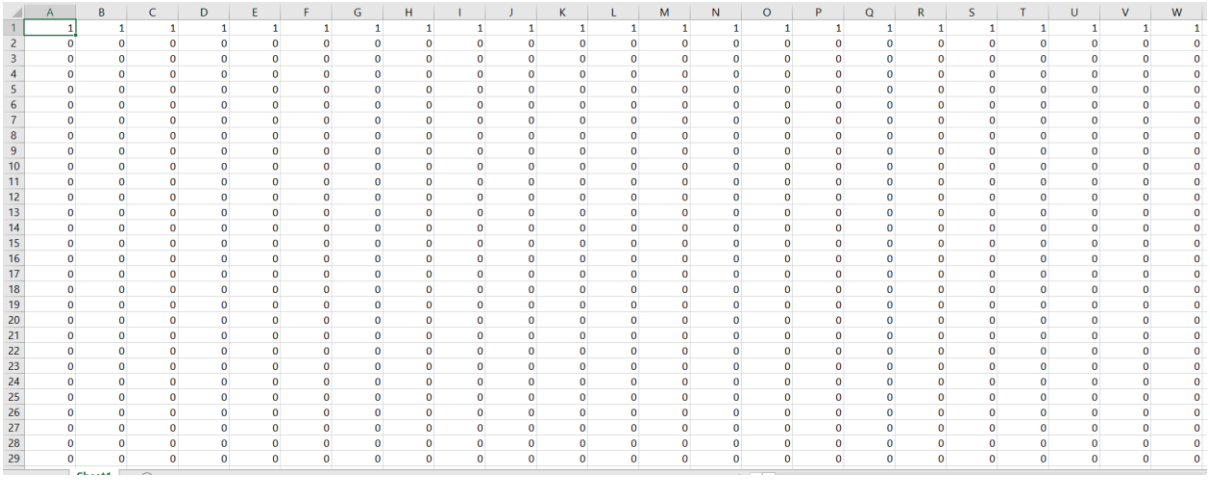


FIGURE 4.2 Target Class

Generally, the number of hidden neurons have to be greater than the number of features. After setting up feature list, target class and hidden neurons we train the network.

After training is finished we plot the confusion matrix (fig. 4.3). It is a specific table layout that allows visualization of the performance of our network. Each row of the matrix represents the instances in a predicted class while each column represents the instances in an actual class.

FIGURE 4.3. Confusion Plot

Images of different confusion plot are given in Appendix 2.

5 Differentiating Plain text and random text using Image Processing

5.1 Fourier Transformation

The Fourier transform decomposes a function of time (a signal) into the frequencies that make it up, in a way similar to how a musical chord can be expressed as the frequencies (or pitches) of its constituent notes. The Fourier transform is called the frequency domain representation of the original signal. The term Fourier transform refers to both the frequency domain representation and the mathematical operation that associates the frequency domain representation to a function of time. The Fourier transform is not limited to functions of time, but in order to have a unified language, the domain of the original function is commonly referred to as the time domain. Functions that are localized in the time domain have Fourier transforms that are spread out across the frequency domain and vice versa, a phenomenon known as the uncertainty principle. In the problem discussed, the data is not continuous but discrete. So instead of Fourier transform, Discrete Fourier Transform (DFT) is used.

5.1.1 Discrete Fourier Transform

The discrete Fourier transform (DFT) converts a finite sequence of equally-spaced samples of a function into a same-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex-valued function of frequency. The interval at which the DTFT is sampled is the reciprocal of the duration of the input sequence. The DFT is therefore said to be a frequency domain representation of the original input sequence.

The DFT is the most important discrete transform, used to perform Fourier analysis in many practical applications. Since it deals with a finite amount of data, it can be implemented in computers by numerical algorithms or even dedicated hardware. These implementations usually employ efficient fast Fourier transform (FFT) algorithms.

5.1.2 Fast Fourier transform (FFT)

A fast Fourier transform (FFT) algorithm computes the discrete Fourier transform (DFT) of a sequence. It converts a signal from its original domain (often time or space) to a representation in the frequency domain. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors. As a result, it manages to reduce the complexity of computing the DFT from O (n^ {2}), which arises if one simply applies the definition of DFT, to O (n\log n), where n is the data size.

5.1.3 Application of FFT

In this report, FFT is applied on images to get the log transformed image. For that I describe a function in MATLAB to create an image out of text.

5.1.3.1 Function getImage

1. The function takes the text as a parameter.
2. It then converts its characters into their corresponding ASCII value.
3. The ideal dimension of the image is then calculated using the length of the text. The ideal dimension must produce a square image.
4. The text is then reshaped into those dimensions to get a text matrix.
5. A color map is selected and the text matrix is scaled to that map.
6. Each character is treated as a pixel and therefore gets an equally scaled RGB values.
7. Those values are stored into a different matrix and are exported to get the image.

Above function is also applied on random HEX data by first mapping the HEX values onto their corresponding Decimal form.

5.1.3.2 Function DFT_image

After getting the image, DFT is applied.

1. Image is loaded into a matrix.
2. DFT of the matrix is calculated using MATLAB command FFT2.
3. Absolute value of the above result gives the Fourier Image of the text.
4. The Absolute value matrix is shifted to origin using MATLAB command FFTSHIFT. It gives the centered Fourier Image.
5. Logarithm of absolute value of FFTSHIFT after adding 1 gives the log transformed image of the text.
6. All the images are exported into secondary storage.

5.1.3.3 Function getSpectrum

This is another function to generate image from the text but this time a frequency spectrum image.

1. Function takes the text as a parameter.
2. It then converts its characters into their corresponding ASCII value.
3. A counter winSize is started from 1 to the size of the text. It defines the size of the window.
4. Frequency of each character is calculated for a particular sized sliding window.
5. All the frequencies are stored in a frequency list which is then averaged out after completion of each window size.
6. Above process is repeated 100 times to get a matrix of 100 rows and 128 columns (characters corresponding to their ASCII).
7. The matrix is then converted into a colored image by the method described in 5.1.3.1.
8. The image is the output of getSpectrum.

5.2 Wavelet Transform

The fundamental idea of wavelet transforms is that the transformation should allow only changes in time extension, but not shape. Unlike the Fourier Transform, we can analyze frequency domain and time domain simultaneously with different resolutions or window size using wavelets (fig 5.1).

Changes in the time extension are expected to conform to the corresponding analysis frequency of the basis function. The higher the required resolution in time, the lower the resolution in frequency has to be. The larger the extension of the analysis windows is chosen, the larger is the value of Δt.

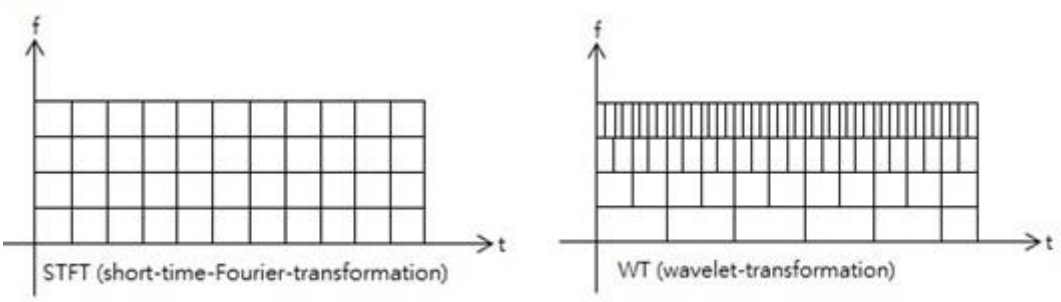


FIGURE 5.1 Comparison of STFT and WT

Source: https://en.wikipedia.org/wiki/Wavelet_transform

There are different types of wavelets. They are generally used image compression. In this report, haar wavelets are used. A function is written in MATLAB to generate discrete wavelet decomposition of an image using DWT2 function of MATLAB. The output of DWT2 generates 4 matrices: approximate, horizontal, vertical and diagonal. DWT2 is then again applied on approximate matrix to further decompose it. Finally, the original image is reproduced using all 4 decompositions.

All the MATLAB codes and Images are attached in Appendix 3.

6 Exploiting Inherited Biases

6.1 Introduction

As discussed in section 3.4, FeW is a lightweight cipher which uses a mix of Feistel and generalized Feistel structures to achieve high efficiency in software-based department. This section focuses on the analysis of lightweight block cipher FeW using the machine learning approach. This approach involves using artificial neural network to find the inherited biases present in the design of FeW.

In this section, cipher output of FeW is used to predict the individual bit of 64-bit long plaintext using neural network-based pattern recognition technique. Available intermediate round data is also used to predict the plaintext.

6.2 Representation of ciphertext or intermediate round data

The main focus is to use available cipher or intermediate round data to predict all 64 bits of input plaintext individually. These data are just some bit sequence and random data does not have any feasible feature. Feature set is represented by the frequency of occurrence of 1-bit, 2-bit, 3-bit, 4-bit, 5-bit and 6-bit sub sequences in sorted order for each subsequence. We only took top 10 most occurring values (in case of 4-bit, 5-bit and 6-bit subsequence) of each sub sequence to get a feature set of 44 dimensions.

This feature set is prepared for each ciphertext sample (ciphertext generated from a given plaintext using a fixed key and FeW encryption algorithm). Thus, we have a set of feature sets that is passed to construct ANN model for training and testing.

6.3 Methodology and Results

Bit stream data from all rounds is collected and used to predict any bit of plaintext in lightweight cipher FeW with a probability more than the chance probability. For the experiment, we took 10000 plaintexts which were used to generate corresponding 10000 ciphertexts using the same key. The intermediate round data was also collected for each of 32 rounds of FeW algorithm.

Pattern Recognition tool of Neural Network (NN) toolbox is MATLAB was used to predict all the 64 bits of plaintext from the cipher text or the intermediate round data collectively. Data sets are categorized with default values of Training (70%), Validation (15%) and Testing (15%). Weights are tuned in line with error during training. Two-layer feed forward network with sigmoid hidden and output neurons is used in this problem. Scaled conjugate gradient backpropagation algorithm is used to train the neural network. For adjusting the synaptic weights, forward phase and backward phase are also included.

Network generalization is measured with the use of Validation samples. These samples are also used for halting the training when measure of generalization reaches maximum. Testing samples can measure the network performance independently after training is done. These samples do not affect the training process.

Following example illustrates the problem. We took a ciphertext as a sample input after validating the NN model. The model is designed as such, it will take the sample input to predict the 1st bit of the plaintext. The number of sample inputs will be used to predict the 1st bit and if the prediction is significantly more than the chance probability i.e. 50%, then the model is consider having knowledge or learned some pattern available in the algorithm of FeW. Similarly, all 64 bits are predicted using above method and if some ith bit is predicted with a probability significantly greater than 50%, the bias could be exploited to mount a cryptanalytic attack. Figure 6.1 shows the confusion plot for the NN model predicting 1st bit using the intermediate round 1 feature set. We also took the intermediate round samples to do the above task of predicting plaintext bits. Until now, the ciphertext was used to predict some specific bits of plain text. The above stated method checks if it can predict any bit of plaintext with a probability better than chance probability (50%). If yes, the biasedness could be exploited by attackers.

Till now, it is concluded that we get 32 sets of bit stream data with each corresponding to one of 32 rounds of FeW lightweight cipher. With each such set, 64 different NN models were trained and validate to predict one of the 64 bits present in the plaintext.

Figure 6.2 shows the detailed results for the above described method. We can observe from the figure that the probability to guess any bit of the 64-bit plaintext is not significantly greater than 50%. The range of predictions we get are from 45.2% to 54.3%. A complete analysis states the there aren't any biases in lightweight cipher FeW that can be exploited using neural network-based pattern recognition approach using mentioned features.

Fig. 6.1. Sample of confusion plot for NNPR model

# Bit	Minimum (%)	Maximum (%)
1	47.7	53.5
2	47.2	52.8
3	45.9	51.5
4	46.9	52.5
5	46.6	53.1
6	48	52.5
7	47.3	53.3
8	47	53.3
9	47.1	53.3
10	46	52.1
11	47.3	52.4
12	47	52.6
13	48.1	52.1
14	47.3	53.9
15	46.6	52.1
16	47.7	52.2
17	47.9	52.7
18	46.3	52
19	47.2	52
20	47.3	52.7
21	47.3	51.9
22	47.3	52.3

# Bit	Minimum (%)	Maximum (%)
23	47.3	52.3
24	48.1	51.4
25	47.7	52.9
26	47.2	53.1
27	47.5	51.1
28	48.2	53
29	47.4	52.6
30	46.3	53
31	46.8	52.3
32	47.7	52.6
33	47.3	52.3
34	47.7	52.4
35	48.3	52
36	47.1	53.1
37	47.7	52.5
38	47.2	52.1
39	46.2	51.5
40	48.1	52.1
41	48.1	51.6
42	47.1	53.5
43	47.5	51.9
44	48.2	53.9

# Bit	Minimum (%)	Maximum (%)
45	48.3	52.3
46	46.9	51.7
47	46.9	53
48	46.9	53.5
49	47.1	52.5
50	47.5	52.3
51	48.1	51.7
52	47.9	52.1
53	46.6	54.3
54	46.9	52.3
55	45.2	53.2
56	47.3	53.1
57	48.6	53.1
58	45.8	51.7
59	47.7	52.1
60	47.6	52
61	47.7	53.3
62	47.3	53.6
63	47.7	52.9
64	46.3	52.6
	MIN = 45.2	MAX = 54.3

Fig. 6.2. Success (%) in prediction of bits at 64-bit positions of plaintext

7 Conclusion

In this report I have described a method to test the randomness of different rounds of PRESENT lightweight block cipher. I also described a method to differentiate between plain text and cipher text using Fourier Transform and Image Processing.

7.1 PRESENT lightweight block cipher

It is concluded from the confusion matrices (Appendix 2) that PRESENT is a strong cipher with a little difference between the outputs of different rounds. As the evidence from the confusion matrices that it is strong enough after round 15 but the round counter to 31 ensures its security from any sort of cryptanalysis. It full fills its goal by offering a level of security commensurate with a 64-bit block size and an 80-bit key. It has implementation requirements similar to many compact stream ciphers. I strongly encourage the analysis of PRESENT ultra-lightweight block cipher.

7.2 Differentiating Plain text and random text using Image Processing

It can be concluded from the results that the method described is not appropriate to differentiate between Plain text and random text. As an evidence, the log transformed images of the two texts are similar.

On the other hand, Spectrum of the image seems promising hence I encourage some further research to be done towards that part.

7.3 Exploiting Inherited Biases

The result of the discussion and experiment shows that the algorithm designed for FeW is able to nullify the attacks by NNPR technique. In future, other block ciphers could be analyzed by using other machine learning tools.

BIBLOGRAPHY

[1] Stallings, William. Cryptology and network security: principles and practices. Pearson Education India, 2006.

[2] Menezes, Alfred J., Paul C. Van Oorschot, and Scott A. Vanstone. Handbook of applied Cryptology. CRC press, 1996.

[3] Bogdanov, Andrey, et al. "PRESENT: An ultra-lightweight block cipher." CHES. Vol. 4727. 2007.

[4] wiki youtube. "Fast Fourier Transform of an Image in Matlab". Online video clip. *YouTube*. https://www.youtube.com/watch?v=nWGirrTlpnk, 2017

[5] Kolhar. "DFT Implementation in MATLAB". Online video clip. *YouTube*. https://www.youtube.com/watch?v=2Q44SyPBOjk&t=512s, 2016

[6] Sengupta. "Discrete Wavelet Transforms". Online video clip. *YouTube*. https://www.youtube.com/watch?v=kp2zPGpKd74 , 2008

[7] Bo Zhu." PRESENT – C". github. https://github.com/bozhu, 2013

APPENDIX 1

Snippet of the PRESENT ultra-lightweight block cipher in C++

```cpp

#ifndef PRESENT_H
#define PRESENT_H

// comment this out if this is used on PC
#define __UINT_T__

#ifndef __UINT_T__
#define __UINT_T__
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned long long uint64_t;
#endif /* __UINT_T__ */

#include <iostream>
using namespace std;

// full-round PRESENT block cipher
#define present(plain, key, cipher) present_rounds((plain), (key), 31, (cipher))

// actually is (sbox[] << 4)
static const uint8_t sbox[16] = {
    0xC0, 0x50, 0x60, 0xB0, 0x90, 0x00, 0xA0, 0xD0, 0x30, 0xE0, 0xF0, 0x80, 0x40, 0x70, 0x10, 0x20,
};

// look-up tables for speeding up permutation layer
static const uint8_t sbox_pmt_3[256] = {
    0xF0, 0xB1, 0xB4, 0xE5, 0xE1, 0xA0, 0xE4, 0xF1, 0xA5, 0xF4, 0xF5, 0xE0, 0xB0, 0xB5, 0xA1, 0xA4,
    0x72, 0x33, 0x36, 0x67, 0x63, 0x22, 0x66, 0x73, 0x27, 0x76, 0x77, 0x62, 0x32, 0x37, 0x23, 0x26,
    0x78, 0x39, 0x3C, 0x6D, 0x69, 0x28, 0x6C, 0x79, 0x2D, 0x7C, 0x7D, 0x68, 0x38, 0x3D, 0x29, 0x2C,
    0xDA, 0x9B, 0x9E, 0xCF, 0xCB, 0x8A, 0xCE, 0xDB, 0x8F, 0xDE, 0xDF, 0xCA, 0x9A, 0x9F, 0x8B, 0x8E,
    0xD2, 0x93, 0x96, 0xC7, 0xC3, 0x82, 0xC6, 0xD3, 0x87, 0xD6, 0xD7, 0xC2, 0x92, 0x97, 0x83, 0x86,
    0x50, 0x11, 0x14, 0x45, 0x41, 0x00, 0x44, 0x51, 0x05, 0x54, 0x55, 0x40, 0x10, 0x15, 0x01, 0x04,
    0xD8, 0x99, 0x9C, 0xCD, 0xC9, 0x88, 0xCC, 0xD9, 0x8D, 0xDC, 0xDD, 0xC8, 0x98, 0x9D, 0x89, 0x8C,
    0xF2, 0xB3, 0xB6, 0xE7, 0xE3, 0xA2, 0xE6, 0xF3, 0xA7, 0xF6, 0xF7, 0xE2, 0xB2, 0xB7, 0xA3, 0xA6,
    0x5A, 0x1B, 0x1E, 0x4F, 0x4B, 0x0A, 0x4E, 0x5B, 0x0F, 0x5E, 0x5F, 0x4A, 0x1A, 0x1F, 0x0B, 0x0E,
    0xF8, 0xB9, 0xBC, 0xED, 0xE9, 0xA8, 0xEC, 0xF9, 0xAD, 0xFC, 0xFD, 0xE8, 0xB8, 0xBD, 0xA9, 0xAC,
    0xFA, 0xBB, 0xBE, 0xEF, 0xEB, 0xAA, 0xEE, 0xFB, 0xAF, 0xFE, 0xFF, 0xEA, 0xBA, 0xBF, 0xAB, 0xAE,
    0xD0, 0x91, 0x94, 0xC5, 0xC1, 0x80, 0xC4, 0xD1, 0x85, 0xD4, 0xD5, 0xC0, 0x90, 0x95, 0x81, 0x84,
    0x70, 0x31, 0x34, 0x65, 0x61, 0x20, 0x64, 0x71, 0x25, 0x74, 0x75, 0x60, 0x30, 0x35, 0x21, 0x24,
    0x7A, 0x3B, 0x3E, 0x6F, 0x6B, 0x2A, 0x6E, 0x7B, 0x2F, 0x7E, 0x7F, 0x6A, 0x3A, 0x3F, 0x2B, 0x2E,
    0x52, 0x13, 0x16, 0x47, 0x43, 0x02, 0x46, 0x53, 0x07, 0x56, 0x57, 0x42, 0x12, 0x17, 0x03, 0x06,
    0x58, 0x19, 0x1C, 0x4D, 0x49, 0x08, 0x4C, 0x59, 0x0D, 0x5C, 0x5D, 0x48, 0x18, 0x1D, 0x09, 0x0C,
};

static const uint8_t sbox_pmt_2[256] = {
    0x3C, 0x6C, 0x2D, 0x79, 0x78, 0x28, 0x39, 0x7C, 0x69, 0x3D, 0x7D, 0x38, 0x2C, 0x6D, 0x68, 0x29,
    0x9C, 0xCC, 0x8D, 0xD9, 0xD8, 0x88, 0x99, 0xDC, 0xC9, 0x9D, 0xDD, 0x98, 0x8C, 0xCD, 0xC8, 0x89,
    0x1E, 0x4E, 0x0F, 0x5B, 0x5A, 0x0A, 0x1B, 0x5E, 0x4B, 0x1F, 0x5F, 0x1A, 0x0E, 0x4F, 0x4A, 0x0B,
    0xB6, 0xE6, 0xA7, 0xF3, 0xF2, 0xA2, 0xB3, 0xF6, 0xE3, 0xB7, 0xF7, 0xB2, 0xA6, 0xE7, 0xE2, 0xA3,
    0xB4, 0xE4, 0xA5, 0xF1, 0xF0, 0xA0, 0xB1, 0xF4, 0xE1, 0xB5, 0xF5, 0xB0, 0xA4, 0xE5, 0xE0, 0xA1,
    0x14, 0x44, 0x05, 0x51, 0x50, 0x00, 0x11, 0x54, 0x41, 0x15, 0x55, 0x10, 0x04, 0x45, 0x40, 0x01,
    0x36, 0x66, 0x27, 0x73, 0x72, 0x22, 0x33, 0x76, 0x63, 0x37, 0x77, 0x32, 0x26, 0x67, 0x62, 0x23,
    0xBC, 0xEC, 0xAD, 0xF9, 0xF8, 0xA8, 0xB9, 0xFC, 0xE9, 0xBD, 0xFD, 0xB8, 0xAC, 0xED, 0xE8, 0xA9,
    0x96, 0xC6, 0x87, 0xD3, 0xD2, 0x82, 0x93, 0xD6, 0xC3, 0x97, 0xD7, 0x92, 0x86, 0xC7, 0xC2, 0x83,
    0x3E, 0x6E, 0x2F, 0x7B, 0x7A, 0x2A, 0x3B, 0x7E, 0x6B, 0x3F, 0x7F, 0x3A, 0x2E, 0x6F, 0x6A, 0x2B,
    0xBE, 0xEE, 0xAF, 0xFB, 0xFA, 0xAA, 0xBB, 0xFE, 0xEB, 0xBF, 0xFF, 0xBA, 0xAE, 0xEF, 0xEA, 0xAB,
    0x34, 0x64, 0x25, 0x71, 0x70, 0x20, 0x31, 0x74, 0x61, 0x35, 0x75, 0x30, 0x24, 0x65, 0x60, 0x21,
    0x1C, 0x4C, 0x0D, 0x59, 0x58, 0x08, 0x19, 0x5C, 0x49, 0x1D, 0x5D, 0x18, 0x0C, 0x4D, 0x48, 0x09,
    0x9E, 0xCE, 0x8F, 0xDB, 0xDA, 0x8A, 0x9B, 0xDE, 0xCB, 0x9F, 0xDF, 0x9A, 0x8E, 0xCF, 0xCA, 0x8B,
    0x94, 0xC4, 0x85, 0xD1, 0xD0, 0x80, 0x91, 0xD4, 0xC1, 0x95, 0xD5, 0x90, 0x84, 0xC5, 0xC0, 0x81,
    0x16, 0x46, 0x07, 0x53, 0x52, 0x02, 0x13, 0x56, 0x43, 0x17, 0x57, 0x12, 0x06, 0x47, 0x42, 0x03,
};

static const uint8_t sbox_pmt_1[256] = {
    0x0F, 0x1B, 0x4B, 0x5E, 0x1F, 0x0A, 0x4E, 0x1F, 0x5A, 0x4F, 0x5F, 0x0E, 0x0B, 0x5B, 0x1A, 0x4A,
    0x27, 0x33, 0x63, 0x76, 0x36, 0x22, 0x66, 0x37, 0x72, 0x67, 0x77, 0x26, 0x23, 0x73, 0x32, 0x62,
    0x87, 0x93, 0xC3, 0xD6, 0x96, 0x82, 0xC6, 0x97, 0xD2, 0xC7, 0xD7, 0x86, 0x83, 0xD3, 0x92, 0xC2,
    0xAD, 0xB9, 0xE9, 0xFC, 0xBC, 0xA8, 0xEC, 0xBD, 0xF8, 0xED, 0xFD, 0xAC, 0xA9, 0xF9, 0xB8, 0xE8,
    0x2D, 0x39, 0x69, 0x7C, 0x3C, 0x28, 0x6C, 0x3D, 0x78, 0x6D, 0x7D, 0x2C, 0x29, 0x79, 0x38, 0x68,
    0x05, 0x11, 0x41, 0x54, 0x14, 0x00, 0x44, 0x15, 0x50, 0x45, 0x55, 0x04, 0x01, 0x51, 0x10, 0x40,
    0x8D, 0x99, 0xC9, 0xDC, 0x9C, 0x88, 0xCC, 0x9D, 0xD8, 0xCD, 0xDD, 0x8C, 0x89, 0xD9, 0x98, 0xC8,
    0x2F, 0x3B, 0x6B, 0x7E, 0x3E, 0x2A, 0x6E, 0x3F, 0x7A, 0x6F, 0x7F, 0x2E, 0x2B, 0x7B, 0x3A, 0x6A,
    0xA5, 0xB1, 0xE1, 0xF4, 0xB4, 0xA0, 0xE4, 0xB5, 0xF0, 0xE5, 0xF5, 0xA4, 0xA1, 0xF1, 0xB0, 0xE0,
    0x8F, 0x9B, 0xCB, 0xDE, 0x9E, 0x8A, 0xCE, 0x9F, 0xDA, 0xCF, 0xDF, 0x8E, 0x8B, 0xDB, 0x9A, 0xCA,
    0xAF, 0xBB, 0xEB, 0xFE, 0xBE, 0xAA, 0xEE, 0xBF, 0xFA, 0xEF, 0xFF, 0xAE, 0xAB, 0xFB, 0xBA, 0xEA,
    0x0D, 0x19, 0x49, 0x5C, 0x1C, 0x08, 0x4C, 0x1D, 0x58, 0x4D, 0x5D, 0x0C, 0x09, 0x59, 0x18, 0x48,
    0x07, 0x13, 0x43, 0x56, 0x16, 0x02, 0x46, 0x17, 0x52, 0x47, 0x57, 0x06, 0x03, 0x53, 0x12, 0x42,
    0xA7, 0xB3, 0xE3, 0xF6, 0xB6, 0xA2, 0xE6, 0xB7, 0xF2, 0xE7, 0xF7, 0xA6, 0xA3, 0xF3, 0xB2, 0xE2,
    0x25, 0x31, 0x61, 0x74, 0x34, 0x20, 0x64, 0x35, 0x70, 0x65, 0x75, 0x24, 0x21, 0x71, 0x30, 0x60,
    0x85, 0x91, 0xC1, 0xD4, 0x94, 0x80, 0xC4, 0x95, 0xD0, 0xC5, 0xD5, 0x84, 0x81, 0xD1, 0x90, 0xC0,
};

static const uint8_t sbox_pmt_0[256] = {
    0xC3, 0xC6, 0xD2, 0x97, 0x87, 0x82, 0x93, 0xC7, 0x96, 0xD3, 0xD7, 0x83, 0xC2, 0xD6, 0x86, 0x92,
    0xC9, 0xCC, 0xD8, 0x9D, 0x8D, 0x88, 0x99, 0xCD, 0x9C, 0xD9, 0xDD, 0x89, 0xC8, 0xDC, 0x8C, 0x98,
    0xE1, 0xE4, 0xF0, 0xB5, 0xA5, 0xA0, 0xB1, 0xE5, 0xB4, 0xF1, 0xF5, 0xA1, 0xE0, 0xF4, 0xA4, 0xB0,
    0x6B, 0x6E, 0x7A, 0x3F, 0x2F, 0x2A, 0x3B, 0x6F, 0x3E, 0x7B, 0x7F, 0x2B, 0x6A, 0x7E, 0x2E, 0x3A,
    0x4B, 0x4E, 0x5A, 0x1F, 0x0F, 0x0A, 0x1B, 0x4F, 0x1E, 0x5B, 0x5F, 0x0B, 0x4A, 0x5E, 0x0E, 0x1A,
    0x41, 0x44, 0x50, 0x15, 0x05, 0x00, 0x11, 0x45, 0x14, 0x51, 0x55, 0x01, 0x40, 0x54, 0x04, 0x10,
    0x63, 0x66, 0x72, 0x37, 0x27, 0x22, 0x33, 0x67, 0x36, 0x73, 0x77, 0x23, 0x62, 0x76, 0x26, 0x32,
    0xCB, 0xCE, 0xDA, 0x9F, 0x8F, 0x8A, 0x9B, 0xCF, 0x9E, 0xDB, 0xDF, 0x8B, 0xCA, 0xDE, 0x8E, 0x9A,
    0x69, 0x6C, 0x78, 0x3D, 0x2D, 0x28, 0x39, 0x6D, 0x3C, 0x79, 0x7D, 0x29, 0x68, 0x7C, 0x2C, 0x38,
    0xE3, 0xE6, 0xF2, 0xB7, 0xA7, 0xA2, 0xB3, 0xE7, 0xB6, 0xF3, 0xF7, 0xA3, 0xE2, 0xF6, 0xA6, 0xB2,
    0xEB, 0xEE, 0xFA, 0xBF, 0xAF, 0xAA, 0xBB, 0xEF, 0xBE, 0xFB, 0xFF, 0xAB, 0xEA, 0xFE, 0xAE, 0xBA,
    0x43, 0x46, 0x52, 0x17, 0x07, 0x02, 0x13, 0x47, 0x16, 0x53, 0x57, 0x03, 0x42, 0x56, 0x06, 0x12,
    0xC1, 0xC4, 0xD0, 0x95, 0x85, 0x80, 0x91, 0xC5, 0x94, 0xD1, 0xD5, 0x81, 0xC0, 0xD4, 0x84, 0x90,
```

```
    0xE9, 0xEC, 0xF8, 0xBD, 0xAD, 0xA8, 0xB9, 0xED, 0xBC, 0xF9, 0xFD, 0xA9, 0xE8, 0xFC, 0xAC, 0xB8,
    0x49, 0x4C, 0x58, 0x1D, 0x0D, 0x08, 0x19, 0x4D, 0x1C, 0x59, 0x5D, 0x09, 0x48, 0x5C, 0x0C, 0x18,
    0x61, 0x64, 0x70, 0x35, 0x25, 0x20, 0x31, 0x65, 0x34, 0x71, 0x75, 0x21, 0x60, 0x74, 0x24, 0x30,
};

void print_cipher(const uint8_t *cipher) {
    int size = sizeof(cipher)/sizeof(cipher[0]);
    for (int i = 0; i < size; ++i) {
        int c = cipher[i];
        cout<<c<<" ";
    }
    cout<<endl;
}

// full-round should be 31, i.e. rounds = 31
// plain and cipher can overlap, so do key and cipher
void present_rounds(const uint8_t *plain, const uint8_t *key, const uint8_t rounds, uint8_t *cipher)
{
    uint8_t round_counter = 1;

    uint8_t state[8];
    uint8_t round_key[10];

    // add key
    state[0] = plain[0] ^ key[0];
    state[1] = plain[1] ^ key[1];
    state[2] = plain[2] ^ key[2];
    state[3] = plain[3] ^ key[3];
    state[4] = plain[4] ^ key[4];
    state[5] = plain[5] ^ key[5];
    state[6] = plain[6] ^ key[6];
    state[7] = plain[7] ^ key[7];

    // update key
    round_key[9] = key[6] << 5 | key[7] >> 3;
    round_key[8] = key[5] << 5 | key[6] >> 3;
    round_key[7] = key[4] << 5 | key[5] >> 3;
    round_key[6] = key[3] << 5 | key[4] >> 3;
    round_key[5] = key[2] << 5 | key[3] >> 3;
    round_key[4] = key[1] << 5 | key[2] >> 3;
    round_key[3] = key[0] << 5 | key[1] >> 3;
    round_key[2] = key[9] << 5 | key[0] >> 3;
    round_key[1] = key[8] << 5 | key[9] >> 3;
    round_key[0] = key[7] << 5 | key[8] >> 3;

    round_key[0] = (round_key[0] & 0x0F) | sbox[round_key[0] >> 4];

    round_key[7] ^= round_counter >> 1;
    round_key[8] ^= round_counter << 7;

    // substitution and permutation
    cipher[0] =
        (sbox_pmt_3[state[0]] & 0xC0) |
        (sbox_pmt_2[state[1]] & 0x30) |
        (sbox_pmt_1[state[2]] & 0x0C) |
        (sbox_pmt_0[state[3]] & 0x03);
    cipher[1] =
        (sbox_pmt_3[state[4]] & 0xC0) |
        (sbox_pmt_2[state[5]] & 0x30) |
        (sbox_pmt_1[state[6]] & 0x0C) |
        (sbox_pmt_0[state[7]] & 0x03);

    cipher[2] =
        (sbox_pmt_0[state[0]] & 0xC0) |
        (sbox_pmt_3[state[1]] & 0x30) |
        (sbox_pmt_2[state[2]] & 0x0C) |
        (sbox_pmt_1[state[3]] & 0x03);
    cipher[3] =
        (sbox_pmt_0[state[4]] & 0xC0) |
        (sbox_pmt_3[state[5]] & 0x30) |
        (sbox_pmt_2[state[6]] & 0x0C) |
        (sbox_pmt_1[state[7]] & 0x03);

    cipher[4] =
        (sbox_pmt_1[state[0]] & 0xC0) |
        (sbox_pmt_0[state[1]] & 0x30) |
        (sbox_pmt_3[state[2]] & 0x0C) |
        (sbox_pmt_2[state[3]] & 0x03);
    cipher[5] =
        (sbox_pmt_1[state[4]] & 0xC0) |
        (sbox_pmt_0[state[5]] & 0x30) |
        (sbox_pmt_3[state[6]] & 0x0C) |
        (sbox_pmt_2[state[7]] & 0x03);

    cipher[6] =
        (sbox_pmt_2[state[0]] & 0xC0) |
        (sbox_pmt_1[state[1]] & 0x30) |
        (sbox_pmt_0[state[2]] & 0x0C) |
        (sbox_pmt_3[state[3]] & 0x03);
    cipher[7] =
        (sbox_pmt_2[state[4]] & 0xC0) |
        (sbox_pmt_1[state[5]] & 0x30) |
        (sbox_pmt_0[state[6]] & 0x0C) |
        (sbox_pmt_3[state[7]] & 0x03);

    print_cipher(cipher);

    for (round_counter = 2; round_counter <= rounds; round_counter++) {
```

```
        state[0] = cipher[0] ^ round_key[0];
        state[1] = cipher[1] ^ round_key[1];
        state[2] = cipher[2] ^ round_key[2];
        state[3] = cipher[3] ^ round_key[3];
        state[4] = cipher[4] ^ round_key[4];
        state[5] = cipher[5] ^ round_key[5];
        state[6] = cipher[6] ^ round_key[6];
        state[7] = cipher[7] ^ round_key[7];

        cipher[0] =
            (sbox_pmt_3[state[0]] & 0xC0) |
            (sbox_pmt_2[state[1]] & 0x30) |
            (sbox_pmt_1[state[2]] & 0x0C) |
            (sbox_pmt_0[state[3]] & 0x03);
        cipher[1] =
            (sbox_pmt_3[state[4]] & 0xC0) |
            (sbox_pmt_2[state[5]] & 0x30) |
            (sbox_pmt_1[state[6]] & 0x0C) |
            (sbox_pmt_0[state[7]] & 0x03);

        cipher[2] =
            (sbox_pmt_0[state[0]] & 0xC0) |
            (sbox_pmt_3[state[1]] & 0x30) |
            (sbox_pmt_2[state[2]] & 0x0C) |
            (sbox_pmt_1[state[3]] & 0x03);
        cipher[3] =
            (sbox_pmt_0[state[4]] & 0xC0) |
            (sbox_pmt_3[state[5]] & 0x30) |
            (sbox_pmt_2[state[6]] & 0x0C) |
            (sbox_pmt_1[state[7]] & 0x03);

        cipher[4] =
            (sbox_pmt_1[state[0]] & 0xC0) |
            (sbox_pmt_0[state[1]] & 0x30) |
            (sbox_pmt_3[state[2]] & 0x0C) |
            (sbox_pmt_2[state[3]] & 0x03);
        cipher[5] =
            (sbox_pmt_1[state[4]] & 0xC0) |
            (sbox_pmt_0[state[5]] & 0x30) |
            (sbox_pmt_3[state[6]] & 0x0C) |
            (sbox_pmt_2[state[7]] & 0x03);

        cipher[6] =
            (sbox_pmt_2[state[0]] & 0xC0) |
            (sbox_pmt_1[state[1]] & 0x30) |
            (sbox_pmt_0[state[2]] & 0x0C) |
            (sbox_pmt_3[state[3]] & 0x03);
        cipher[7] =
            (sbox_pmt_2[state[4]] & 0xC0) |
            (sbox_pmt_1[state[5]] & 0x30) |
            (sbox_pmt_0[state[6]] & 0x0C) |
            (sbox_pmt_3[state[7]] & 0x03);

        round_key[5] ^= round_counter << 2; // do this first, which may be faster

        // use state[] for temporary storage
        state[2] = round_key[9];
        state[1] = round_key[8];
        state[0] = round_key[7];

        round_key[9] = round_key[6] << 5 | round_key[7] >> 3;
        round_key[8] = round_key[5] << 5 | round_key[6] >> 3;
        round_key[7] = round_key[4] << 5 | round_key[5] >> 3;
        round_key[6] = round_key[3] << 5 | round_key[4] >> 3;
        round_key[5] = round_key[2] << 5 | round_key[3] >> 3;
        round_key[4] = round_key[1] << 5 | round_key[2] >> 3;
        round_key[3] = round_key[0] << 5 | round_key[1] >> 3;
        round_key[2] = state[2] << 5 | round_key[0] >> 3;
        round_key[1] = state[1] << 5 | state[2] >> 3;
        round_key[0] = state[0] << 5 | state[1] >> 3;

        round_key[0] = (round_key[0] & 0x0F) | sbox[round_key[0] >> 4];

        print_cipher(cipher);

    }

    // if round is not equal to 31, then do not perform the last adding key operation
    // this can be used in constructing PRESENT based algorithm, such as MAC
    if (31 == rounds) {
        cipher[0] ^= round_key[0];
        cipher[1] ^= round_key[1];
        cipher[2] ^= round_key[2];
        cipher[3] ^= round_key[3];
        cipher[4] ^= round_key[4];
        cipher[5] ^= round_key[5];
        cipher[6] ^= round_key[6];
        cipher[7] ^= round_key[7];

        print_cipher(cipher);
    }
}

#endif /* PRESENT_H */
```

APPENDIX 2

Images of different confusion plots.

1. Confusion Plot between results round 1 and round 2 (fig. A2.1)

FIGURE A2.1 Round 1 and Round 2

2. Confusion Plot between results round 1 and round 15 (fig. A2.2)

FIGURE A2.2 Round 1 and Round 15

3. Confusion Plot between results round 1 and round 31 (fig. A2.3)

FIGURE A2.3 Round 1 and Round 31

4. Confusion Plot between results round 15 and round 31 (fig. A2.4)

FIGURE A2.4 Round 15 and Round 31

APPENDIX 3

Images of MATLAB codes.

1. MATLAB code for function getImage (fig. A3.1)

```
function A = getImage(b)
close all; %clc
Length = size(b,2);
n = round(int64(realsqrt(Length)));
for i = 1:n
    if (mod(Length,i) == 0)
        if (Length/i == i)
            row = i;
            column = i;
        else
            row = i;
            column = Length/i;
        end
    end
end
G = reshape(b.',row,column).';
C = jet(256);
L = size(C,1);
% Scale the matrix to the range of the map.
Gs = round(interp1(linspace(min(G(:)),max(G(:)),L),1:L,G));
A = reshape(C(Gs,:),[size(Gs) 3]); % Make RGB image from scaled.
image(A)
%imshow(A)
% imwrite(A,'SampleData\SampleText\text.jpeg');
```

FIGURE A3.1 getImage

2. MATLAB code for function wavelet transform (fig. A3.2)

```
clear all ; close all
x = imread('peppers.jpg');
%figure; imshow(x);

[xar,xhr,xvr,xdr] = dwt2(x(:,:,1),'haar');
[xag,xhg,xvg,xdg] = dwt2(x(:,:,2),'haar');
[xab,xhb,xvb,xdb] = dwt2(x(:,:,3),'haar');

xa(:,:,1) = xar ; xa(:,:,2) = xag ; xa(:,:,3) = xab ;
xh(:,:,1) = xhr ; xh(:,:,2) = xhg ; xh(:,:,3) = xhb ;
xv(:,:,1) = xvr ; xv(:,:,2) = xvg ; xv(:,:,3) = xvb ;
xd(:,:,1) = xdr ; xd(:,:,2) = xdg ; xd(:,:,3) = xdb ;

%figure, imshow(xa/255) ;
%figure, imshow(xh) ;
%figure, imshow(xv) ;
%figure, imshow(xd) ;

%X1 = [xa*0.003 log10(xv)*0.3 ; log10(xh)*0.3 log10(xd)*0.3] ;
X1 = [xa/255 log2(xv) ; log2(xh) log2(xd)] ;
figure ; imshow(X1)

[xaar,xhhr,xvvr,xddr] = dwt2(xa(:,:,1),'haar');
[xaag,xhhg,xvvg,xddg] = dwt2(xa(:,:,2),'haar');
[xaab,xhhb,xvvb,xddb] = dwt2(xa(:,:,3),'haar');
xaa(:,:,1) = xaar ; xaa(:,:,2) = xaag ; xaa(:,:,3) = xaab ;
xhh(:,:,1) = xhhr ; xhh(:,:,2) = xhhg ; xhh(:,:,3) = xhhb ;
xvv(:,:,1) = xvvr ; xaa(:,:,2) = xvvg ; xvv(:,:,3) = xvvb ;
xdd(:,:,1) = xddr ; xdd(:,:,2) = xddg ; xdd(:,:,3) = xddb ;
%figure,imshow(xaa/255);
%figure,imshow(xhh);
%figure,imshow(xvv);
%figure,imshow(xdd);
X11 = [ xaa*0.001 log10(xvv)*0.3 ; log10(xhh)*0.3 log10(xdd)*0.3 ] ;
%figure; imshow(X11)

[r,c,s] = size(xv);
%figure ; imshow([X11(1:r,1:c,:) xv*0.05 ; xh*0.05 xd*0.05 ])
```

FIGURE A3.2 Wavelet Transform

3. MATLAB code for function getSpectrum (fig. A3.3)

```
function getSpectrum(Array)
Size = size(Array,2);
winAvg = [];
% for winSize=0:(Size-1)
winList = round(linspace(0,(Size-1)));
for i = 1:100
    winSize = winList(i);
    freqList = [];
%     j = 0;
    for winIndex=1:(Size-winSize)
%         j = j+1;
        freqList(winIndex,:) = getFreq(Array(1,winIndex:(winIndex+winSize)));
    end
    winAvg(i,:) = getAvg(freqList);
end
Print(winAvg)
```

FIGURE A3.3 getSpectrum

4. MATLAB code for function DFT_image (fig. A3.4)

```
clear all; close all; clc

imdata = imread('hex_data\Data0000.jpeg');
figure(1);imshow(imdata); title('Original Image');imwrite(imdata,'hex_data\Original_Image.jpeg');

imdata = rgb2gray(imdata);
figure(2); imshow(imdata); title('Gray Image');imwrite(imdata,'hex_data\Gray_Image.jpeg');

%Get Fourier Transform of an image
F = fft2(imdata);
%Fourier transform of an image
s = abs(F);
figure(3);imshow(s,[]);title('Fourier Transform of an image');h = getframe; im1 = h.cdata;imwrite(im1,'hex_data\FTransform.jpeg');
%get the centered spectrum
Fsh = fftshift(F);
figure(4);imshow(abs(Fsh),[0 5000]);title('Centered Fourier Transform of an image');h = getframe; im1 = h.cdata;imwrite(im1,'hex_data\CFTransform.jpeg');

%apply log transform
S2 = log(1+abs(Fsh));
figure(5);imshow(S2,[]);title('Log Transformed image');h = getframe; im1 = h.cdata;imwrite(im1,'hex_data\LogTransform.jpeg');

%reconstruct the image

F = ifftshift(Fsh);
f = ifft2(F);
figure(6);imshow(f,[]);title('Reconstructed image');h = getframe; im1 = h.cdata;imwrite(im1,'hex_data\Reconstruct.jpeg');
```

FIGURE A3.4 DFT_image

5. Image of Hex Data (fig. A3.5)

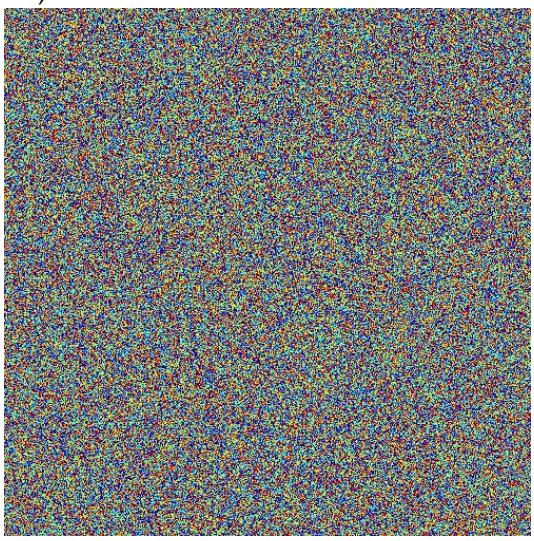


FIGURE A3.5 Hex_data (500X500)

6. Log transformed Image of Hex Data (fig. A3.6)

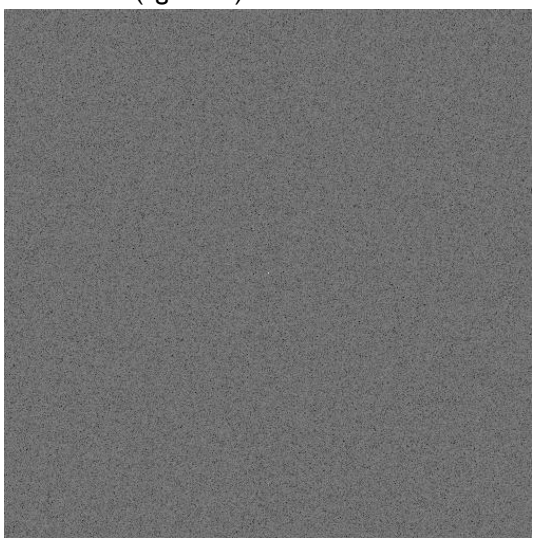


FIGURE A3.6 Log_Hex_data (500x500)

7. Enlarged Image of Text Data (fig. A3.7)

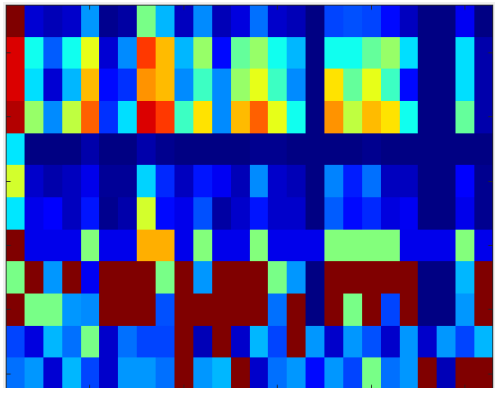


FIGURE A3.7 Text_data (28x28)

8. Enlarged Log Transformed Image of text Data (fig. A3.8)

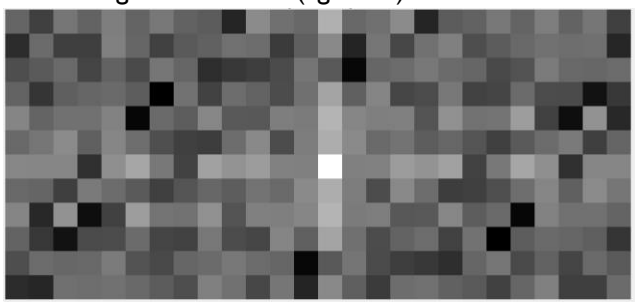


FIGURE A3.8 Log_Text_data (28x28)

9. Image of Text Spectrum (fig. A3.9)

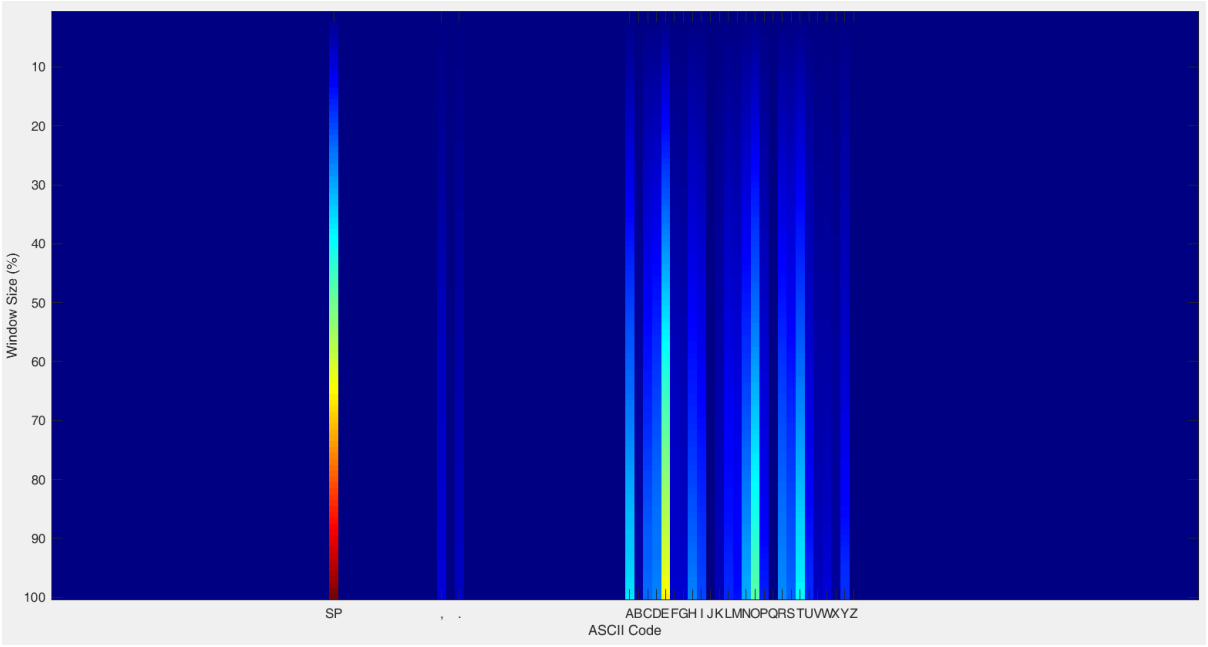


FIGURE A3.9 Text Spectrum

10. Log Transformed Image of Spectrum (fig. A3.10)

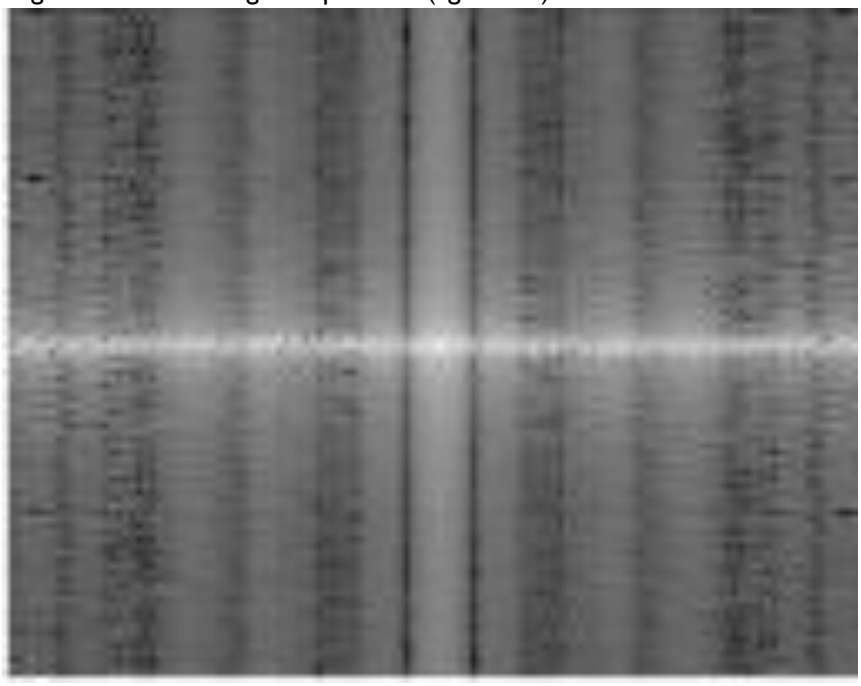


FIGURE A3.10 Log Text Spectrum

11. Random Image for Wavelet Transformation (fig. A3.11)

FIGURE A3.11 Peppers

12. Image Decomposition of Peppers (fig. A3.12)

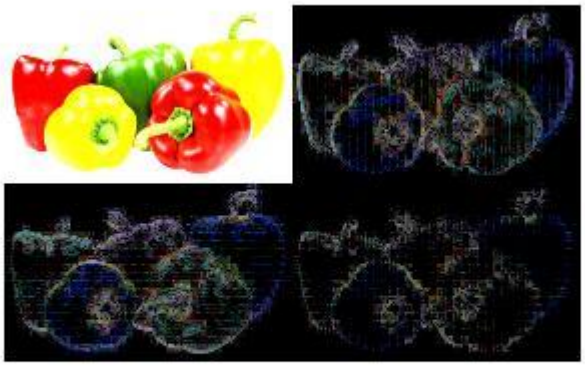


FIGURE A3.12 a) Approximate image (top left), b) Vertical image (top right), c) Horizontal image (bottom left), d) diagonal image (bottom right)