"A Comparative Analysis of Various Encryption & Decryption Techniques"

A

***Project Report** submitted in partial*
fulfillment of the requirements for the
award of the degree of

BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE & ENGINEERING

by

Name	**Roll No.**
Satvik Sharma	**R164216060**
Nitish Singh	**R164216042**
Nishant Singh	**R164216041**
Raghav Gupta	**R164216047**

under the guidance of

Mr. Ahatsham

School of Computer Science

Department of Systemics

University of Petroleum & Energy Studies

Bidholi, Via Prem Nagar, Dehradun, UK

December – 2018

CANDIDATE’S DECLARATION

We hereby certify that the project work entitled “A Comparative Analysis of Various Encryption and Decryption Techniques” in partial fulfilment of the requirements for the award of the Degree of BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE AND ENGINEERING with specialization in Internet Of Things and Smart Cities and submitted to the School of Computer Science, Department of Systemics, University of Petroleum & Energy Studies, Dehradun, is an authentic record of our work carried out during a period from August,2018 to December,2018 under the supervision of Mr. Ahatsham, Assistant Professor, School of Computer Science Department of Informatics.

The matter presented in this project has not been submitted by us for the award of any other degree of this or any other University.

Satvik Sharma R164216060	Nitish Singh R164216042
Nishant Singh R164216041	Raghav Gupta R164216047

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date: _____________2018

Mr.Ahatsham
Project Guide

Dr. Neelu Jyoti Ahuja
HOD - Systemics
School of Computer Science
University of Petroleum & Energy Studies
Dehradun – 248 001 (Uttarakhand)

ACKNOWLEDGEMENT

We wish to express our deep gratitude to our guide **Mr.Ahatsham**, for all advice, encouragement and constant support he has given us throughout our project work. This work would not have been possible without his support and valuable suggestions.

We sincerely thank to our respected HoD of the Department, **Dr. Neelu Jyoti Ahuja**, for his great support in doing our project in **Data Security** at **SoCS**.

We are also grateful to **Dr. Manish Prateek , Director** SoCS, UPES for giving us the necessary facilities to carry out our project work successfully.

We would like to thank all our **friends** for their help and constructive criticism during our project work. Finally we have no words to express our sincere gratitude to our **parents** who have shown us this world and for every support they have given us.

Satvik Sharma	**Nitish Singh**	**Nishant Singh**	**Raghav Gupta**
R164216060	**R164216042**	**R164216041**	**R164216047**

ABSTRACT

In the modern era, storage and transfer of data has become increasingly important. Data encryption is vastly used to ensure security everywhere. Security is an essential factor in every field such as Government Organization, Data Centers, E-commerce, Defense etc. where and so forth wherever internet is being utilized. Different types of data have their own distinct attributes so we need a diverge set of techniques to safeguard data from unapproved access. A system which anchor information on a set-up from an unapproved client is known as Cryptography. To safeguard data it is essential to know that which algorithm provides better security, efficiency and effectiveness. This project introduces analysis of various symmetric and asymmetric key encryption algorithms like RSA, AES, DES, SHA256, MD5 etc. based on different parameters. There is not a single tool available that provides us with the feature of encrypting the data based on different parameters (maximum speed, minimum vulnerabilities and maximum efficiency) so we are trying to make an effort to develop a platform where the user can encrypt the data as per their requirements.

Keyword: *Encryption, Cryptography, Symmetric, Asymmetric, RSA, AES, DES, SHA256, MD5*

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction:

Data that can be read and understood with common strategy is named as plaintext. The method of covering plaintext with aim to conceal its content is known as encryption. Encryption goal is to model a message incomprehensible to everyone, except to the intended recipient. Encryption is used to safeguard against the abuse of data. Cipher text is non-meaningful message which is produced after scrambling of plain message. Encryption is used to generate cipher text from plain text and to hide information from anyone the data is not intended for. Decryption is the art of converting the cipher text back to its original form. The discipline of understanding the math used to encrypt and decrypt data is known as cryptography. Cryptography encourages to gather delicate data and transfer it through the untrusted systems with a specific end goal to keep it mixed up from opening aside from the mean recipient.

Where cryptography is the art of anchoring information, cryptanalysis is the art of investigating and finding vulnerabilities in secured communication medium. A professional cryptanalyst uses analytical reasoning, many math tools, pattern finding techniques etc. to test and break an encryption techniques. Cryptanalysis and cryptography both fall under the paradigm cryptology. A mathematical function utilized for the process of encrypting and decrypting data is called a cryptographic algorithm, or cipher. A cryptographic calculation system leads with the mix of a key a word, number, or articulation to encode the plaintext. The indistinguishable plaintext scrambles to unique figure content with not at all like keys. The security of encoded information is totally dependent on two critical viewpoints i.e. the quality of the cryptographic calculation and the secrecy of the key. A cryptosystem is entitled due to the presence of a cryptographic algorithm, along with all potential keys and all the working protocols.

2. Problem Statement:

At present, there are many encryption and decryption algorithms being used in a variety of Applications but there is no definite tool through which a regular user can get an in-depth insight of different techniques about their encryption quality, computation time, memory used, efficiency etc.

This project aims to create a system in which a regular user can clearly visualize and differentiate between different techniques used in encryption & decryption of data and enable them to deploy these techniques according to one's own preferences.

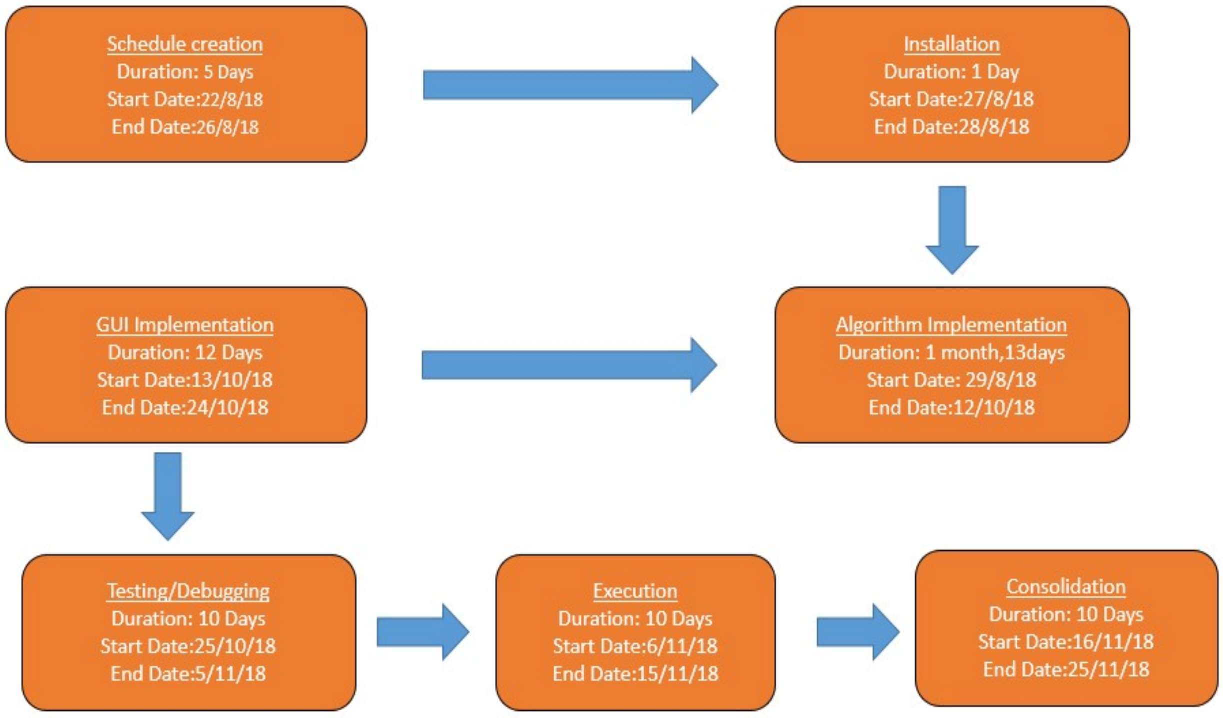
3. Objectives:

The two fundamental qualities that distinguish and separate one encryption algorithm calculation from another are its capacity to anchor the ensured information against assaults and its speed and proficiency in doing as such. This project provides a performance comparison between the commonly used different encryption & decryption algorithms.

The primary objectives are listed below-:

- To study about various encryption & decryption algorithms.
- To develop an efficient and productive tool for their comparative analysis.

4. Pert Chart:


Schedule creation
Duration: 5 Days
Start Date:22/8/18
End Date:26/8/18
Installation
Duration: 1 Day
Start Date:27/8/18
End Date:28/8/18
GUI Implementation
Duration: 12 Days
Start Date:13/10/18
End Date:24/10/18
Algorithm Implementation
Duration: 1 month,13days
Start Date: 29/8/18
End Date:12/10/18
Testing/Debugging
Duration: 10 Days
Start Date:25/10/18
End Date:5/11/18
Execution
Duration: 10 Days
Start Date:6/11/18
End Date:15/11/18
Consolidation
Duration: 10 Days
Start Date:16/11/18
End Date:25/11/18

5. Text Based Algorithm:

5.1 Twofish:

5.1.1 Introduction:

Twofish is a block cipher by Counterpane Labs, published in 1998.Twofish has a 128-bit block size, a key size ranging from 128 to 256 bits, and is optimized for 32-bit CPUs. Currently there is no successful cryptanalysis of Twofish. Twofish is an encryption algorithm based on an earlier algorithm Blowfish.

Block cycle principles:

- Block size
- Key size
- Number of rounds.
- Number of sub keys.
- Round function (logical function).
- Plain text in two equal halves.

5.1.2 Twofish Algorithm:

2.1 Encryption:

The encryption algorithm is as follows: it takes 128 bits (16 bytes) of input (the block size), and produces 128 bits (16 bytes) of output, using the subkeys generated from the key schedule, hence requiring a context pointer. The 16 bytes are then separated into 4 words. I have named this word array as oper. Just casting a word pointer type cast works out because the endianness in x86 is the same as required by the cipher (little-endian). The first step is that of input whitening. Each of the input words is XORed with the first four subkeys (0, 1, 2, 3). This is called the input whitening. Then 16 iterations (rounds) follow. In each round, a series of steps is performed. The graphical scheme explains this nicely. The first two words in oper, oper[0] and oper[1], are passed to the function F (will be described later), which outputs two words, which I've named f[0] and f[1] (array of two words). After this, oper[2] is XORed with f[0], then right rotated by one bit, while oper[3] is rotated left by one bit, then is XORed with f[1]. Then, oper[0] is swapped with oper[2], and oper[1] is swapped with oper[3]. We will refer to this step as the 'swap' step. After all the rounds have been completed, the last swap is undone. After that output whitening is performed. oper[0], oper[1], oper[2] and oper[3] are XORed with subkeys 4, 5, 6, 7, respectively.[1]

Two Fish has a feistel structure. The function F is responsible for a feistel structure. F is symmetrical (reversible) just like the XOR operation. The PHT is referred to as the Pseudo Hadamard Transformation, which is a pretty simple operation, as you can see. g is a specific version of the h function, which only takes one word (while the list denoted l is the S-Box from the context). It is defined as a macro.

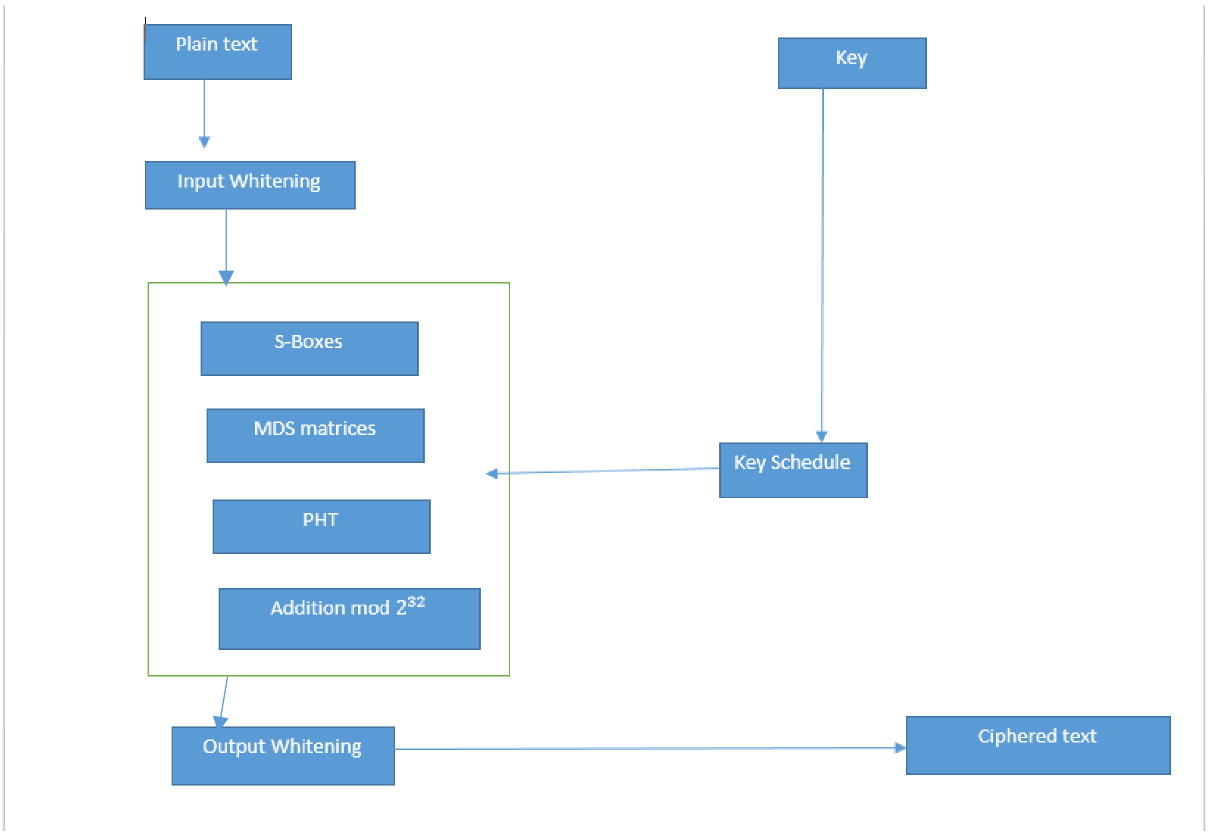


Figure 1.1

2.2 Decryption:

We begin by undoing the output whitening. Then we iterate from 15 through 0, it is because we need the rounds decrypted in the reverse order they were encrypted. For each round, we undo the swap we did in the encryption, then, because F is symmetric, we leave it as it is. After all the rounds have been reversed, we undo the input whitening, and we're done.[1]

5.1.3 Twofish Building Blocks:

3.1 Feistel Network:

Most of the block cipher techniques will follow Feistel structure. In the Feistel structure firstly the plain structure is divided into two equal halves namely left and right half, on the right half we have to apply a function. In functions we will use a separate key 1 and the output from this function will be XORed with the left half. The output received from the left half and the right half will be swapped i.e. the left half will be stored in rightmost bits and the right half will be stored in the leftmost bits (Figure 1.1). This is called single round. If any plain text follows this structure we call it as feistel structure. Number of rounds depends upon the algorithm used. Again function is applied on the right side and a separate key 2 is used, the result is XORed with the left side and the output bits are swapped. In feistel network the security mainly depends on the Round function, Number of rounds and the Number of Subkeys (If there are 10 rounds we need to generate 10 subkeys from the master key).

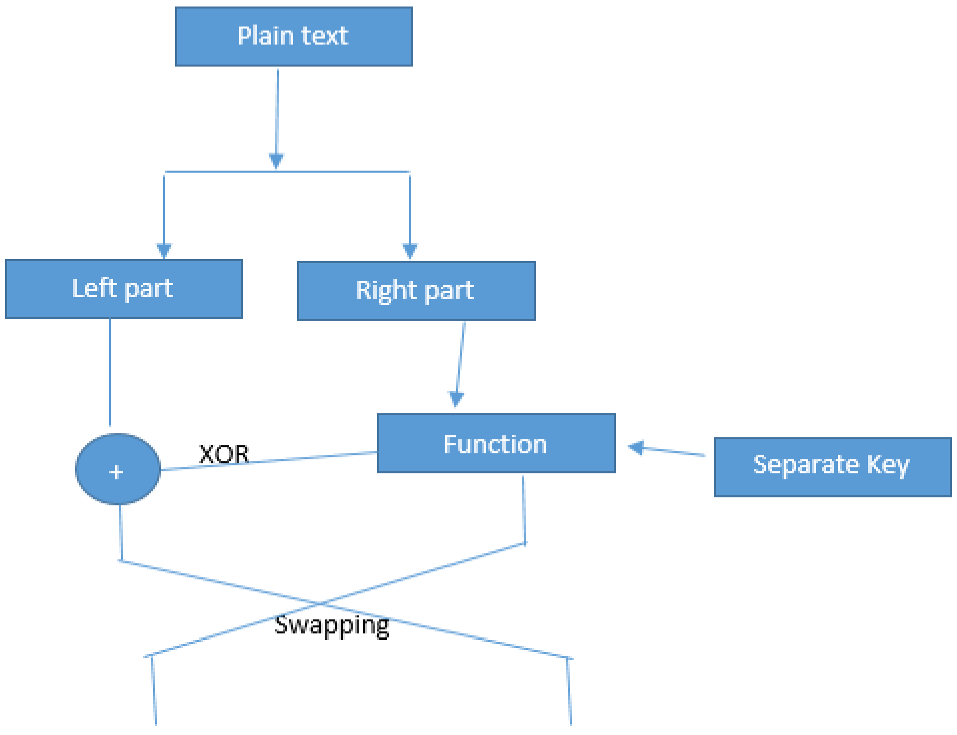


Figure 1.2 Round Function

3.2 **Pseudo-Hadamard Transforms:**

The pseudo- Hadamard transform is a reversible transformation of a bit string that provides cryptographic diffusion. H is a symmetric and non-sinusoidal function.

We can express the above transformation in matrix form as:

For 1D = H. f

For 2D = H. f. H^t

$$H = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

3.3 **GF Polynomial:**

Finite fields of order 2^m are called binary fields or characteristic-two finite fields. They are of special interest because they are particularly efficient for implementation in hardware, or on a binary computer. [2]

The elements of GF (2^m) are binary polynomials, i.e. polynomials whose coefficients are either 0 or 1. There are 2^m such polynomials in the field and the degree of each polynomial is no more than m-1. Therefore the elements can be represented as m-bit strings. Each bit in the bit string corresponding to the coefficient in the polynomial at the same position.

GF(2^8) polynomial is x^8+x^4+x^3+ x +1

a * b = 1 Mod P

Where a is input, b is GF(2^8)

3.4 **Reed Solmon Codec:**

An MDS matrix (Maximum Distance Separable) is a matrix representing a function with certain diffusion properties that have useful applications in cryptography

Reed-Solomon codes are block-based error correcting codes with a wide range of applications in digital communications and storage. Reed-Soloman Codes have the MDS property and are frequently used to obtain the MDS matrices used in cryptographic algorithms.

Encoding matrix * Original Data = Original Data + Parity

Example:

$$\begin{bmatrix} 01\ 00\ 00\ 00 \\ 00\ 01\ 00\ 00 \\ 00\ 00\ 01\ 00 \\ 00\ 00\ 00\ 01 \\ 1b\ 1c\ 12\ 14 \\ 1c\ 1b\ 14\ 12 \end{bmatrix} * \begin{bmatrix} A\ B\ C\ D \\ E\ F\ G\ H \\ I\ \ J\ K\ L \\ M\ N\ O\ P \end{bmatrix} = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \\ 51 & 52 & 53 & 49 \\ 55 & 56 & 57 & 25 \end{bmatrix}$$

5.1.4 Key scheduling:

The key schedule is performed when a context is allocated. It contains the key length in bytes, four key dependent S-Box words, and an array of 40 words for the subkeys generated by the key schedule.The key schedule begins by defining a variable k. k is assigned the value of the key length in bits divided by 64, but since we are operating in bytes, we use k = keylen / 8.

The key is 192 bits or 256 bits, respectively), memory for the context is passed to twofish_ctx_new as an array of keylen bytes. Then we define m as an array of keylen / 4 words (since one word is 4 bytes). The specification says that the words should be arranged in the little-endian convention, hence this will work only on architectures using the little-endian convention to represent words. This has been tested on x86. Endianess is not a big problem, and it can be very easily ported to architectures using the big-endian convention.

After checking that the key's length is 16 bytes, 24 bytes or 32 bytes (128 bits, allocated, then the key length is initialized. After this, the array of words m, representing the key, is split into two smaller arrays, m_even and m_odd, each containing words in m with even and odd indices, respectively. Since half of the indices are odd, and half are even (keylen IS divisible by two), we can say that the size of one of the arrays is (keylen / 4) / 2 = keylen / 8. But keylen / 8 is k. So we iterate upto K .What you see after the assigning of the even/odd splitting, is the matrix multiplication, to calculate the S-Box words. A predefined matrix RS (referred to as the Reed-Solomon code in the spec), is multiplied with a vector made up from the key bytes. RS is a 8x4 matrix, so the vector made up of the key bytes is of height 8 (and since it's a vector, it can be also referred to as a 1x8 matrix). The math works out, since k is defined as keylen / 8, and the vector contains 8 bytes, and there are k sbox-es, hence k vectors. For each i = 0 ... i = k - 1, eight bytes starting at the index 8 * i from the key, are defined as the vector to be multiplied with the RS matrix. The result is then stored in ctx->sbox[i]. The multiplication is performed over GF(2^8), so addition is XOR, and multiplication is performed through gf2_8_mul with a reduction polynomial of 0x014d, as specified by the spec.

Now that the S-Box words have been calculated, twofish_ctx_new only has to generate the 40 subkey words. Because there are 40 subkey words to be generated, and because in each two of them one is generated with m_even, and the other with m_odd, we only iterate up to 20, calculating two subkeys per iteration. There are operations in the following loop that we have not yet discussed. h is a function defined in the spec, which we will discuss right after this loop. RHO is defined as the word 0x01010101, which when multiplied by a byte, has the ability to make a word made of four identical bytes (named RHO because of the symbol in the spec). ROTL is a macro which performs a left bit rotation of a word. Besides these, the order of operations is identically as specified in the spec. [3]

The function h referenced in the code above, is defined in the spec as a function which takes as input a word x, a list of k words denoted l, and the variable k. Because we will later need to access the bytes of l individually, we define l as a byte array (which would be of length k * 4 = keylen / 2). The function is made of four steps, two of which depend on k. If k is 4 both of these steps will be executed. If k is 3 only one of these steps will be executed (the second step). And if k is 2, none of these steps will be executed. After the first three steps have been executed (or less than three depending on k), the resultant word is the matrix product over GF(2^8) of the predefined 4x4 MDS matrix (refered to as the Maximum Distance Separable code), with the bytes of the word y. The word y is first assigned the word x, and then it is modified by the first three steps (or less, depending on k). The reduction polynomial is 0x0169 for MDS.

The g_q is an array of two arrays of 256 bytes each. They are generated in twofish_init, to show the actual generation process, but if you're aiming for speed, you could hardcode the table.

For you who have also read my article on Rijndael, it is very similar to the code in my AES implementation, with the exception that the reduction polynomial is not global, but it's given as a parameter. This is because Twofish uses different reduction polynomials for MDS and RS. The algorithm of the multiplication is very straight-forward. It's just like you do multiplication by hand, except that the value of a bit is either one or zero, and you only multiply by one and zero. This is effectively implemented as an addition (XOR, because we are operating over GF(2^8)) if the bit is set, or do nothing, if the bit is 0. We also define a function for destroying the context. [1]

5.1.5 Flame Graph:

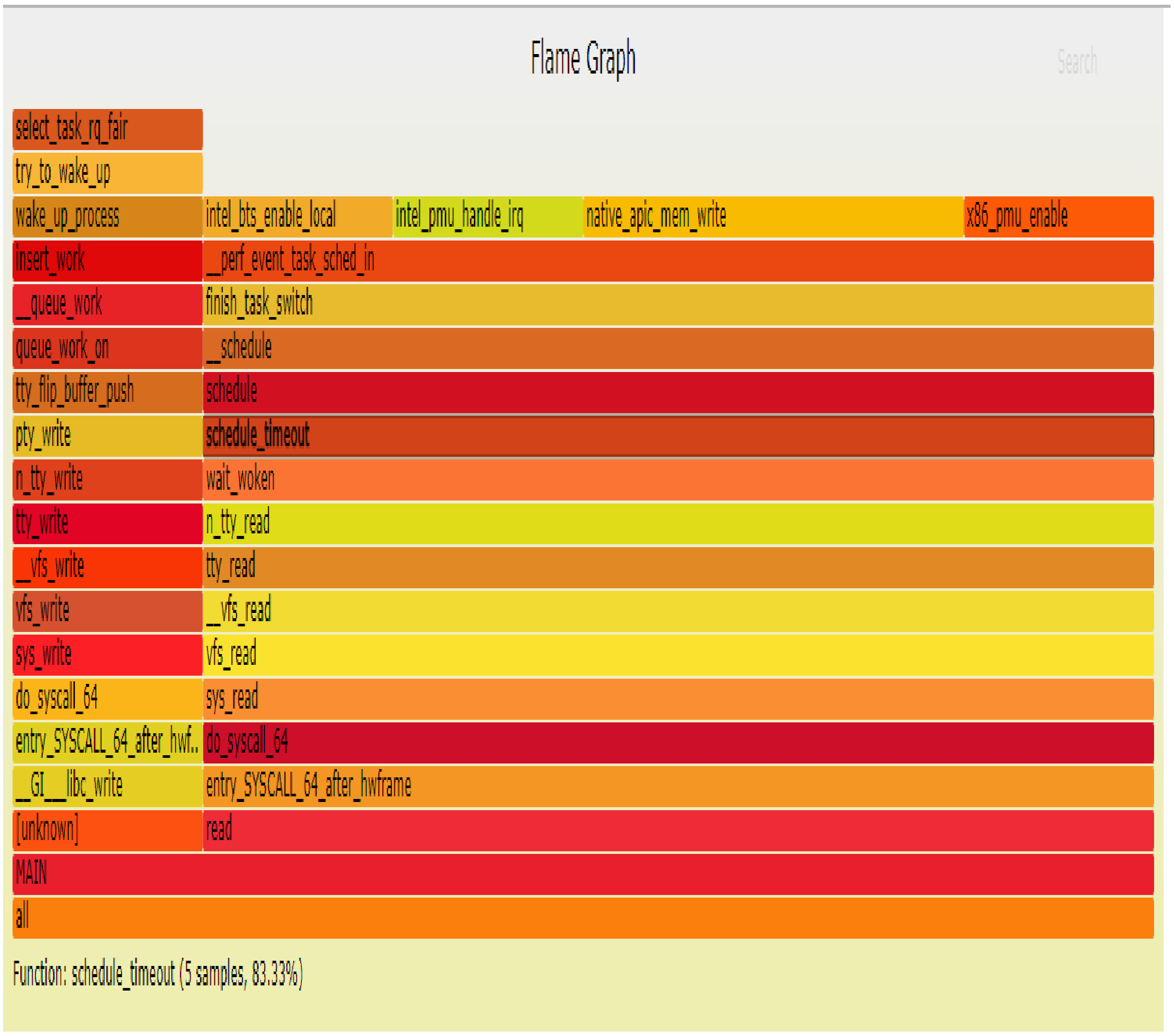


Figure 1.3

5.1.6 Code Path:

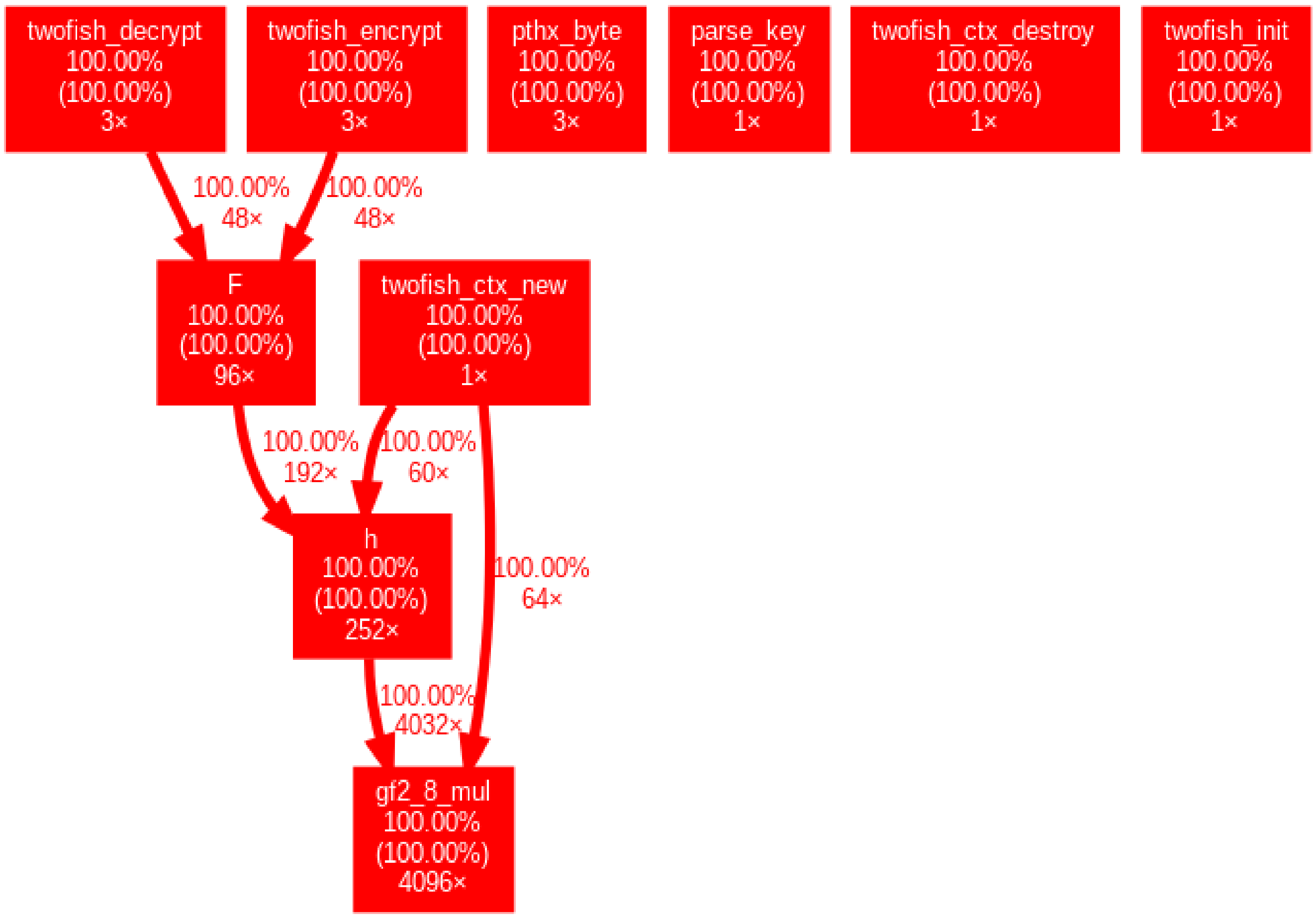


Figure 1.4

5.2 <u>RSA (Rivest Shamir Adleman) Algorithm</u>

5.2.1 Introduction:

The Rivest Shamir Adleman (RSA) algorithm is one of the most popular and secure public-key encryption methods [1]. It used by computer to encrypt and decrypt the messages. RSA uses public and private key, messages encrypted using the public key can only be decrypted by the private key. This algorithm is based on modular exponentiation [2]. The proposed RSA is used for system that need high securities with normal speed. As a future work multiple file encryption and decryption can be possible. The project application was designed to take the efficiency and reusability into account. Great level of securities is achieved using this algorithm.

RSA guarantees the origin of sent information, only the sender with his own private key is able to encrypt the message therefore transform the message into unreadable form consequently the receiver will have the confirmation of the origin because he will be able to decrypt the message only through the corresponding public key. The sender cannot state that the message has not been encrypted with the private key because the private key used for the encryption is unique and it's the owner's responsibility to make sure that it is not used by non-authorized third parties. [1]

5.2.2 RSA algorithm:-

- In order to be algorithm very secure we have to consider two large prime numbers p and q, if we take small numbers than it will not be secure as it should.
- After consider p and q calculate n and calculate Euler's totient function

$$n=p*q$$
$$\phi(n) = (p-1)(q-1)$$

- Select a small odd integer e which is relatively prime to $\phi(n)$, where $\phi(n)=(p-1)*(q-1)$.
- Calculate the value of d(explained below). D is calculated as modular multiplicative inverse of e modulo n.
- Publish the pair P=(e,n) as a public key.
- Keep secret the pair S=(d,n) as a private key.

The message is encrypted using P=(e,n) using following formula.

$$P(M) = M^{e} \bmod n$$

The message can be decrypted using P=(d,n) using following formula.

$$S(C) = C^d \bmod n$$

Secret key pair (d, n) should be kept secret.

To calculate the value of d, we use the various number theories from mathematics. We Select e which is relatively prime to $\phi(n)$. This mean the GCD of e and $\phi(n)$ is always 1. i.e.

$$\text{GCD}(e,\phi(n))=1$$

Let x and y be two parameters such that they satisfy following mathematical expression,

$$\text{GCD}(a,b)=d=ax+by$$

If we replace value of a and b with e and $\phi(n)$ respectively, we get

$$\text{GCD}(e,\phi(n))=1=ex+\phi(n)y$$

We already know the value of e and ϕ and now we can use the Extended Euclid's Algorithm to get the value of x and y. We can re-arrange the above expression as below

$$ex+\phi(n)y=1$$

Taking (mod $\phi(n)$) on both side

$$ex+\phi(n)y\equiv 1(\bmod \phi(n))$$

Since $\phi(n)y(\bmod \phi(n))$ is 0, the final expression becomes

$$ex\equiv 1(\bmod \phi(n))$$

X is now called a modular multiplicative inverse of e.

We can finally solve for d using a Modular linear Equation Solver which uses Extended Euclid's Algorithm.

After we calculate the value of d, next step is to use public key pair (e,n) to encrypt the message using

$$P(M) = M^e \bmod n$$

Where M is an important message to encrypt. Even for a large value of e, (Me mod n) can be calculated using a modular exponentiation in very less amount of time. Similarly using the private key pair (d,n), we can decrypt the message using

$$S(C) = C^d \bmod n$$

Where C is the encrypted message and S(C) is the original message M. [3]

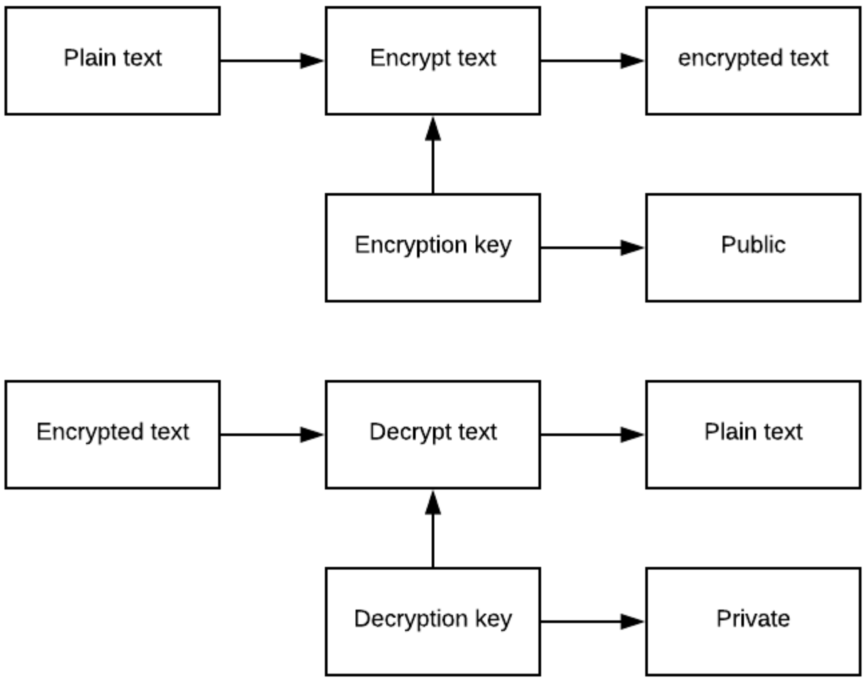


Figure 2.1

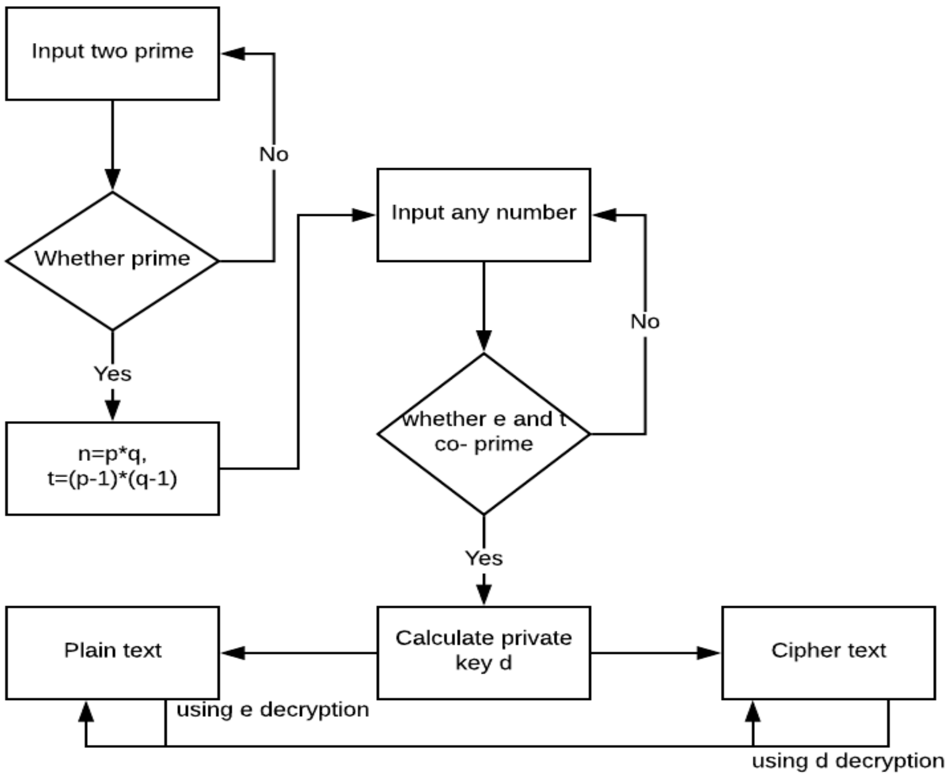


Figure 2.2

5.2.3 CONCLUSION:-

Since RSA proposed from the nineteen seventies, through the practice of 20 years, has been widely used, has become a most popular encryption standard. With the development of computer network and e-commerce technology, provides a broader space for the application of RSA technology. In many hardware, RSA software and library software product kernel, is convenient for people to use. [4]

In this study, based on fully research and a deep understanding of the principle of tradition RSA algorithm, the RSA algorithm is implemented in VC environment and analyzes the security of RSA algorithm and its disadvantages. On the whole, the RSA algorithm is a good algorithm.

5.2.4 Flame Graph:

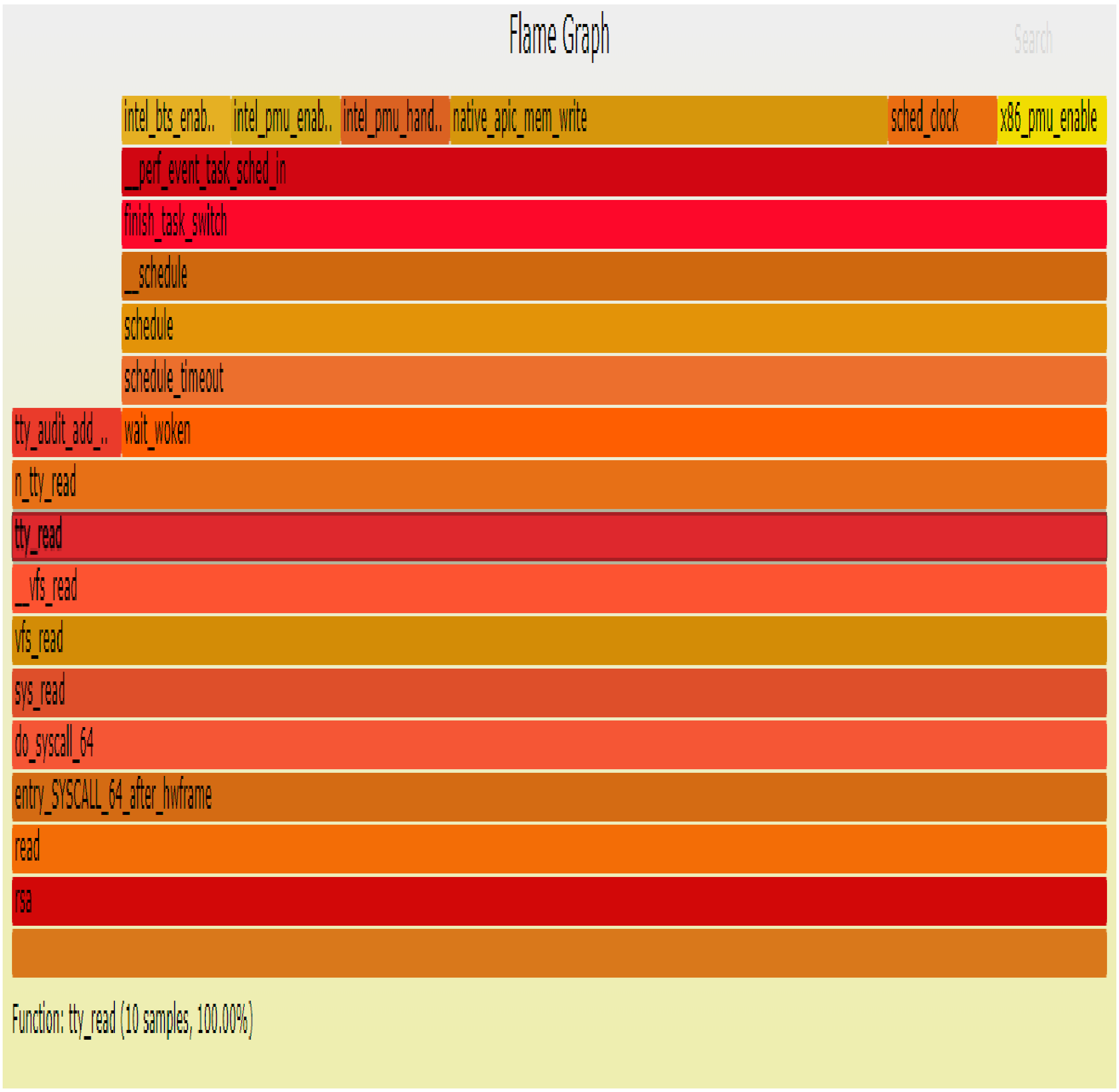


Figure 2.3

5.2.5 Code Path:

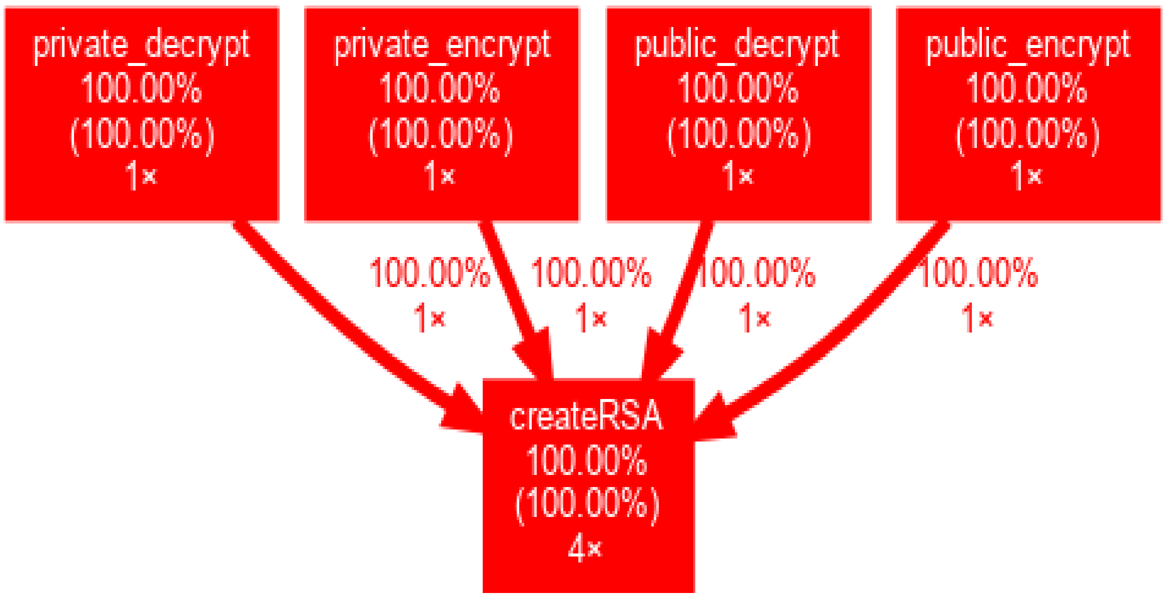


Figure 2.4

5.3 Comparative Analysis between Image based Algorithms:

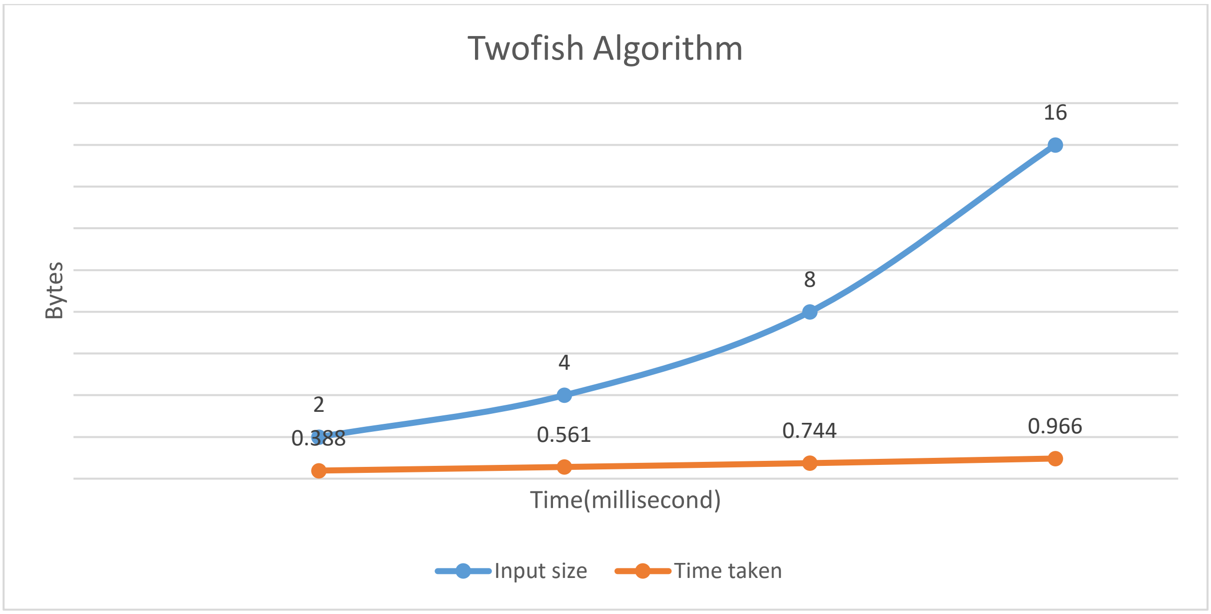


Figure 3.1

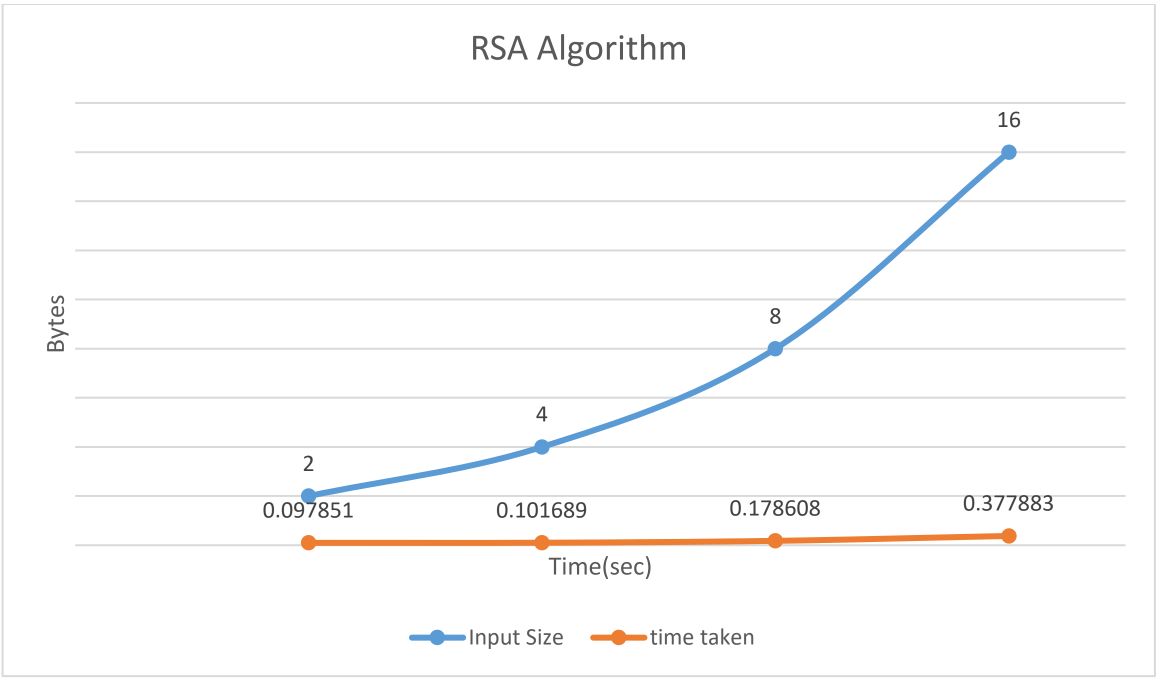
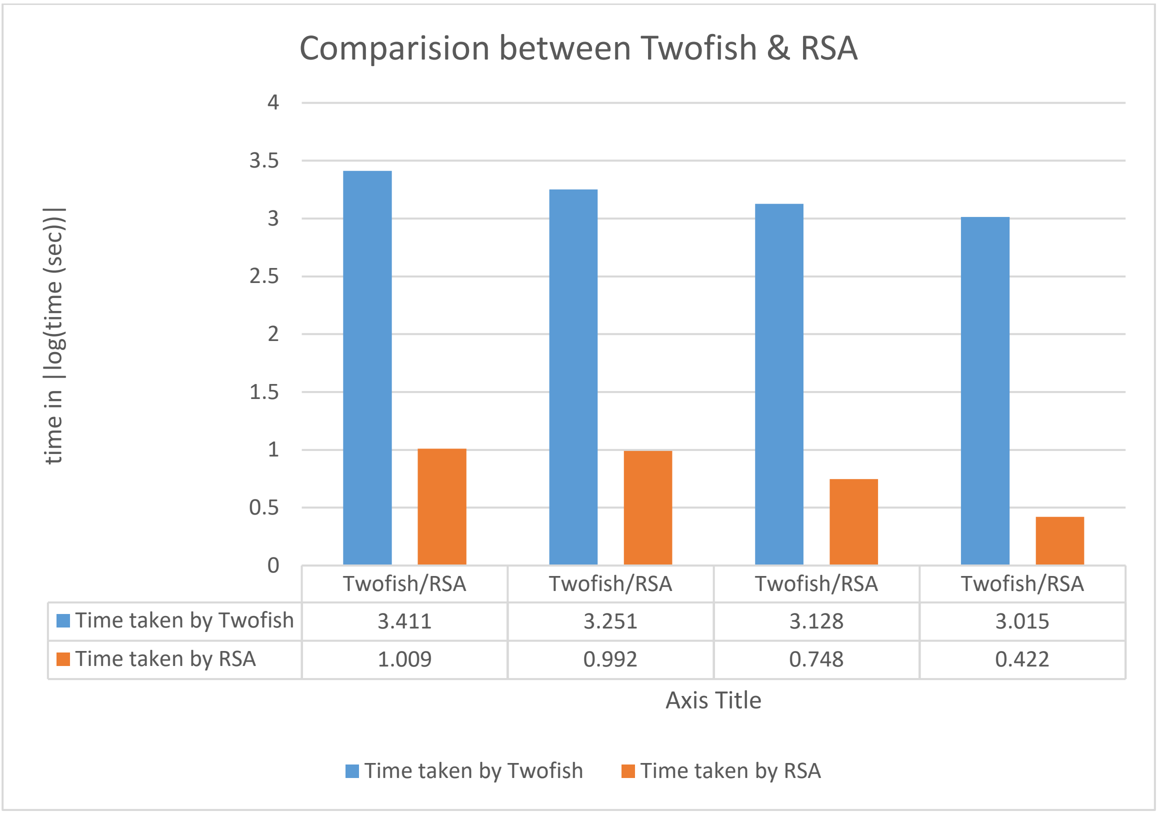


Figure 3.2



	Twofish/RSA	Twofish/RSA	Twofish/RSA	Twofish/RSA
🟦 Time taken by Twofish	3.411	3.251	3.128	3.015
🟧 Time taken by RSA	1.009	0.992	0.748	0.422

Figure 3.3

6. Digital Signature Based Algorithm:

6.1 MD5:

6.1.1 Introduction:

MD5 stands for message-digest algorithm. It takes as input a message of arbitrary length and then produces as output a 128-bit "message digest" of the input. It ensures that messages integrity is correct and it is safe to use. MD5 is a one-way hash algorithm that addresses two main concerns that are created when communicating over a network: authenticity and data integrity. The MD5 algorithm is basically used for digital signature applications, where a large file must be "compressed" in a very safe and secured manner before being encrypted.

So the MD5 algorithm is designed so that it could be quite fast on 32-bit machines. The MD5 algorithm is an upgradation of the MD4 message-digest algorithm. Since it is an up graded version of MD4 Algorithm but it is bit slower than MD4 algorithm but far more secured than MD4. As MD4 was made to be very fast but because of that it had to risk its document and files as because of its fast speed its lacks then security criteria so to overcome this problem MD5 was designed for better security reasons. MD5 is fast and simple, yet offers a higher level of security than MD4 and is much more reliable than a checksum

The MD5 message-digest algorithm is commonly used HASH function which is meant to produce 128 bit hash value. . Although MD5 turned into to start with designed for use as a cryptographic hash function, it's been observed to be afflicted by considerable vulnerabilities. It can nonetheless be used as a checksum to verify data integrity, but best against unintentional corruption.

6.1.2 MD5 Algorithm

MD5 algorithm uses four rounds, each applying one of four non-linear functions to each sixteen 32-bit segments of a 512-bit block source text. The result is a 128-bit digest. Figure 1 is a graph representation that illustrates the structure of the MD5 algorithm. MD5 algorithm takes a b-bit message as input, where b is an arbitrary nonnegative integer. The following five steps are performed in C programming language to compute the message digest of the input message.

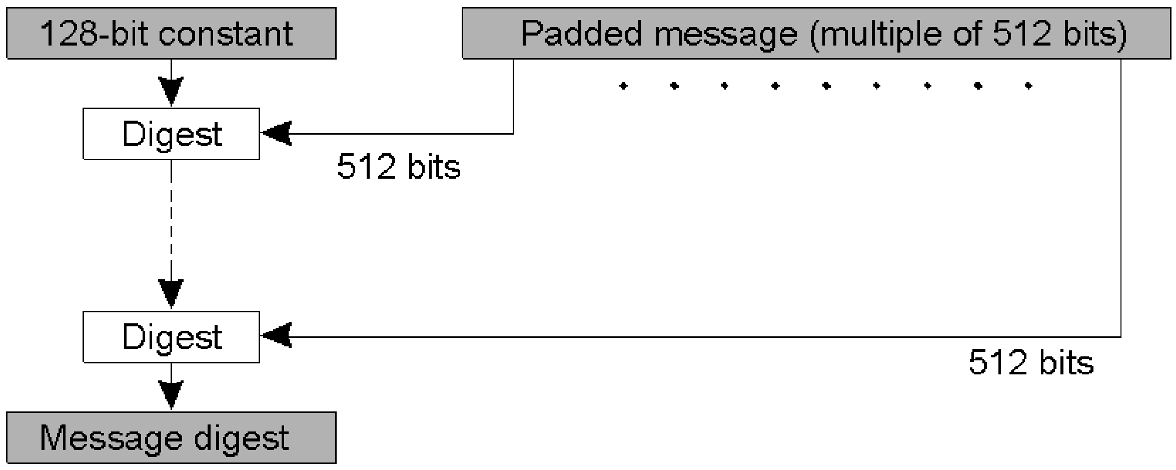


Fig 4.1 The structure of MD5 algorithm

2.1 Example on how MD5 runs

Suppose we send a file over the internet so how could we make sure that in between the process of sending and receiving that is in the middle of the process that some hacker has intercepted and modified that file and send it to the receiver or let's say if I have created a software and shared a download link on my website so if a hacker hacks my website and put his file in place of my download link so how the third person who is downloading the file from the website will ensure that he has downloaded the actual file or not So for that we create MD5 checksum for our file (MD5 is just a string which we generate based on our message) so what the algorithm does is it reads every single bit of the message and does some operations on the string which leads to continuous iteration on that string for each character.

So after the string is modified and final string is made which is unique for my message and now if someone changes the byte or character and if the user gets the message in that case it can easily find that it is not similar MD5 sum for that particular message while matching the strings.

So in this process what we do is we tell that this is my file and this is my MD5 checksum so if you download the file, then in your command line there is a MD5 sum command and when you give MD5 sum and file name then it calculate the MD5 and display it to you and you can just match that MD5 sum on the website and if it get matched then it ensures that it is the original file and no one has tempered it and it ensures that message integrity is correct.

We also have an example of communication : At some time during communication when it is going over the network and at the same time there could an error induced that is these wires goes through public platform and different places which leads to the possibility that because of the surrounding sound message could get tempered as there was too much noise in the surroundings and the message bit we are sending in that case some of the bits may get corrupt and change and file may get damaged and in that case when we run that MD5 sum we get failure reason being someone has disturbed it or the file is corrupt. That is what MD5 sum is meant for, it is not meant for encrypting the file or message It is an unique identifier of my message that whenever we apply MD5 algorithm on it, it will give you the same string that is digesting the file.

6.1.3 WORKING

1. Used to find the message digest
2.

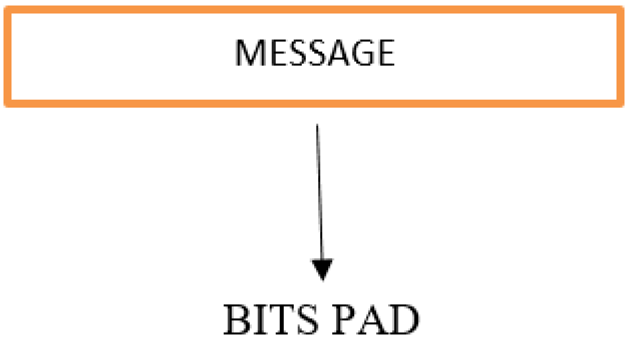


Fig 4.2

3. Make it multiple of 512 bit
4. And after it comes to multiple of 512 bits then we divide each of its message in 512 bit block.

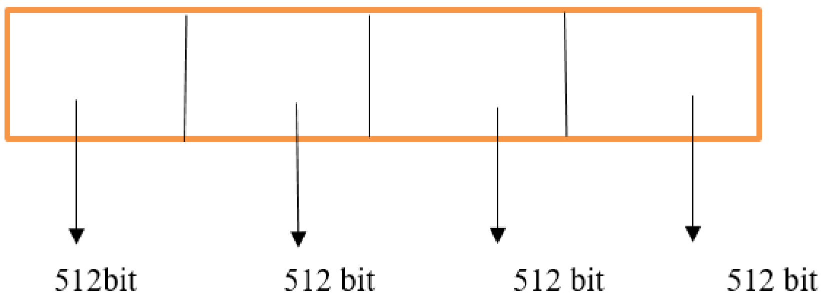


Fig 4.3

5. Then in next step, in each 512 bit we perform 4 round of operations and each round consist of 16 operation which implies a total of 64 operations being performed
6. After performing 64 bit operations the output of the operation we get we Fed that output in next 512 bit block
7. Similarly for all 512 bit block we perform same operations that is feeding the output values in the next block (previous block output $\xrightarrow{\text{FED}}$ as input in next block.)
8. And the last block output will be the Message digest.

3.1 ALGORITHM

Step 1. Append padding bits.

Step 2. Append length bits.

Step 3. Initialize the MD buffer.

Step 4. Process each 512 bit block.

Step 5. Output message digest.

Step 1. Append padding bits- The input message is "padded" (extended) so that its length (in bits) equals to 448 mod 512. That is, the message is extended so that it is just 64 bits shy of being a couple of 512 bits lengthy. Padding is continually executed, although the duration of the message is already congruent to 448, modulo 512. Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448 mod 512. In all, at least one bit and at most 512 bits are appended.

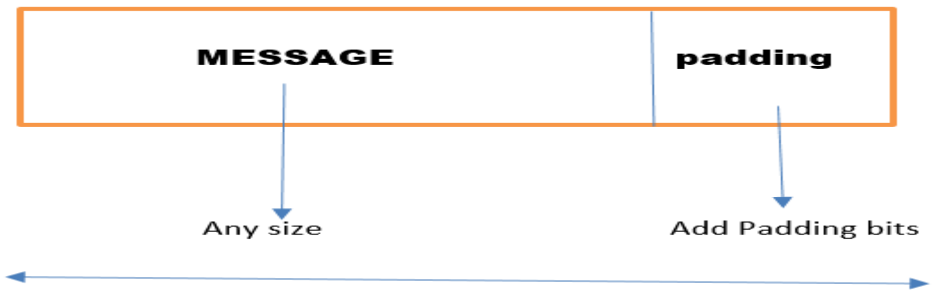


Fig 4.4

Step 2. Append Length- Here we have a message where we already have added padding bits, here we add length bit

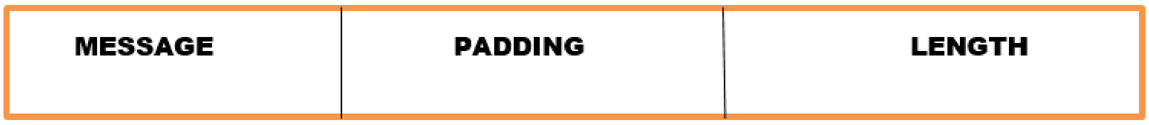


Fig 3.4

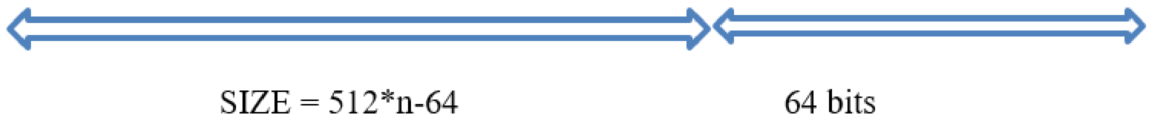


Fig 4.5

(To convert it into 512 multiple bit our length should be of 64 bit)

TOTAL SIZE OF THE MESSAGE i.e. message + padding + length = 512*n

The size of the message bits will be added. EX: If the message size is 2^8 so we add 8 bit in length or for 2^64 we add 64 bit in length. And if it has more than 2^64 we take MOD of 2^64.

Explanation

A 64-bit illustration of b (the length of the message before the padding bits were delivered) is appended to the end result of the preceding step. If b is greater than 2^64, then only the low-order 64 bits of b are used. (These bits are affixed as two 32-bit words and affixed low-arrange word first as per the previous conventions. The resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits. The input message will have a length that is an exact multiple of 16 (32-bit) words. Let M [0 ... N-1] denote the words of the resulting message, where N is a multiple of 16.

Step 3. Initialize MD Buffer

A four-word buffer (A,B,C,D) is used to compute the message digest. Here each of A, B, C, D is a 32-bit register. These registers are Initialized to the following values in hexadecimal, low-order bytes

First:

word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10

Using these values we initialize the buffer

Since there are 64 operations to be performed in 4 rounds and each round has 16 operations for that we use one function.

512 bit block
64 operations
4 Rounds

Fig 4.6

Step 4. Process Message in 16-Word Blocks

We first define four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

F (B, C, D) = BC or not (B) D

G (B, C, D) = BD or C not (D)

H (B, C, D) = B xor C xor D

I (B, C, D) = C xor (B or not (D))

The four buffers (A, B, C and D) messages (content) are joined now with the input words, using the four auxiliary functions (F, G, H and I).4 rounds are performed and each involves 16 basic operations. The Processing block P is applied to the four buffers (A, B, C and D), by using message word M[i] and constant K[i]. The item "<<<s" denotes a binary left shift by s bits. The four type of IRF (info related functions) that each take as input three 32-bit words and produce same bits of output i.e. 32-bit word. They apply the logical operators ^, v, ! and xor to the input bits.

Explanation of 4 Rounds
Round 1 – 16 operation use function 'F'
Round 2- 16 operation use function 'G'
Round 3- 16 operation use function 'H'
Round 4- 16 operation use function 'I'

TOTAL 64 OPERATIONS
That is starting 16 bit uses 'F' function next 16 uses 'G' function next 16 uses 'H' function and last 16 uses 'I' function.

These 4 functions F,G,H,I use (B,C,D) as a input and these B,C,D buffer are of 32 bit each and output of all will be in form of 32 bit. This is a single operation and for each block there will be 64 operations which will be performed in 4 rounds and we use 1 function for each round to process each 512 bit block.

In each bit position, F acts as a condition such that if B then C else D. The function F could have been defined using “addition” instead of “or” since BC and not (B) D will never have 1's in the same bit position.

The functions G, H, and I are similar to the function F, which performs in "bit-wise parallel" to produce its output from the bits of B, C, and D so that if the corresponding bits of B, C, and D are independent and unbiased. Therefore, each bit of G (B, C, D), H (B, C, D), and I (B, C, D) will be independent and unbiased.

This step uses a 64-element table T [1 ... 64] constructed from the sine function. Let T [i] denote the i-th element of the table, which is equal to the integer part of 4294967296 times abs (sin (i)), where i is in radians. Then, performs the 4 rounds of hashing for each 16-word block.

Figure 3.6 Explains the process of each 512 bit block to generate the output-:

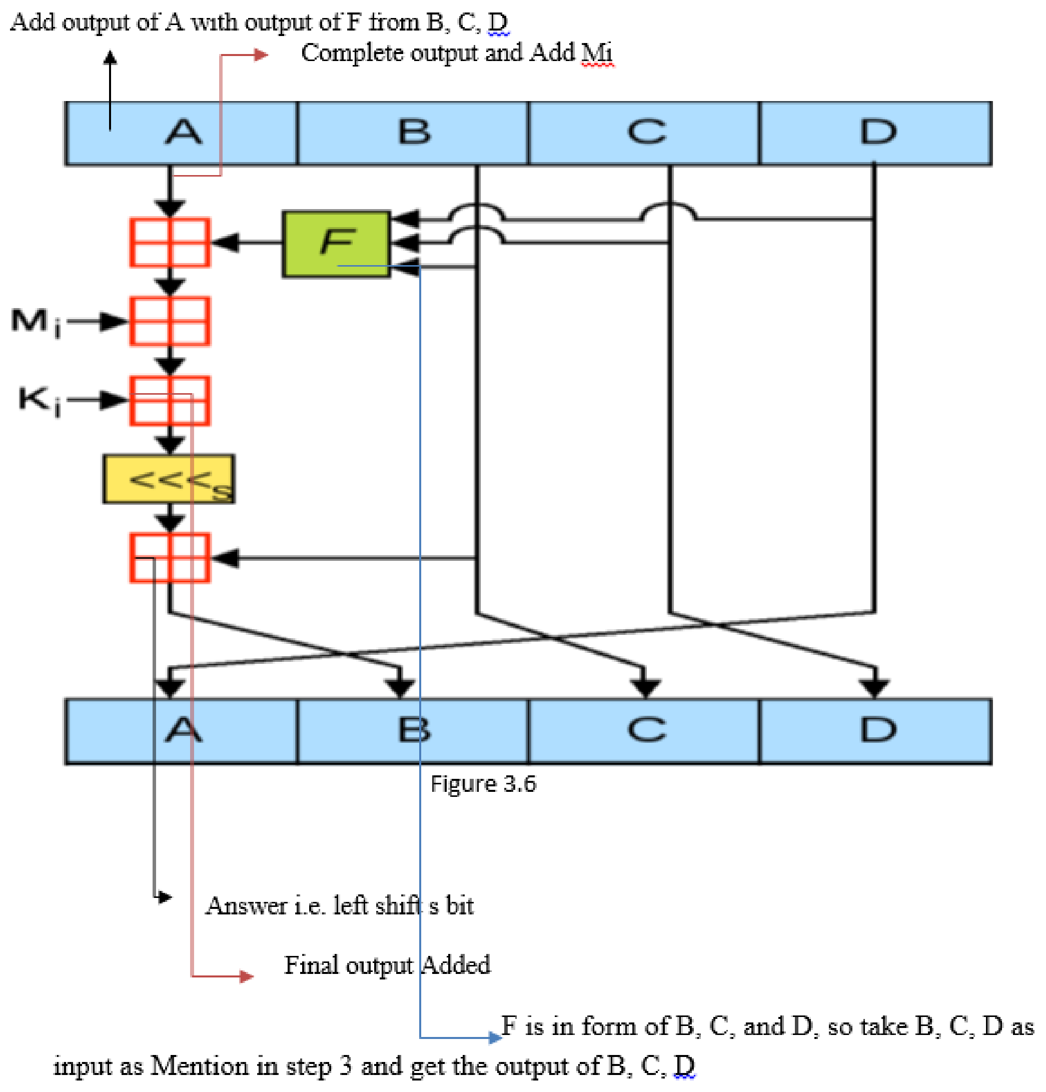

Add output of A with output of F from B, C, D
Complete output and Add Mi
A
B
C
D
F
M i
K i
<<< s
A
B
C
D
Figure 3.6
Answer i.e. left shift s bit
Final output Added
F is in form of B, C, and D, so take B, C, D as input as Mention in step 3 and get the output of B, C, D

Figure 4.7

In this fig 3.6 there is a single operation on 'F'. MD5 consist of 64 bit of these operations grouped in 4 round of 16 operations. 'F' is nonlinear function; one function is used in each round.M i denotes a 32 bit block of message input. K i denotes 32 bit constant, different for each operation. <<<s Left shift by s i.e. Left bit rotation by s place where s varies for each operation and

Denotes Addition of modulo 2^32

T igure 3.6 shows operation on 'F'. Similarly we do next operation for 'G', 'H', 'I' then total each 64 bit operation will be performed with total of 512 bit block will be processed and then output of A,B,C,D will be fed in the next 512 bit block and will continue till last 512 bit block . And the last A,B, C, D output will be our Message Digest (total of 128 bit message digest form 32*4 each 32 bit block).

6. Operation to be performed for implementation

[abcd k s i] denote the operation $a = b + ((a + F (b, c, d) + X [k] + T [i]) <<< s)$.

Do following 16 operation –

[ABCD 0 7 1]	[DABC 1 12 2]	[CDAB 2 17 3]	[BCDA 3 22 4]
[ABCD 4 7 5]	[DABC 5 12 6]	[CDAB 6 17 7]	[BCDA 7 22 8]
[ABCD 8 7 9]	[DABC 9 12 10]	[CDAB 10 17 11]	[BCDA 11 22 12]
[ABCD 12 7 13]	[DABC 13 12 14]	[CDAB 14 17 15]	[BCDA 15 22 16]

6.1.4 Conclusion

The MD5 message-digest algorithm is an extension of the MD4 message-digest algorithm, which changed into evolved by using Rivest in 1990. MD5 is slightly slower than MD4, but it is greater secure. Comparing to other digest algorithms, MD5 is straightforward to implement, and provides a "fingerprint" or message digest of a message of arbitrary length. It performs very fast on 32-bit machine. It is conjectured that the difficulty of arising with two messages having the equal message digest is at the order of 2^64 operations, and that the difficulty of developing with any message having a given message digest is at the order of 2^128 operations.

The MD5 message-process calculation is easy to execute, and gives a "unique finger impression" or message process of a message of arbitrary length. It is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of 2^64 operations, and that the difficulty of coming up with any message having a given message digest is on the order of 2^128 operations. The MD5 calculation has been deliberately examined for shortcomings. It is, however, a relatively new algorithm and further security analysis is of course justified, as is the case with any new proposal of this sort.

6.1.5 Flame Graph:

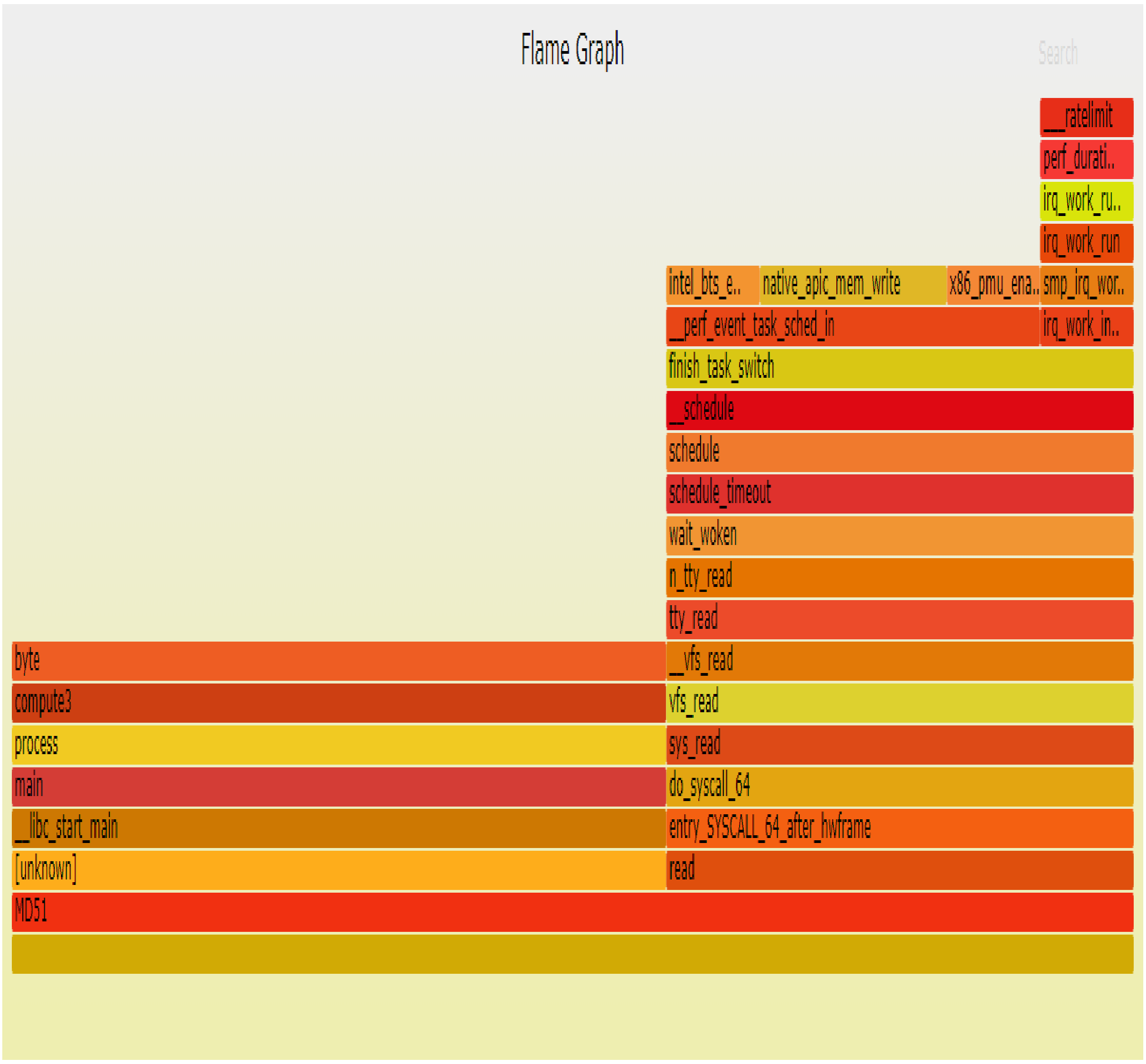


Figure 4.8

6.1.6 Code Path:

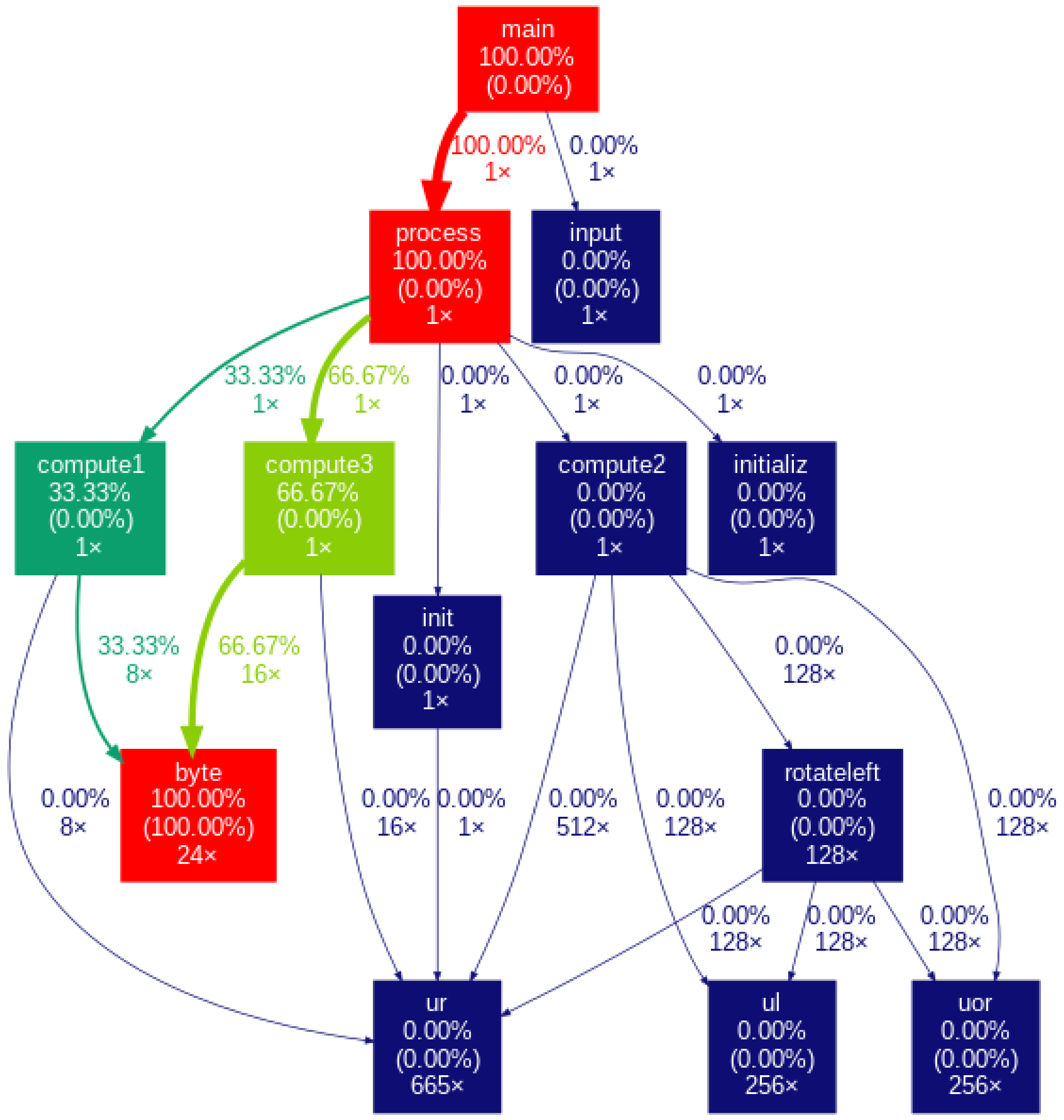


Figure 4.9

6.2 SHA256:

6.2.1 Introduction:

SHA-256 (secure hash algorithm, FIPS 182-2) is a cryptographic hash function with digest length of 256 bits. It is a keyless hash function; that is, an MDC (Manipulation Detection Code). Where, a message processed by block of $512=16 \times 32$ bit, each blocks requiring 64 rounds. SHA-256 used to hash a message, M, having length L bits, where $0 \leq L < 2^{64}$. The algorithm uses 1) a message schedule of sixty-four 32-bit words, 2) eight working variables of 32bits each, and 3) a hash value of eight 32-bit words. The final result of SHA-256 is a 256-bit message digest.

6.2.2 Overview:

SHA-256 operates in the manner of MD4, MD5, and SHA-1: The message to be hashed is first

1. padded with its length in such a way that the result is a multiple of 512 bits long, and then
2. parsed into 512-bit message blocks $M^{(1)}, M^{(2)},\ldots\ldots\ldots, M^{(N)}$.

The message blocks are processed one at a time: Beginning with a fixed initial hash value $H^{(0)}$, sequentially compute

$$H^{(i)} = H^{(i-1)} + C_{M}{}^{(i)}(H^{(i-1)}),$$

where C is the SHA-256 compressionfunction and + means word-wise mod 2^{32} addition. $H^{(N)}$ is the hash of M.

6.2.3 Basic operations

- Boolean operations AND, XOR and OR, denoted by $\wedge$, $\oplus$ and $\vee$, respectively
- Bitwise complement, denoted by $\overline{}$.
- Integer addition modulo 232, denoted by A +B.

Each of them operates on 32-bit words. For the last operation, binary words are interpreted as integers written in base 2.

- RotR(A,n) denotes the circular right shift of n bits of the binary word A.
- ShR(A,n) denotes the right shift of n bits of the binary word A.
- A‖B denotes the concatenation of the binary words A and B.

3.1 Functions and constants

- First, eight variables are set to their initial values, given by the first 32 bits of the fractional part of the square roots of the first 8 prime numbers:

$H_1^{(0)}$ = 0x6a09e667, $H_2^{(0)}$ = 0xbb67ae85, $H_3^{(0)}$ = 0x3c6ef372, $H_4^{(0)}$ = 0xa54ff53a,
$H_5^{(0)}$ = 0x510e527f, $H_6^{(0)}$ = 0x9b05688c, $H_7^{(0)}$ = 0x1f83d9ab, $H_8^{(0)}$ = 0x5be0cd19

- The algorithm uses the functions:

$$\mathrm{Ch}(X,Y,Z) = (X \wedge Y) \oplus (X \wedge Z),$$
$$\mathrm{Maj}(X,Y,Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z),$$
$$\Sigma 0(X) = \mathrm{RotR}(X,2) \oplus \mathrm{RotR}(X,13) \oplus \mathrm{RotR}(X,22),$$
$$\Sigma 1(X) = \mathrm{RotR}(X,6) \oplus \mathrm{RotR}(X,11) \oplus \mathrm{RotR}(X,25),$$
$$\sigma 0(X) = \mathrm{RotR}(X,7) \oplus \mathrm{RotR}(X,18) \oplus \mathrm{ShR}(X,3),$$
$$\sigma 1(X) = \mathrm{RotR}(X,17) \oplus \mathrm{RotR}(X,19) \oplus \mathrm{ShR}(X,10),$$

- The 64 binary words K_j given by the 32 first bits of the fractional parts of the cube roots of the first 64 prime numbers:

0x428a2f98 0x71374491 0xb5c0fbcf 0xe9b5dba5 0x3956c25b 0x59f111f1
0x923f82a4 0xab1c5ed5 0xd807aa98 0x12835b01 0x243185be 0x550c7dc3
0x72be5d74 0x80deb1fe 0x9bdc06a7 0xc19bf174 0xe49b69c1 0xefbe4786
0x0fc19dc6 0x240ca1cc 0x2de92c6f 0x4a7484aa 0x5cb0a9dc 0x76f988da 0x983e5152
0xa831c66d 0xb00327c8 0xbf597fc7 0xc6e00bf3 0xd5a79147 0x06ca6351
0x14292967 0x27b70a85 0x2e1b2138 0x4d2c6dfc 0x53380d13 0x650a7354
0x766a0abb 0x81c2c92e 0x92722c85 0xa2bfe8a1 0xa81a664b 0xc24b8b70
0xc76c51a3 0xd192e819 0xd6990624 0xf40e3585 0x106aa070 0x19a4c116
0x1e376c08 0x2748774c 0x34b0bcb5 0x391c0cb3 0x4ed8aa4a 0x5b9cca4f
0x682e6ff3 0x748f82ee 0x78a5636f 0x84c87814 0x8cc70208 0x90befffa 0xa4506ceb
0xbef9a3f7 0xc67178f2

- Expanded message blocks W_0, W_1, …., W_{63} are computed as follows via the SHA-256 message schedule:

```
$W_t = M_t^{(i)}$ , for t=0,1,…..,15 and,
For j=16 to 63
{
$W_t = \sigma 1(W_{t-2}) + W_{t-7} + \sigma 0(W_{t-15}) + W_{t-16}$
}
```

3.2 Preprocessing

Computation of the hash of a message begins by preparing the message:

1. Pad the message in the usual way: Suppose the length of the message M, in bits, is L, Append the bit "1" to the end of the message, and then k zero bits, where k is the smallest non-negative solution to the equation $L+1+k \equiv 448 \bmod 512$. To this append the 64-bit block which is equal to the number L written in binary. For example, the (8-bit ASCII) message "abc" has length $8\times3 = 24$ so it is padded with a one, then $448-(24+1) = 423$ zero bits, and then its length to become the 512-bit padded message.

   The length of the padded message should now be a multiple of 512 bits.
2. Parse the message into N 512-bit blocks $M^{(1)}$, $M^{(2)}$,………., $M^{(N)}$. The first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$ .We use the big-endian convention throughout, so with in each 32-bit word, the left-most bit is stored in the most significant bit position.

3.3 Block Decomposition

For each block $M \in \{0,1\}^{512}$, 64 words of 32 bits each are constructed as follows:

- the first 16 are obtained by splitting M in 32-bit blocks.

$$M = W_0 \| W_1 \| \cdot\cdot\cdot \| W_{14} \| W_{15}$$

- the remaining 48 are obtained with the formula:

$$W_t = \sigma1(W_{t-2}) + W_{t-7} + \sigma0(W_{t-15}) + W_{t-16}\,,\ \ 16 \le t \le 63$$

6.2.4 Hash Computation

4.1 Main loop:

The hash computation proceeds as follows:

For i = 1 toN (N = number of blocks in the padded message)

{

- Initialize registers a, b, c, d, e, f, g, h with the (i-1)st intermediate hash value(= the initial hash value when i = 1)

$a = H_0^{(i-1)}$, $b = H_1^{(i-1)}$, $c = H_2^{(i-1)}$, $d = H_3^{(i-1)}$, $e = H_4^{(i-1)}$, $f = H_5^{(i-1)}$, $g = H_6^{(i-1)}$, $h = H_7^{(i-1)}$

- Apply the SHA-256 compression function to update registers a, b,…….., h

```
For t = 0 to 63
{
    Compute Ch(e, f, g), Maj(a, b, c), $\sum_0(a)$, $\sum_1(e)$, and $W_t$ (see Definitions above)
    $T_1=h+\sum_1(e)+ Ch(e, f, g)+ K_j+ W_t$
    $T_2= \sum_0(a)+ Maj(a, b, c)$
    h=g,
    g=f,
    f=e,
    $e=d+T_1,$
    d=c,
    c=b,
    b=a,
    $a=T_1+ T_2$
}
```

- Compute the ith intermediate hash value $H^{(i)}$.

$H_0^{(i)} = a + H_0^{(i-1)}$

$H_1^{(i)} = b + H_1^{(i-1)}$

$H_2^{(i)} = c + H_2^{(i-1)}$

$H_3^{(i)} = d + H_4^{(i-1)}$

$H_4^{(i)} = e + H_4^{(i-1)}$

$H_5^{(i)} = f + H_5^{(i-1)}$

$H_6^{(i)} = g + H_6^{(i-1)}$

$H_7^{(i)} = h + H_7^{(i-1)}$

}

- After repeating above steps total of N times (i.e., after processing $M^{(N)}$), the resulting 256-bit message digest of the message, M, is

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$$

6.2.5 Flame Graph:

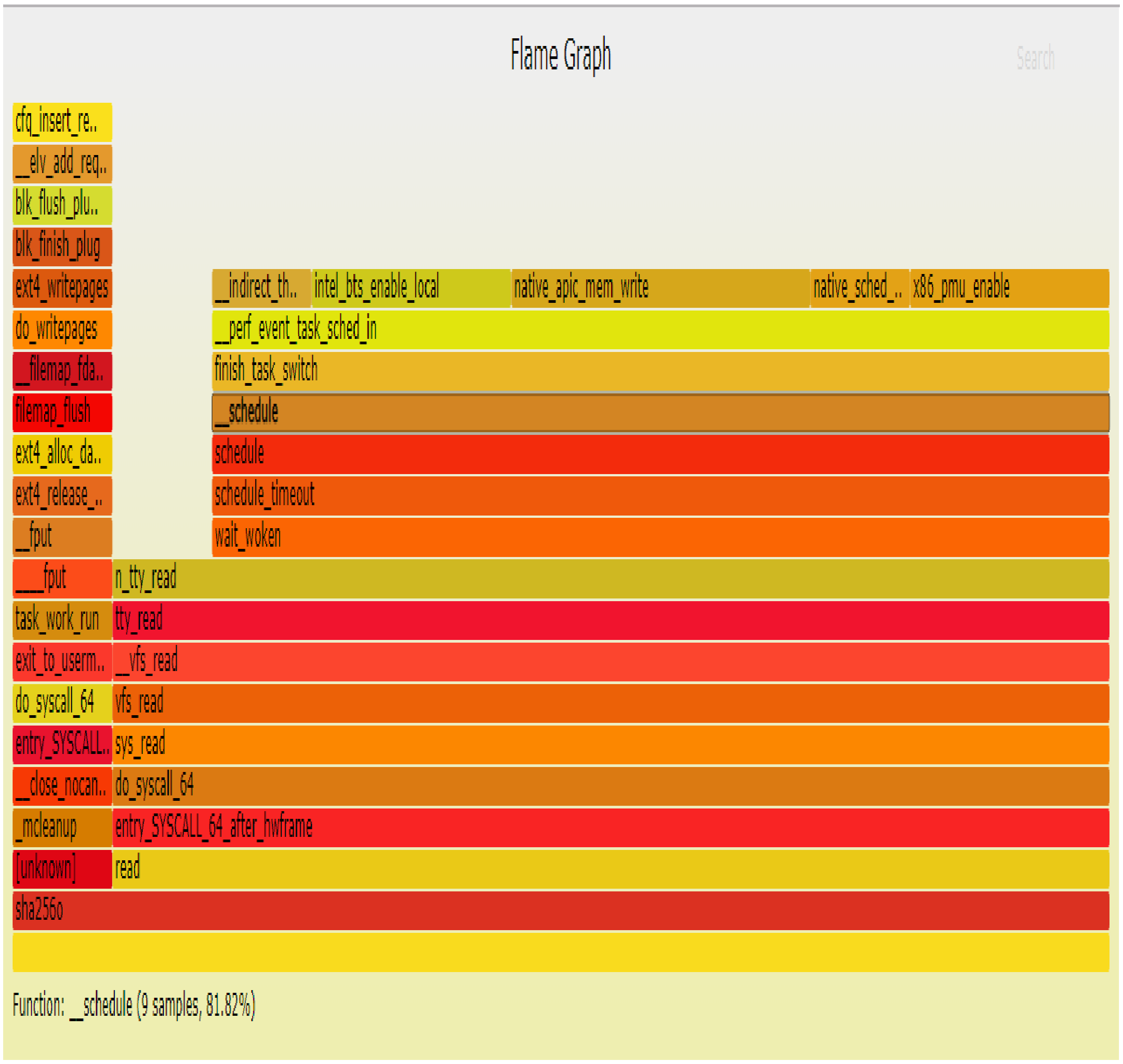


Figure 5.1

6.3 Comparative Analysis of Digital signature Based Algorithms:

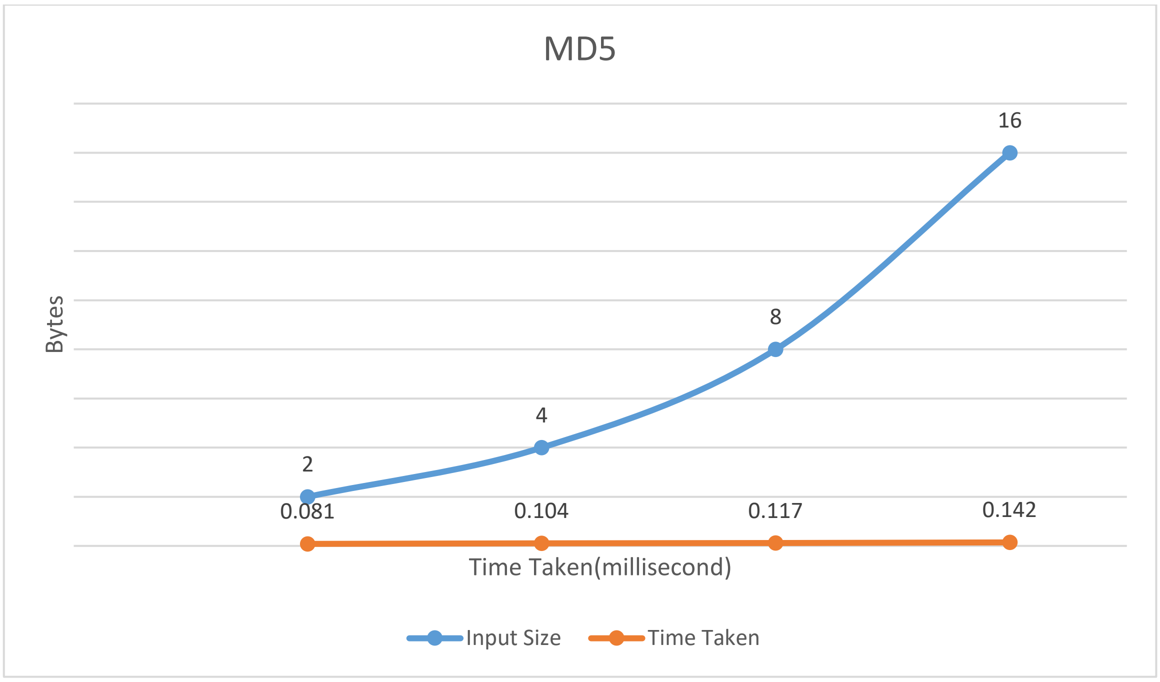


Figure 6.1

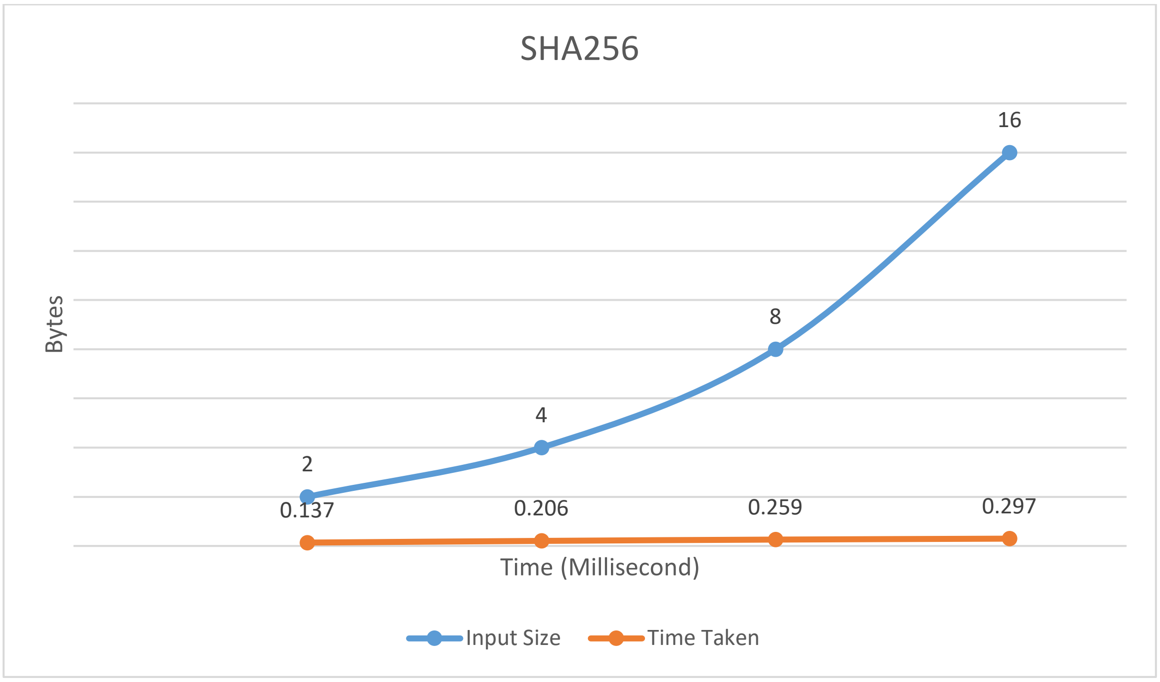


Figure 6.2

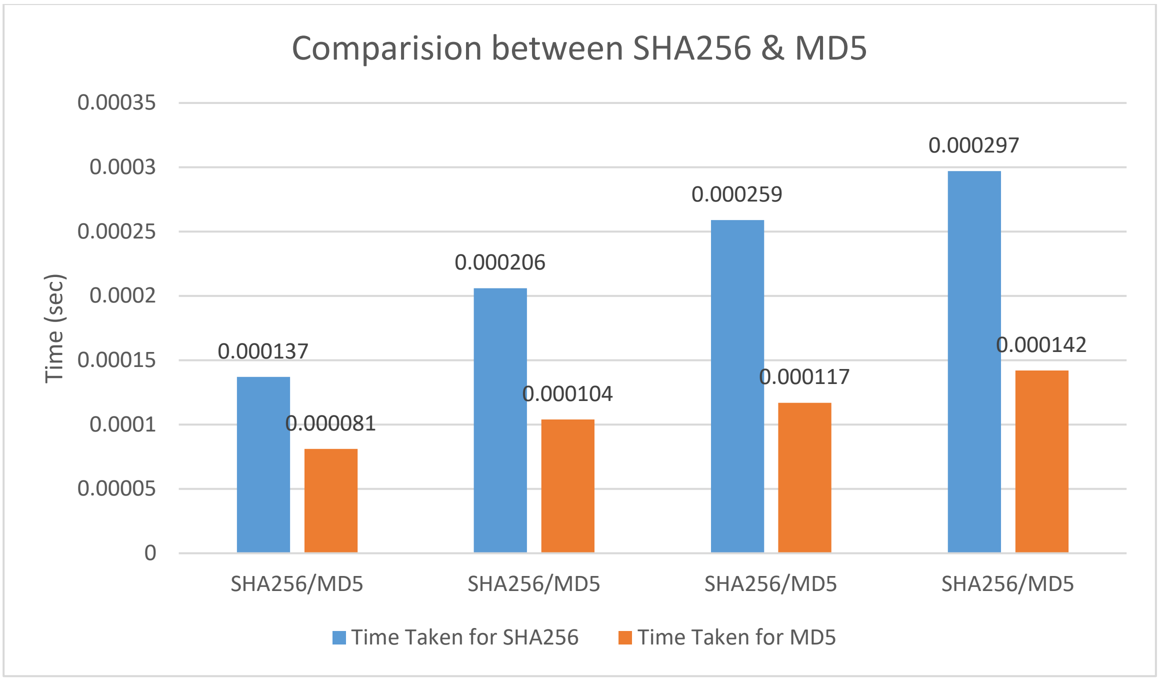

Comparision between SHA256 & MD5
Time (sec)
0.00035
0.0003
0.00025
0.0002
0.00015
0.0001
0.00005
0
0.000137
0.000081
0.000206
0.000104
0.000259
0.000117
0.000297
0.000142
SHA256/MD5
SHA256/MD5
SHA256/MD5
SHA256/MD5
Time Taken for SHA256
Time Taken for MD5

Figure 6.3

7. Image Based Encryption:

7.1 Rubik’s Cube Algorithm:

7.1.1 Introduction:

In the modern era, exchange of information and multimedia data has increased exponentially through the internet. However, these digital advantages have also brought the downside such as illegitimate copying and distribution of digital multimedia data. To combat these challenges different techniques such as encryption and digital data watermarking have been proposed. In this project our emphasis is on the security of image data exchanged throughout the internet. Many symmetric and asymmetric encryption algorithm such as private key technique base DES (Data Encryption Standard) and AES (Advanced Encryption Standard), public key technique base RSA (Rivest Shamir Adleman), ECC (Elliptic Curve Cryptography), Chaos based image encryption, etc. have been proposed and are being used for protection of image data. However, most of them while providing excellent security to images may not be favourable for securing images where fast and secured image enciphering is needed on networks for data exchange applications. In this project we aim to write an image encryption technique based on Rubik’s cube algorithm principal. This algorithm performs image encryption work by scrambling the position of pixel of native image by applying Rubik’s cube principal. Then using two unique private keys it applies XOR operation to firstly the pixel rows and then the pixel columns of new scrambled image. The outcome of the security analysis of Rubik’s cube principal demonstrate that this algorithm can accomplish great encryption and can also resist various attacks like noise base attacks, statistical attack and differential attack.

7.1.2 Rubik’s Cube Image Algorithm:

In this segment the Rubik’s cube encryption algorithm along with decryption is detailed as follows-:

2.1 **Encryption Algorithm**

The program assumes that the original image is a grayscale image of size 256 X 256 pixels. For now, let us assume I_0 represents the pixel values of matrix M X N of an □-bit grayscale image. The encryption steps are detailed as follows-:

1. We are generating two random vectors K_R (rows) and K_c (columns) of length M X N respectively. Now we take elements $K_R(i)$ and $K_C(j)$ each take a random value from set A = {0,1,2,...,$2^{□}$-1}. The value of both K_R and K_C must not have constant values.
2. We will determine the number of times perform this algorithm and set $ITER_{MAX}$ and initialize the counter ITER at 0.
3. For each row i of image I_0
    a. We will compute the sum of all elements in the row i, this sum is denoted by (i).

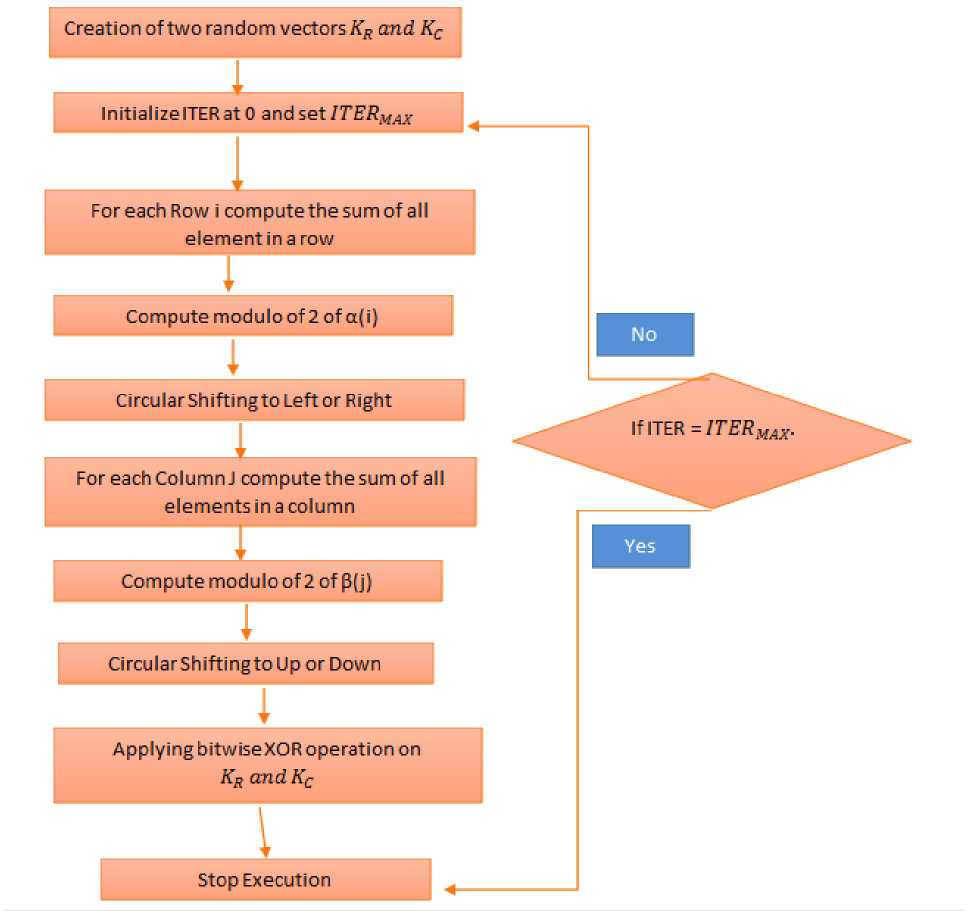
    $$(i)= \sum_{j=1}^{N} \quad I_0 (i,j),\ i=1,2,...,M$$

    b. We will compute modulo of 2 of (i), which is denoted by $M_{a(i)}$.
    c. The row i is circular shifted to left or right direction by $K_R(i)$ positions and the first pixels moves in the last pixels position. It circular shifts according to described as follows-:
        i. if $M_{□(i)} = 0 \rightarrow$ right circular shift
        ii. else $\rightarrow$ left circular shift
4. Now for each column j of image I_0
    a. We will compute the sum of all elements in the row j, this sum is denoted by (j).

    $$(j) = \sum_{i=1}^{M} \quad I_0 (i,j),\ j= 1,2,...,N$$

    b. We will compute modulo of 2 of (j), which is denoted by $M_{□(j)}$.
    c. The column j is circular shifted to up or down direction by $K_C(i)$ positions. It circular shifts according to described as follows-:
        i. if $M_{□(j)} = 0 \rightarrow$ up circular shift
        ii. Else $\rightarrow$ down circular shift
5. The step 3 and 4 detailed above will create a scrambled image which is denoted by I_{SCR}.
6. Using the vector K_C described earlier, we will now apply bitwise XOR operator function to each row of the newly scrambled image I_{SCR} using the following described expression-:
    a. $I_1 (2i-1, j) = I_{SCR} (2) \oplus K_C (j)$ (for the odd value of i)
    b. $I_1 (2i, j) = I_{SCR} (2i, j) \oplus rot\ 180(K_C (j))$ (for even value of i)
    c. where $\oplus$ and rot 180(K_C) represents the bitwise operator XOR and the flipping of K_C from left to right, respectively

7. Using vector K_R described earlier, we will now apply bitwise XOR operator function to each column of newly formed image I_1 from above step using the following described expression-:
    a. I_{ENC} (i, 2j-1) = I_1 (i, 2j -1) $\oplus K_R$ (j) (for the odd value of j)
    b. I_{ENC} (i, 2j) = I_1(o, 2j) $\oplus$rot 180 (K_R (j)) (for the even value of j)
    c. where the rot 180(K_R) is indicating the left to right flip of vector K_R.
8. If ITER = $ITER_{MAX}$ then encrypted image I_{ENC} is created and encryption process is done. Otherwise, the encryption algorithm branches back to Step 3 and continues till ITER condition is satisfied.
9. The vectors K_R, K_C and the max iteration number $ITER_{MAX}$ are considered as secret keys in the proposed encryption algorithm. For, the fastest encryption set $ITER_{MAX}$ = 1 as it is single iteration. For higher secure image we can increase the $ITER_{MAX}$ value.



Creation of two random vectors K_R and K_C
Initialize ITER at 0 and set $ITER_{MAX}$
For each Row i compute the sum of all element in a row
Compute modulo of 2 of α(i)
Circular Shifting to Left or Right
For each Column J compute the sum of all elements in a column
Compute modulo of 2 of β(j)
Circular Shifting to Up or Down
Applying bitwise XOR operation on K_R and K_C
Stop Execution
No
If ITER = $ITER_{MAX}$.
Yes

Fig. 7.1

2.2 **Decryption Algorithm**

The original initial decrypted image I_0 is now retrieved from the newly formed encrypted image I_{ENC} by applying the encryption steps in reverse utilizing the following steps-:

1. First initialize the ITER = 0.
2. Now increment the ITER counter by on: ITER = ITER + 1
3. The bitwise XOR operator function is applied on the vector K_R and each column of the encrypted image I_{ENC} as shown follows:
    a. I_1 (i, 2j -1) = I_{ENC}(i, 2j-1) $\oplus K_R$ (j) (for odd value of j)
    b. I_1 (i, 2j) = I_{ENC} (i, 2j) $\oplus$rot 180(K_R (j)) (for even value of j)
4. Now using the K_C vector the bitwise XOR operator is applied to each row of the image I_1:
    a. I_{SCR} (2i-1, j) =I_1(2i-1, j) $\oplus K_C$ (j) (for odd value of i)
    b. I_{SCR} (2i, j) = I_1 (2i, j) $\oplus$rot 180 (K_C (j)) (for even value of i)
5. For each column j of the scrambled image I_{SCR}
    a. We will compute the sum of all elements in that column j which is denoted as $□_{SCR}$(j):
        i. $□_{SCR}(j) = \sum_{i=1}^{M} \quad I_{SCR}(i, j)$, j = 1,2,...,N
    b. We will compute modulo of 2 of $□_{SCR}$(j) which is denoted by $M□_{SCR}$(j).
    c. The column j is circular shifted in up or down directions by K_c(i) positions according to the following conditions-:
        i. if $M□_{SCR}(j) = 0$ → up circular shift
        ii. else → down circular shift
6. For each row i of scrambled image I_{SCR}
    a. We will compute the sum of all elements in row i which is denoted by $□_{SCR}$(i):
        i. $□_{SCR}(i) = \sum_{j=1}^{N} \quad I_{SCR}(i, j)$, i = 1,2,...,M
    b. We will compute modulo 2 of $□_{SCR}$(j) which is denoted by $M□_{SCR}$(j).
    c. The row i is circular shifted in right or left directions by K_R(i) positions according to the following conditions-:
        i. if $M□_{SCR}(j) = 0$ → right circular shift
        ii. else → left circular shift
7. If ITER = $ITER_{MAX}$, then image I_{ENC} is decrypted and the decryption process is done. Otherwise, the decryption algorithm branches back to Step 2.

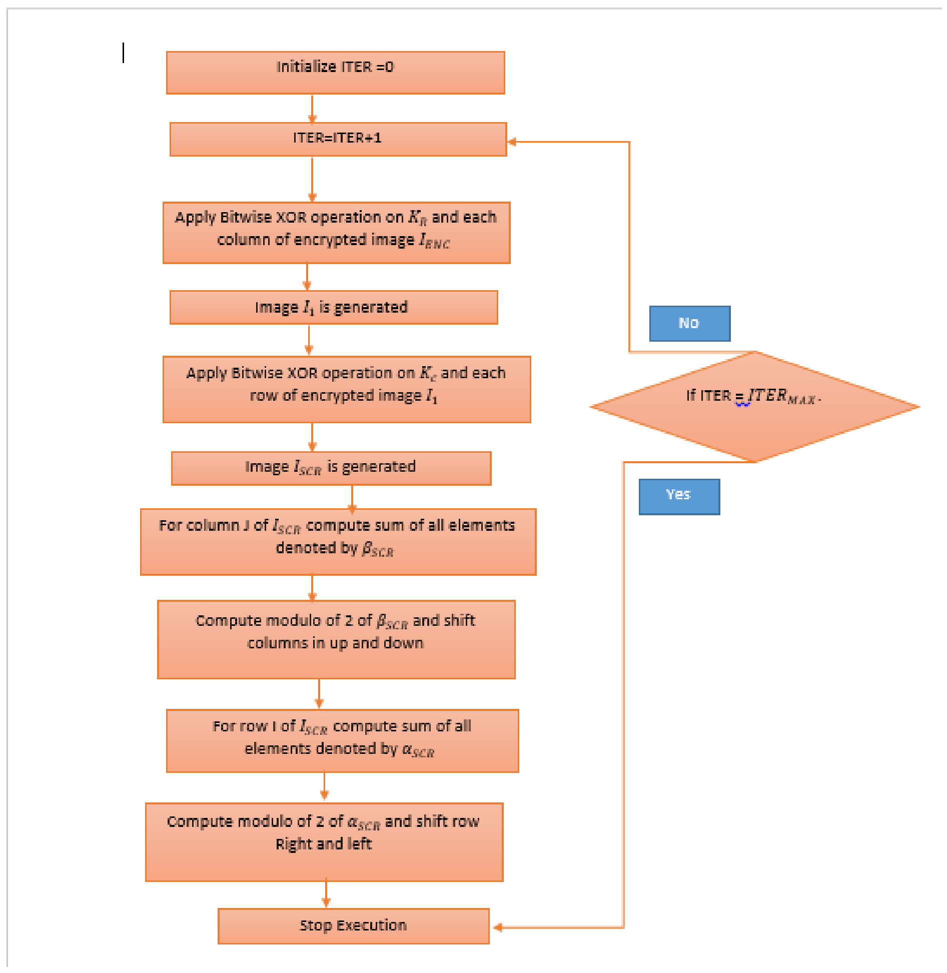
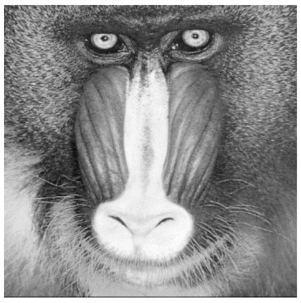
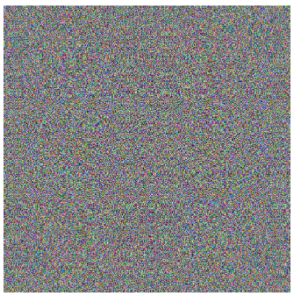

Initialize ITER =0
ITER=ITER+1
Apply Bitwise XOR operation on K_R and each column of encrypted image I_{ENC}
Image I_1 is generated
Apply Bitwise XOR operation on K_c and each row of encrypted image I_1
Image I_{SCR} is generated
For column J of I_{SCR} compute sum of all elements denoted by β_{SCR}
Compute modulo of 2 of β_{SCR} and shift columns in up and down
For row I of I_{SCR} compute sum of all elements denoted by α_{SCR}
Compute modulo of 2 of α_{SCR} and shift row Right and left
Stop Execution
No
If ITER = $ITER_{MAX}$.
Yes

Fig. 7.2

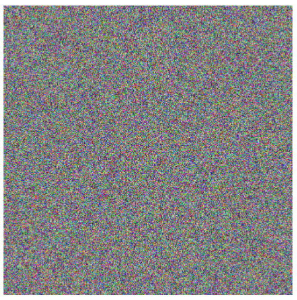
7.1.3 Experimental Results:

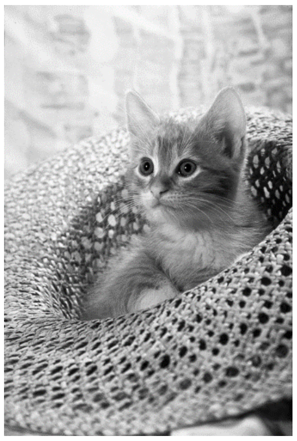
Original and their respective encrypted images.

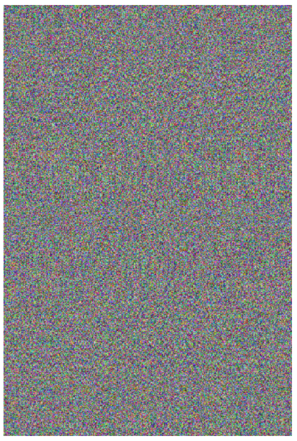

(a) Baboon (original)

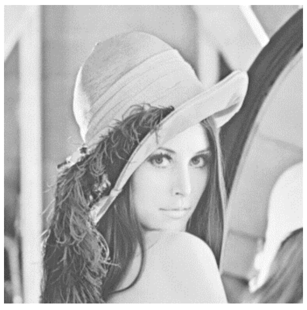

(a) Baboon (encrypted)

(b) Blackbuck (original)


(b) Blackbuck (encrypted)


(c) Cat (original)


(c) Cat (encrypted)


(d) Lena (original)

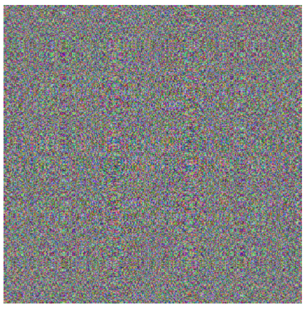
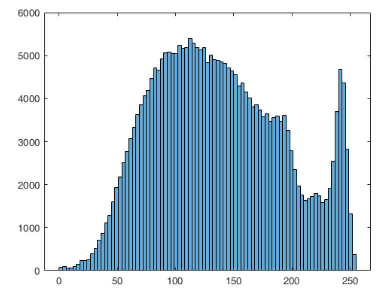
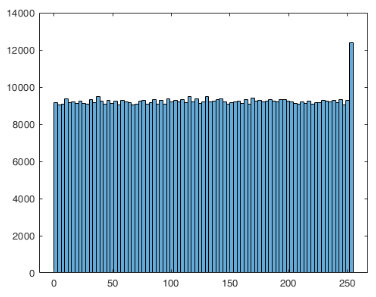
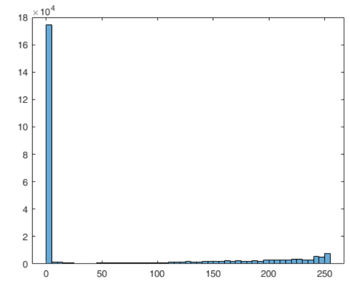

(d) Lena (encrypted)

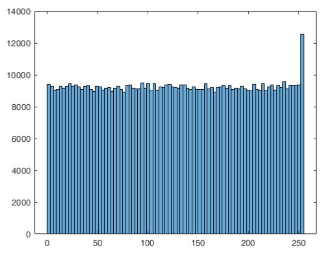
Fig. 7.3

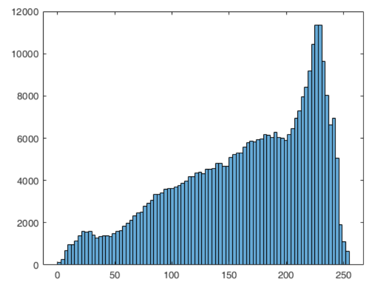
Histograms of original and encrypted images.

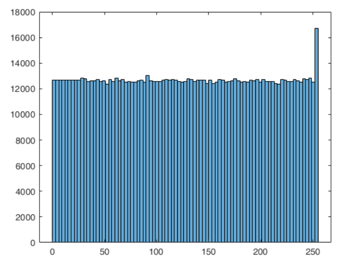

(a) Original image: Baboon

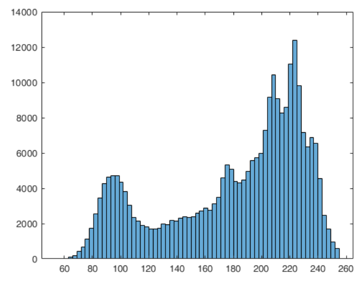

(a) Encrypted image: Baboon


(b) Original image: Blackbuck


(b) Encrypted image: Blackbuck


(c) Original image: Cat


(c) Encrypted image: Cat


(d) Original image: Lena

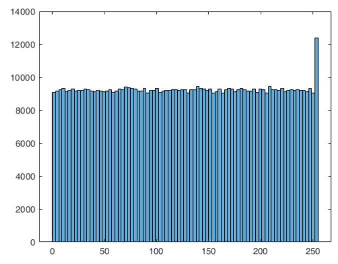
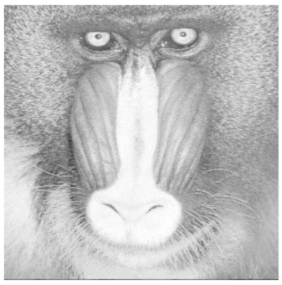
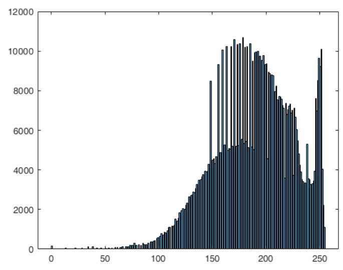
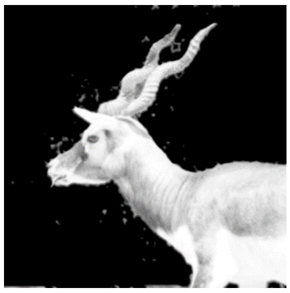

(d) Encrypted image: Lena

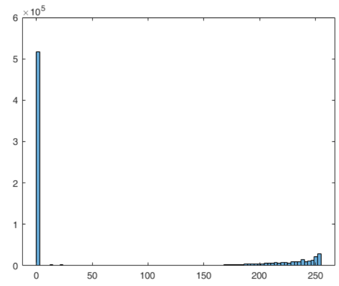
Fig. 7.4

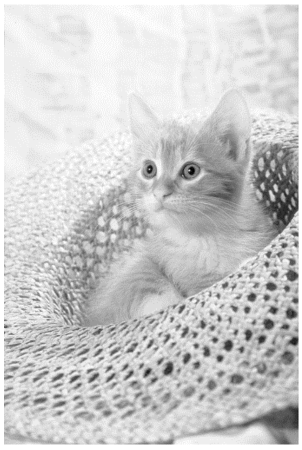
Decrypted images and their respective histograms.

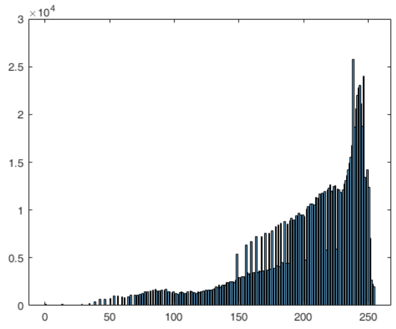

(a) Baboon (decrypted)

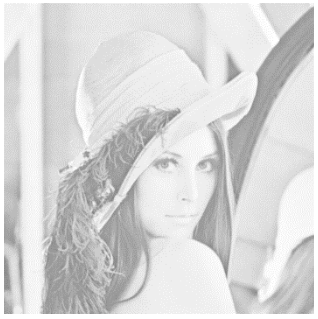

(a) Baboon (decrypted histogram)


(b) Blackbuck (decrypted)


(b) Blackbuck (decrypted histogram)


(c) Cat (decrypted)


(c) Cat (decrypted histogram)


(d) Lena (decrypted)

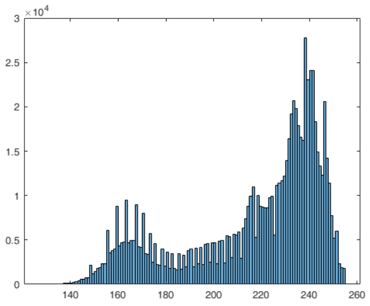

(d) Lena (decrypted histogram)

Fig. 7.5

7.1.4 Conclusion:

In this project, a fast and secure image encryption algorithm based on Rubik's cube principle his algorithms perform image encryption work by scrambling the position of pixel of native image by applying Rubik's cube principal. Then using two unique private keys it applies XOR operation to firstly the pixel rows and then the pixel columns of new scrambled image. The outcome of the security analysis of Rubik's cube principal demonstrate that this algorithm can accomplish great encryption and can also resist various attacks like noise base attack, statistical attack and differential attack is implemented.

7.2 AES CTR Block Algorithm:

7.2.1 Introduction:

The Advanced Encryption Standard (AES), also known as Rijndael is a block cipher. There are five modes of operations for AES which are defined by NIST. Each of these modes has different characteristics. The five modes are: Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feed-Back (CFB), Output Feed-Back (OFB), and Counter (CTR). Only AES Counter mode (AES-CTR) is applied here.

7.2.2 Overview of Counter-Mode Encryption:

CTR requires the encryptor to generate a unique per-packet value, and communicate this value to the decryptor. This per-packet value is called an initialization vector (IV). The same IV and key combination must not be used more than once. Encryptor can generate the IV in any manner that ensures uniqueness. Common approaches to IV generation include incrementing a counter for each packet and linear feedback shift registers (LFSRs). In this report, we don't consider LFSRs. To encrypt using CTR-mode encryption, one starts with a plaintext M (an arbitrary bit string), a key K, and a counter CTR, where CTR is an n-bit string. Let C be the XOR (excusive-or) of M and the first M bits of the pad E K (CTR) || E K (CTR + 1) || E K (CTR + 2) · · ·. The cipher-text is (CTR, C), or, more generally, C together with something adequate to recover ctr. To decrypt cipher-text (CTR, C) compute the plaintext M as the XOR of C and the first C bits of the pad E K (CTR) || E K (CTR + 1) || E K (CTR + 2) · · ·. Therefore, decryption is the same as encryption with M and C interchanged. Often we refer to C itself, rather than (CTR, C), as the cipher-text. In the recommended usage scenario, the party encrypting maintains an integer counter, nonce, initially 0, and produces the string CTR as the 128-bit string which encodes the number nonce·2^64 . (In other words, nonce is regarded as a 64-bit binary number, and CTR is constructed by appending to this number 64 zero-bits.) The number nonce is incremented following each encryption. Typically, one transmits C along with a string which encodes nonce. A well-designed standard for CTR mode should not be overly prescriptive about how CTR is formed or what beyond C is explicitly communicated between sender and receiver. To illustrate some possibilities:

1 The value CTR is derived from a nonce by the method just described, and the cipher-text specifies both nonce and C.

2 The same, except that no nonce-value is explicitly transmitted to the receiver because the sender and the receiver maintain state and communicate over a reliable channel.

3 The same, except that nonce starts at a random value in 0 · · · 2 64−1 instead of starting at 0.

4 CTR is a random 128-bit string, selected afresh with each message sent.

5 CTR is determined implicitly by other protocol elements, such as an accompanying sequence number (e.g., in the context of IPSEC).

In Counter Mode, successive input blocks are generated by a next-counter function, which can be a simple operation such as an integer increment modulo 2 w, where w is a convenient register width. The details of the next-counter function are unimportant; that function does not provide any security properties other than the uniqueness of the inputs to the block cipher1. These details could be important if the block cipher is a legacy cipher (e.g. DES), but they are not important when considering ciphers like AES that are believed to be secure. CTR mode has significant implementation advantages as below:

- Software efficiency: By eliminating the computational dependency between Ci and C j, CTR-mode encryption enables effective utilization of aggressive pipelining, multiple instruction dispatch per clock cycle, a large number of registers, and SIMD instructions.
- Hardware efficiency: CTR model is fully parallelizable. One can be computing blocks C 1, C 2 , · · · all at the same time, limited only by the amount of hardware that one throws at the problem.
- Preprocessing: Preprocessing can be used, in some environments, to increase speed. That is, one can compute the pad in ”spare cycles,” even before one knows the plaintext M . When M is known, it is XORed with the already-computed pad.
- Random-access: The ith cipher-text block, Ci, can be encrypted in a random-access fashion. In hard-disk encryption, this is important because bulk data needs to be encrypted quickly. When using CBC mode, properly encrypting the ith block requires one to first encrypt the i − 1 prior blocks.
- Provable security: The concrete security bounds one gets for CTR-mode encryption, using a block cipher, are no worse than what one gets for CBC encryption. Simplicity. With CTR mode, both encryption and decryption depend only on E. Moreover, the decryption key scheduling need not be implemented. Together, this makes implementations simpler, and may yield a significant throughput increase in hardware.

7.2.3 Encryption in CTR Mode:

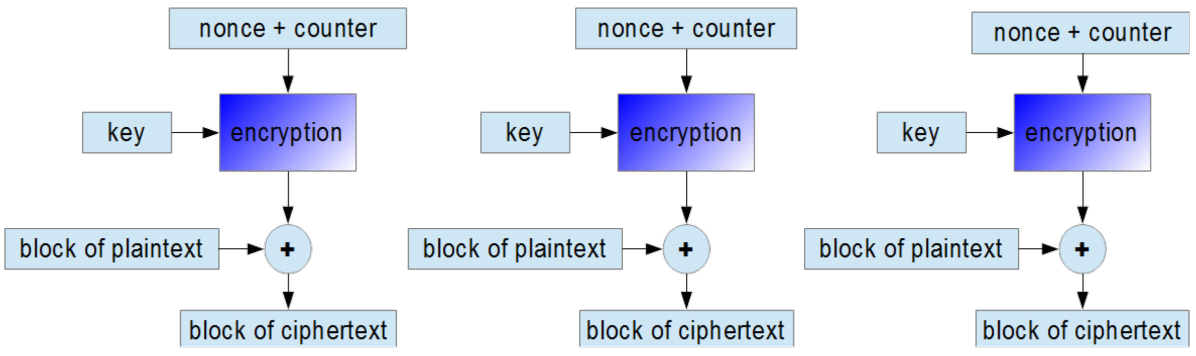


Fig. 7.6

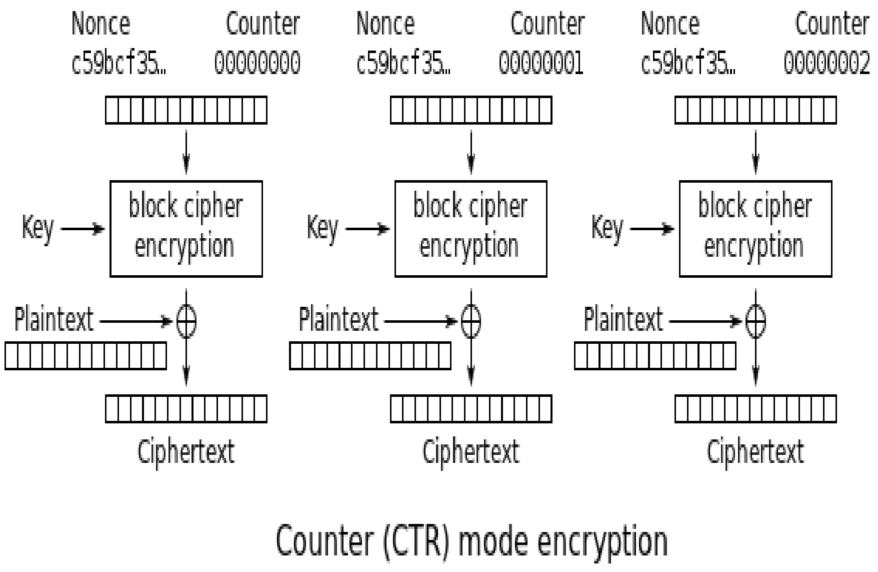


Fig. 7.7

7.2.4 Decryption in CTR Mode:

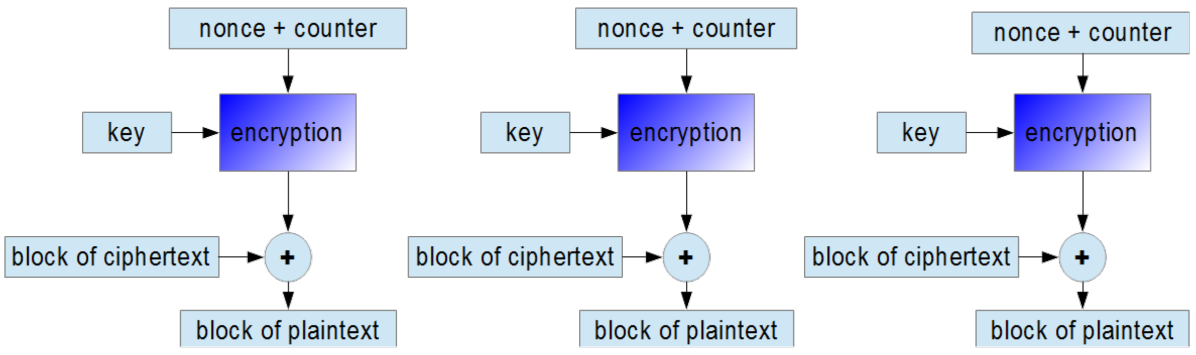


Fig. 7.8

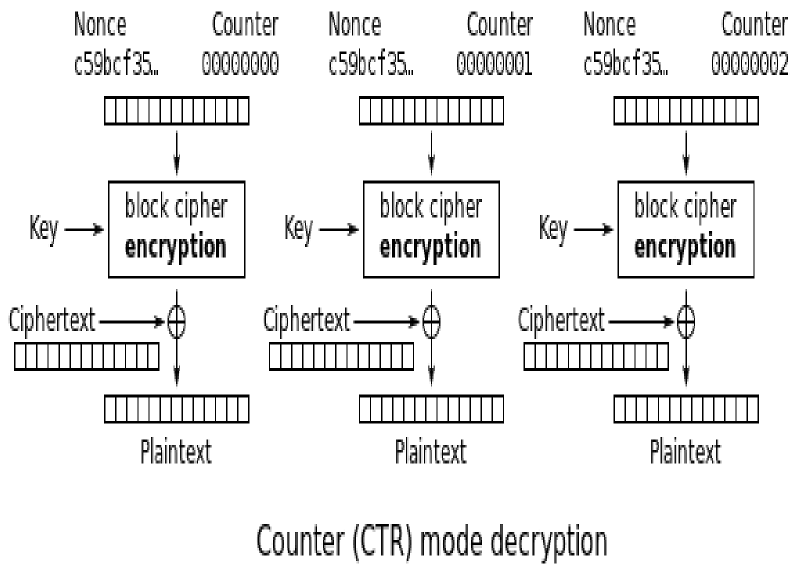


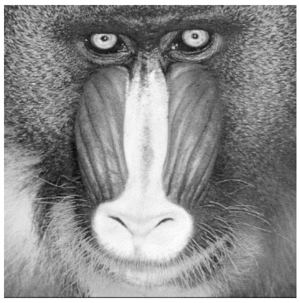
Fig. 7.9

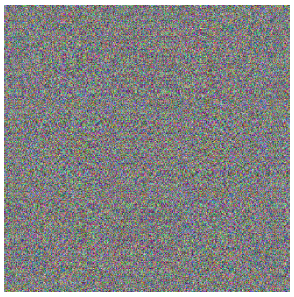
7.2.5 Disadvantages of CTR Mode:

- No integrity: CTR-mode encryption provides no message integrity. However, it is not normally considered the purpose of encryption to provide for message integrity, and other common modes, like CBC, likewise fail to provide any meaningful message integrity. Error propagation. If a bit-flip error occurs in some block of cipher-text, after decryption, the error is localized to the corresponding bit of the corresponding block of plaintext.
- Statefull encryption: The situation with respect to statefullness is essentially the same as with CBC mode.
- Sensitivity to usage errors: It is crucial in CTR-mode encryption that a counter-value not be reused (both the CTR -value that logically accompanies C, and also all “internal” counters that mask plaintext blocks). Standard should make clear what is required of the counter-values in order to be secure and recommend a small number of secure usage scenarios.
- Interaction with weak ciphers: Successive blocks CTR and CTR +1 usually have small Hamming difference. This has led to the concern that an attacker can obtain many plaintext pairs with a known small plaintext difference, which would facilitate the differential cryptanalysis.

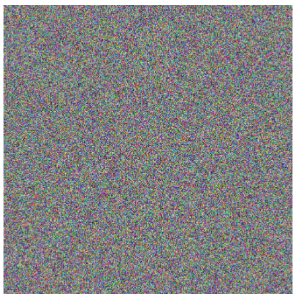
7.2.6 Experimental Results:

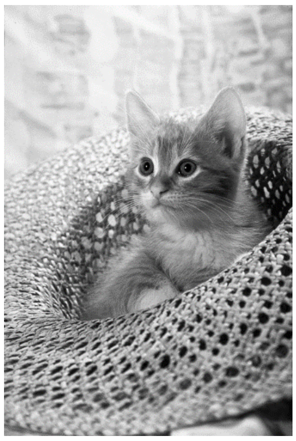
Original and their respective encrypted images.

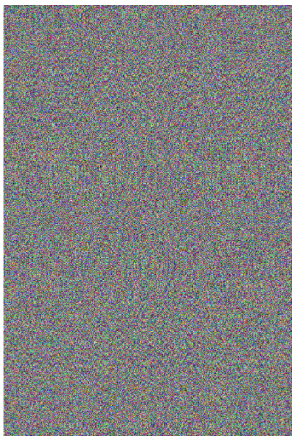

(a) Baboon (original)

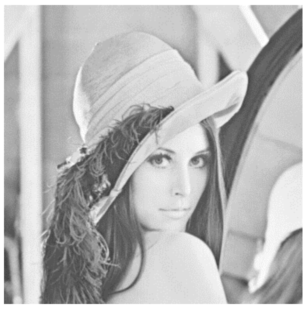

(a) Baboon (encrypted)

(b) Blackbuck (original)


(b) Blackbuck (encrypted)


(c) Cat (original)


(c) Cat (encrypted)


(d) Lena (original)

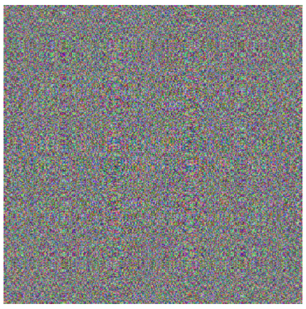
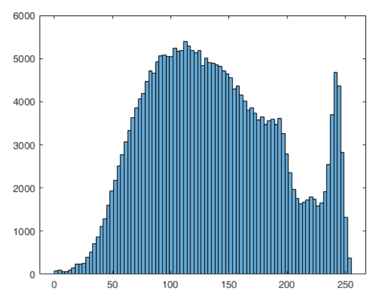
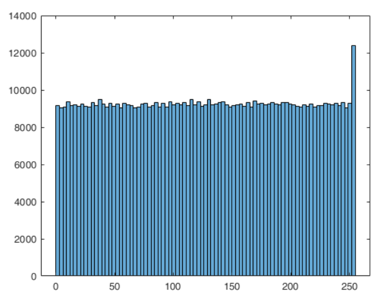
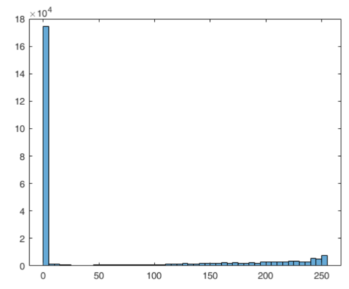

(d) Lena (encrypted)

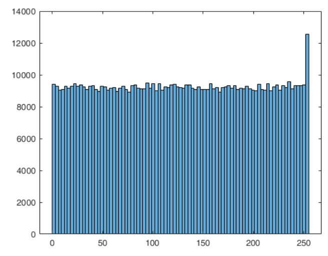
Fig. 7.10

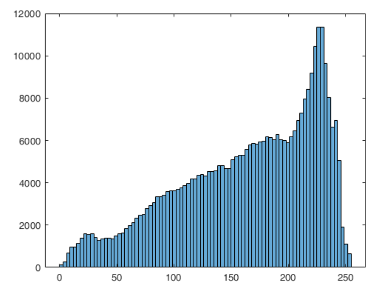
Histograms of original and encrypted images.

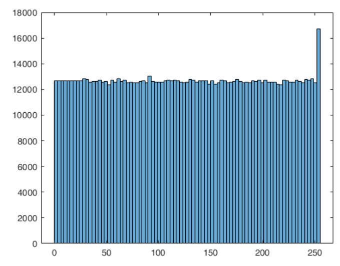

(a) Original image: Baboon

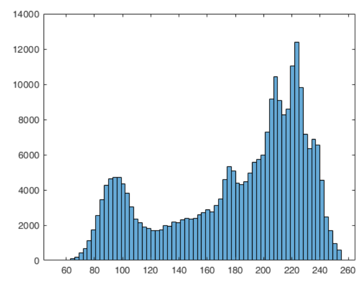

(a) Encrypted image: Baboon


(b) Original image: Blackbuck


(b) Encrypted image: Blackbuck


(c) Original image: Cat


(c) Encrypted image: Cat


(d) Original image: Lena

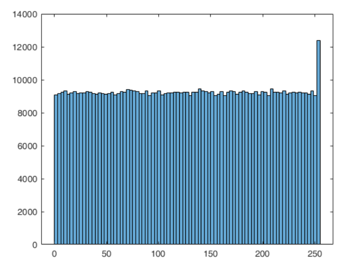
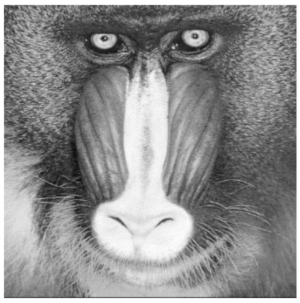
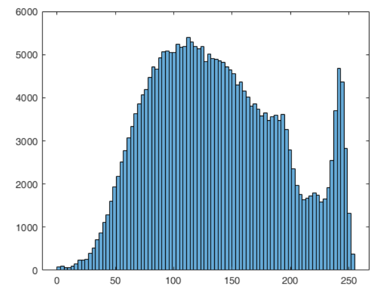

(d) Encrypted image: Lena

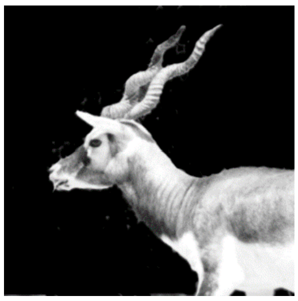
Fig. 7.11

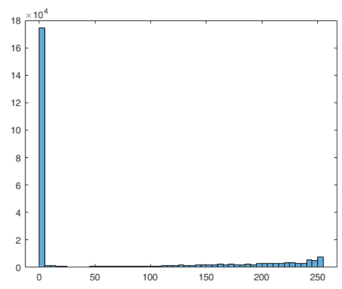
Decrypted images and their respective histograms.


(a) Baboon (decrypted)

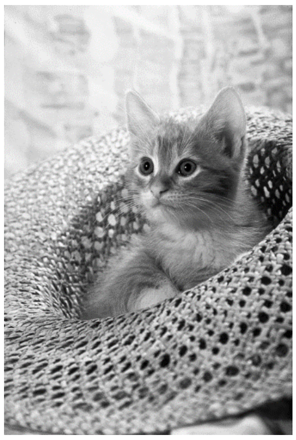


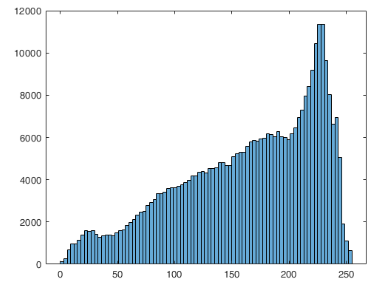
(a) Baboon (decrypted histogram)


(b) Blackbuck (decrypted)

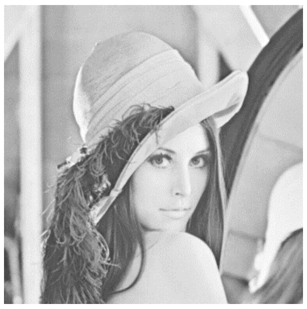


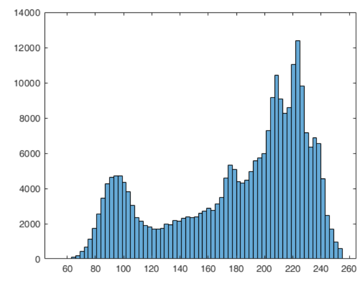
(b) Blackbuck (decrypted histogram)


(c) Cat (decrypted)



(c) Cat (decrypted histogram)


(d) Lena (decrypted)



(d) Lena (decrypted histogram)

Fig. 7.12

7.2.7 Flame Graph:

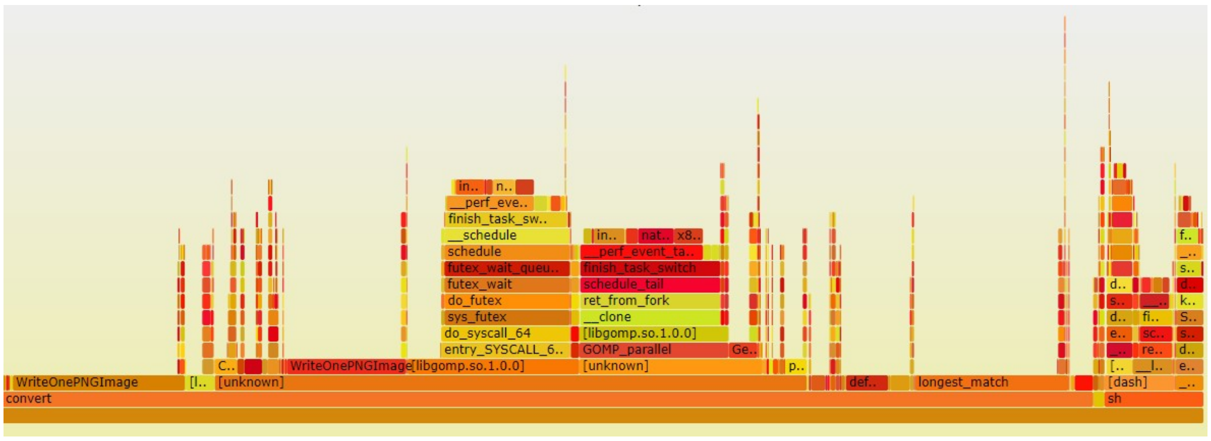


Fig. 7.13

7.2.8 Conclusions:

Counter mode has been well-analyzed by the theoretical cryptography community and security practitioners. Its systems security implications are different than those of CBC mode, but are not problematic. Implementers can leverage experience with other additive ciphers (such as RC4 and SEAL) when adding CTR to cryptographic systems. We suggest that CTR be used in conjunction with a message authentication code, that it be used with automated key management, and that the counter value by maintained within the cryptographic module implementing CTR. Precomputation attacks are a realistic threat. Adding a 64-bit unpredictable value to the initial counter provides a reasonable level of protection against these attacks, consistent with recommended cipher strengths. This unpredictable value should be considered part of the CTR key, for the purposes of key management.

7.3 Comparative Analysis of Image Algorithms:

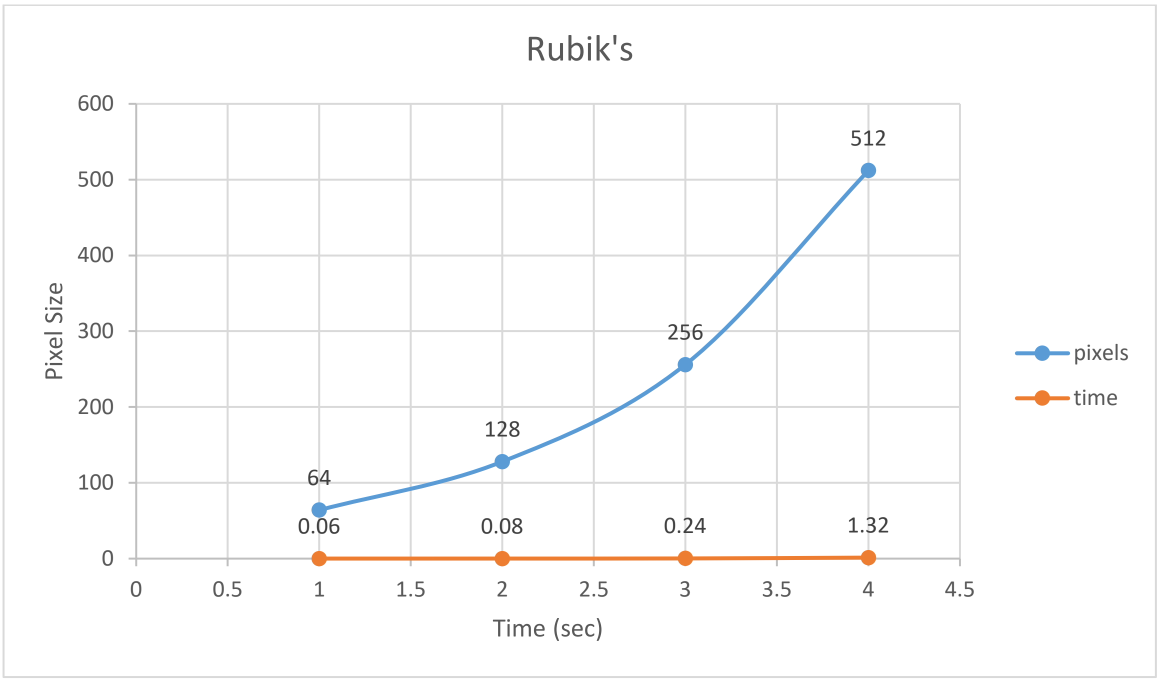


Fig. 7.14

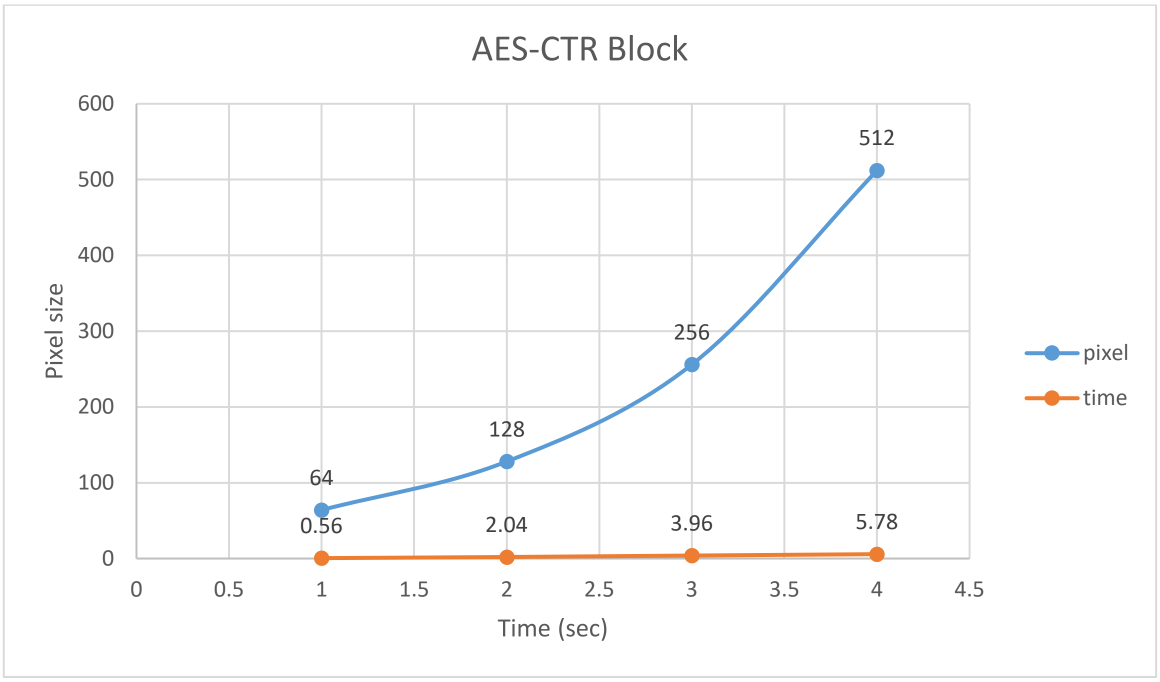


Fig. 7.15

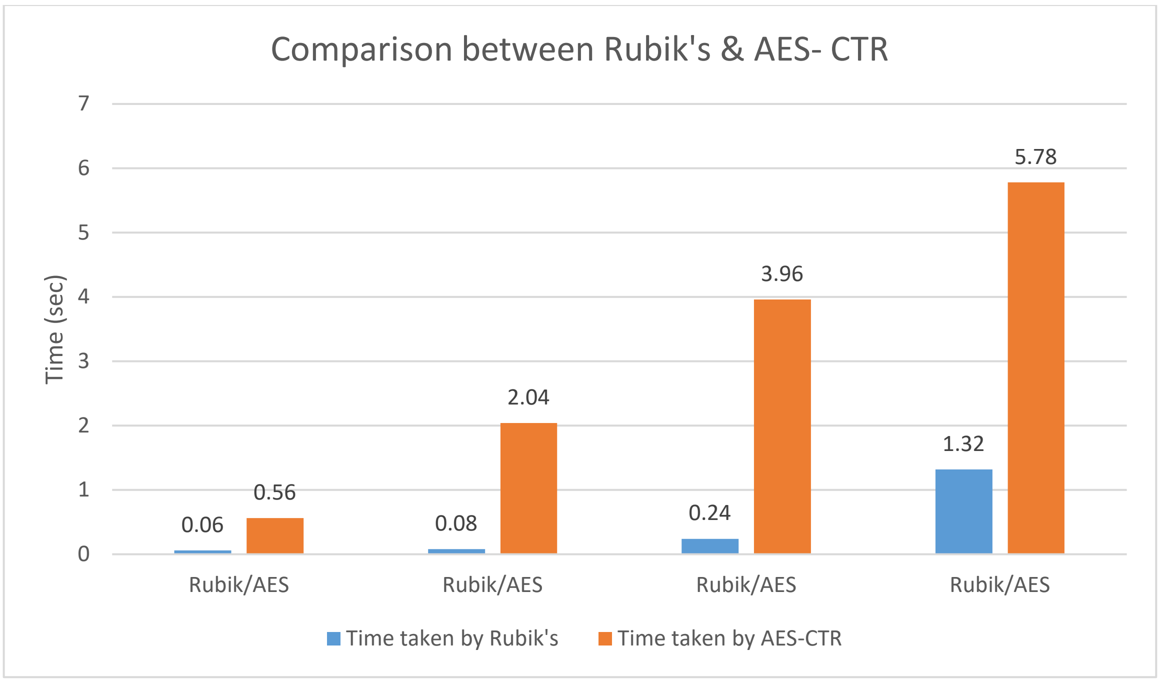

Comparison between Rubik's & AES- CTR
Time (sec)
7
6
5
4
3
2
1
0
0.06
0.56
0.08
2.04
0.24
3.96
1.32
5.78
Rubik/AES
Rubik/AES
Rubik/AES
Rubik/AES
Time taken by Rubik's
Time taken by AES-CTR

Fig. 7.16

8. References:

1. Comparative Analysis of Encryption Algorithms for Various Types of Data Files for Data Security by Kalyani P. Karule, Neha V. Nagrale. International Journal of Scientific Engineering and Applied Science (IJSEAS) – Volume-2, Issue-2, February 2016 ISSN: 2395-3470 www.ijseas.com

2. Comparative Analysis of Performance Efficiency and Security Measures of Some Encryption Algorithms by AL.Jeeva, Dr.V.Palanisamy, K.Kanagaram. International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 Vol. 2, Issue 3, May-Jun 2012, pp.3033-3037 www.ijera.com

3. A Secure Image Encryption Algorithm Based on Rubik's Cube Principle by Khaled Loukhaoukha, Jean-Yves Chouinard, and Abdellah Berdai. Journal of Electrical and Computer Engineering Volume 2012, Article ID 173931

4. A comparative study of Message Digest 5(MD5) and SHA256 algorithm by D Rachmawati, J T Tarigan and A B C Ginting. IOP Conf. Series: Journal of Physics: Conf. Series 978 (2018) 012116