Developing an application performing Image Encryption Algorithms

Harikesh Kumar
harikesh59@gmail.com

Utkarsh Aanand
Ua.985@gmail.com

Purpose

Our Purpose is to create a Image encryption and Decryption application using certain Algorithms and Perform several Tests like NPCR, UACI etc.

Table of Contents

RESULTS

Scope

Images are widely used in different-different processes. Nowadays, information security is becoming more important in data storage and transmission. Therefore, the security of image data from unauthorized uses is important. Image encryption method prepared information unreadable. Therefore, no hacker or eavesdropper, including server administrators and others, have access to original message or any other type of transmitted information through public networks such as internet.

Definitions, Acronyms, and Abbreviations

AES: Advanced Encryption Standard

DES: Data Encryption Standard

NPCR: Number of changing pixel rate

UACI: Unified averaged changed intensity

IDE: Integrated development environment

TDES: Triple- Data Encryption Standard

References

- Cryptography World { http://www.cryptographyworld.com/algo.htm}
- Wikipidea { Wikipidea https://en.wikipedia.org/wiki/Advanced_Encryption _Standard#Description_of_the_cipher }
- Mathworks {http://www.mathworks.com/matlabcentral/fileexchange/43603npcr-and-uaci-measurements-with-statisticaltests/content/NPCR_and_UACI.m }

Overview

Develop an application with different image encryption algorithms and compare their sensitivity for minor changes in the plain input images. Include features such as NPCR, UACI, Correlation and Shannon entropy calculation between the input plain image and corresponding cipher image

General Description

The program is developed by keeping in mind the security of the images. In this program, Three Algorithms are implemented in this program which are DES (Data Encryption Standard), AES (Advanced Encryption Standard), and Blowfish and Chaotic image encryption using perceptron Model of neurons.

These algorithms takes image as an input and also creates an encrypted image as output. There are also four tests performed by the program to the between the input plain image and corresponding cipher image which includes NPCR, UACI, Correlation and Shannon entropy calculation. We compare the results in the Mat lab code and display the output

Specific Requirements

External Interface Requirements

User Interfaces : Matlab

Hardware Interfaces and Operating System:

Windows 8.1 and Ubuntu 14.04 LS Ram: 8 GB Processor: AMD A6-5250M ,2.9 Ghz   3.1.3

Software Interfaces IDE Used:

- Netbeans IDE 8.0.2 {JAVA}
- Matlab r2013

Inputs

These algorithms takes any image of any format any dimension.

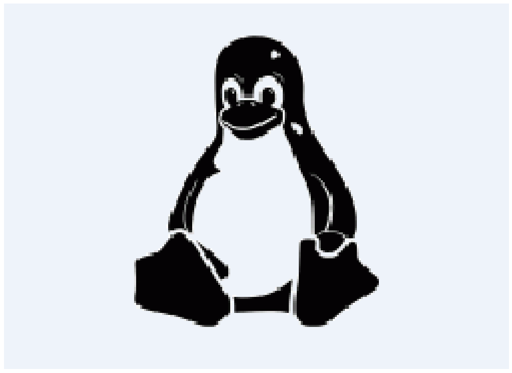
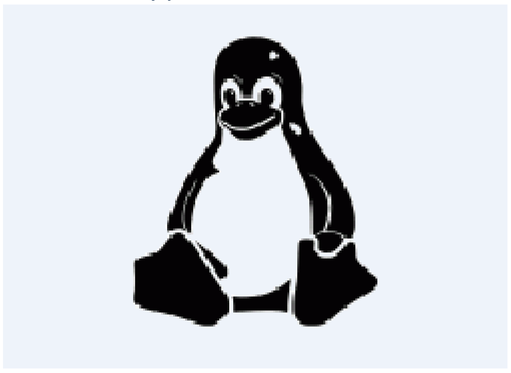


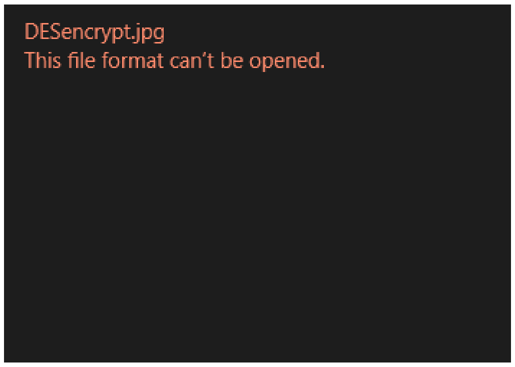
Figure 1 { Input Image}

Outputs and Encryption of the Images:

DES Encrypt OUTPUT:

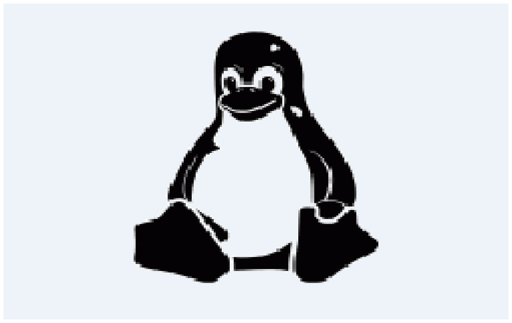


Original Image



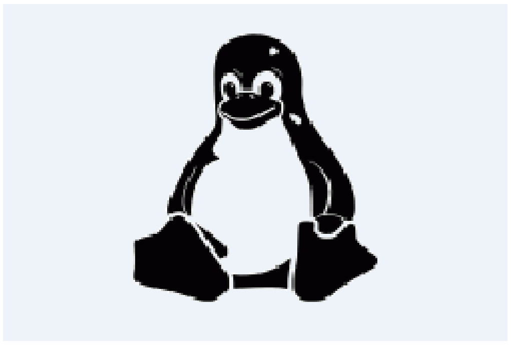
DESencrypt.jpg
This file format can't be opened.

Encrypted Image

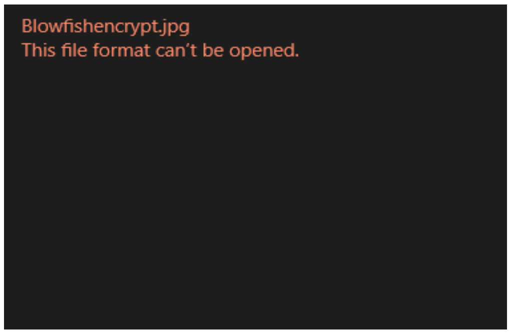


Decrypted Image

Blowfish Algorithm:

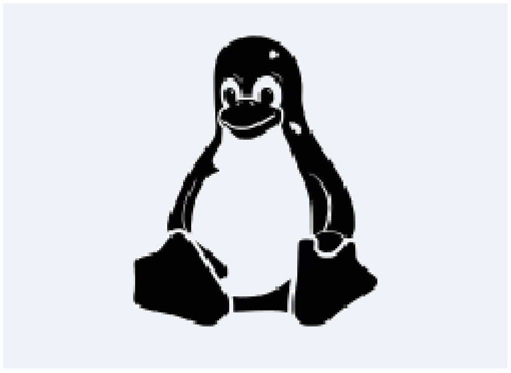


Original Image



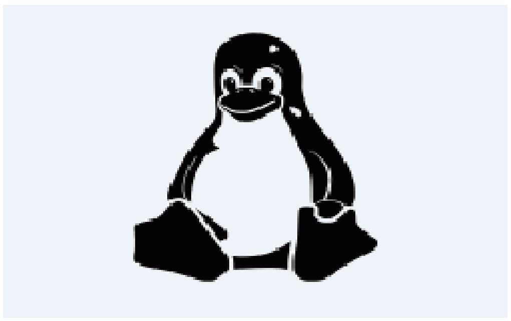
Blowfishencrypt.jpg
This file format can't be opened.

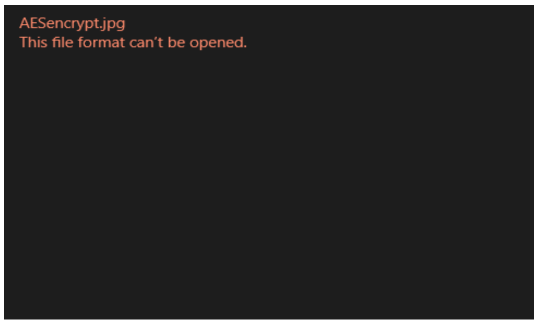
Encrypted Image



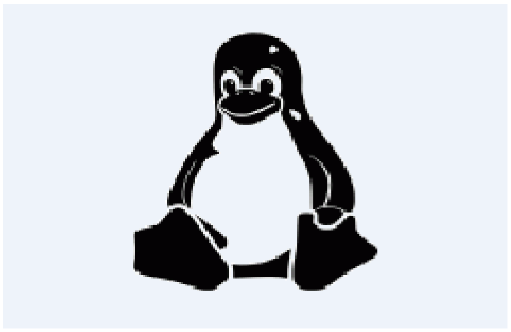
Decrypted Image

AES Algorithm:

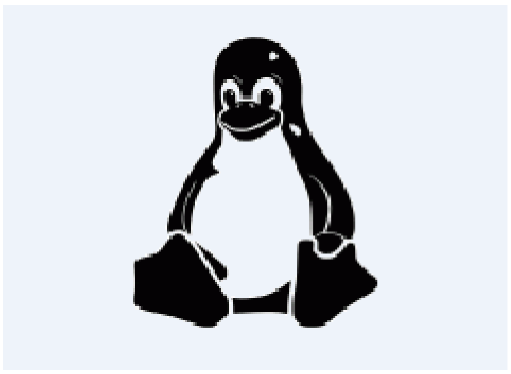

Original Image

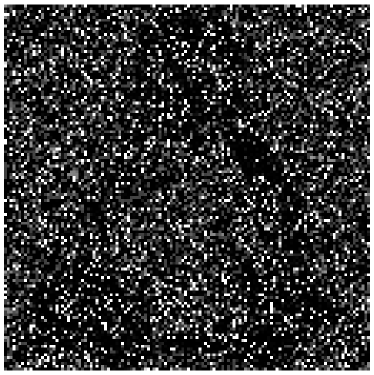

AESencrypt.jpg
This file format can't be opened.

Encrypted Image


Decrypted Image

Chaotic Encryption Image:

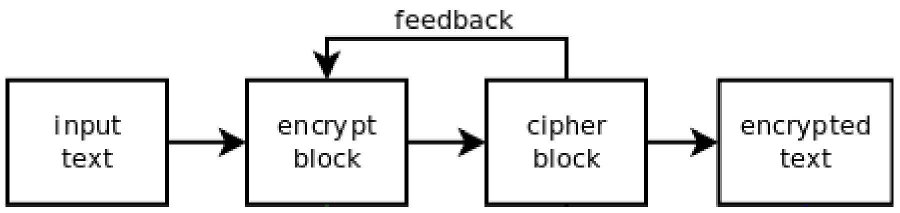

Original Image


Encrypted Image

Procedure and Algorithm Explanation

DES Algorithm (Data Standard Algorithm):

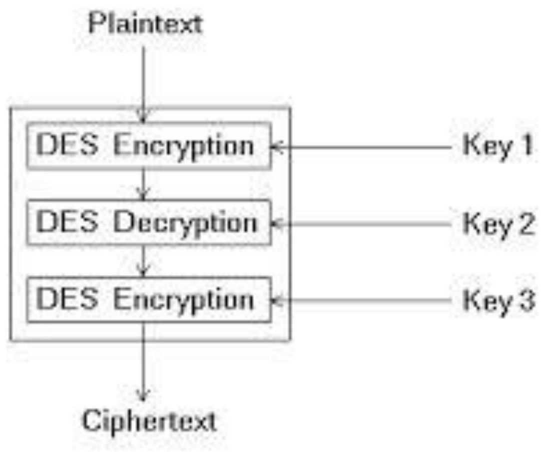
DES (the Data Encryption Standard) is a symmetric block cipher developed by IBM. The algorithm uses a 56-bit key to encipher/decipher a 64-bit block of data. The key is always presented as a 64-bit block, every 8th bit of which is ignored. However, it is usual to set each 8th bit so that each group of 8 bits has an odd number of bits set to 1.



The algorithm is best suited to implementation in hardware, probably to discourage implementations in software, which tend to be slow by comparison. However, modern computers are so fast that satisfactory software implementations are readily available.

DES is the most widely used symmetric algorithm in the world, despite claims that the key length is too short. Ever since DES was first announced, controversy has raged about whether 56 bits is long enough to guarantee security.

The key length argument goes like this. Assuming that the only feasible attack on DES is to try each key in turn until the right one is found, then 1,000,000 machines each capable of testing 1,000,000 keys per second would find (on average) one key every 12 hours. Most reasonable people might find this rather comforting and a good measure of the strength of the algorithm. Those who consider the exhaustive key-search attack to be a real possibility (and to be fair the technology to do such a search is becoming a reality) can overcome the problem by using double or triple length keys. In fact, double length keys have been recommended for the financial industry for many years.



For TDES

Use of multiple length keys leads us to the Triple-DES algorithm, in which DES is applied three times. If we consider a triple length key to consist of three 56-bit keys

K1, K2, K3 then encryption is as follows:

- Encrypt with K1
- Decrypt with K2
- Encrypt with K3

Decryption is the reverse process:

- Decrypt with K3
- Encrypt with K2
- Decrypt with K1

Setting K3 equal to K1 in these processes gives us a double length key K1, K2.
Setting K1, K2 and K3 all equal to K has the same effect as using a single-length (56-bit key).
Thus it is possible for a system using triple-DES to be compatible with a system using single-DES.

AES Algorithm (Advanced Encryption Standard):

Description of the cipher

AES is based on a design principle known as a substitution-permutation network, combination of both substitution and permutation, and is fast in both software and hardware.[1] Unlike its predecessor DES, AES does not use a Feistel network. AES is a variant of Rijndael which has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits.
By contrast, the Rijndael specification *per se* is specified with block and key sizes that may be any multiple of 32 bits, both with a minimum of 128 and a maximum of 256 bits.

AES operates on a 4×4 column-major order matrix of bytes, termed the *state*, although some versions of Rijndael have a larger block size and have additional columns in the state. Most AES calculations are done in a special finite field.

The key size used for an AES cipher specifies the number of repetitions of transformation rounds that convert the input, called the plaintext, into the final output, called the ciphertext. The number of cycles of repetition are as follows:

10 cycles of repetition for 128-bit keys.
12 cycles of repetition for 192-bit keys.
14 cycles of repetition for 256-bit keys.

Each round consists of several processing steps, each containing four similar but different stages, including one that depends on the encryption key itself. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key.

High-level description of the algorithm

1. Key Expansions - Round keys are derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more.

Initial Round

1. Add Round Key - Each byte of the state is combined with a block of the round key using bitwise Xor.

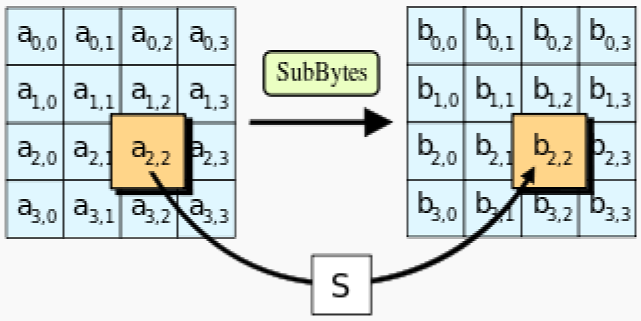
Rounds

2. Sub bytes - A non-linear substitution step where each byte is replaced with another according to a lookup table.
3. Shift Rows - A transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
4. Mix Columns - A mixing operation which operates on the columns of the state, combining the four bytes in each column.
5. Add Round Key

Final Round (no Mix Columns)

6. Sub Bytes
7. Shift Rows
8. Add Round Key.

The SubBytes step


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
SubBytes
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
S

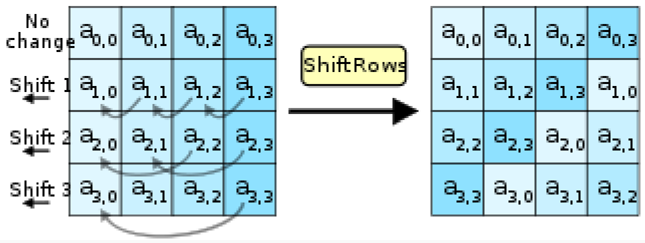
In the SubBytes step, each byte in the state is replaced with its entry in a fixed 8-bit lookup table, *S*; $b_{ij} = S(a_{ij})$.

In the `SubBytes` step, each byte $a_{i,j}$ in the *state* matrix is replaced with a `SubByte` $Sa_{(i,j)}$ using an 8-bit substitution box, the Rijndael S-box. This operation provides the non-linearity in the cipher.

The S-box used is derived from the multiplicative inverse over **GF**(*2*^{8}), known to have good non-linearity properties. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation.

The S-box is also chosen to avoid any fixed points (and so is a derangement), i.e., $S(a_{i,j}) \neq a_{i,j}$, and also any opposite fixed points. While performing the decryption, Inverse SubBytes step is used, which requires first taking the affine transformation and then finding the multiplicative inverse (just reversing the steps used in SubBytes step).

The `ShiftRows` step

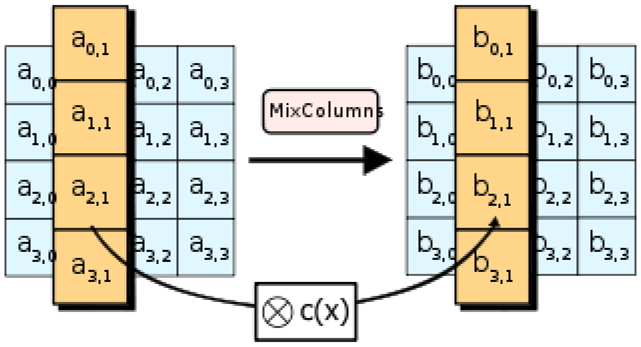


In the `ShiftRows` step, bytes in each row of the state are shifted cyclically to the left. The number of places each byte is shifted differs for each row.

The `ShiftRows` step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. For blocks of sizes 128 bits and 192 bits, the shifting pattern is the same. Row n is shifted left circular by n-1 bytes. In this way, each column of the output state of the `ShiftRows` step is composed of bytes from each column of the input state. (Rijndael variants with a larger block size have slightly different offsets).

For a 256-bit block, the first row is unchanged and the shifting for the second, third and fourth row is 1 byte, 3 bytes and 4 bytes respectively—this change only applies for the Rijndael cipher when used with a 256-bit block, as AES does not use 256-bit blocks. The importance of this step is to avoid the columns being linearly independent, in which case, AES degenerates into four independent block ciphers.

The `MixColumns` step


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
MixColumns
$\otimes c(x)$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$

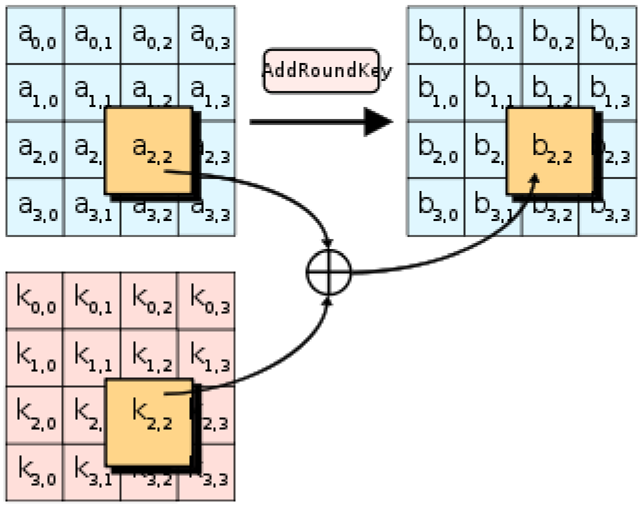
In the `MixColumns` step, each column of the state is multiplied with a fixed polynomial *c(x)*.
In the `MixColumns` step, the four bytes of each column of the state are combined using an invertible linear transformation. The `MixColumns` function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes.
Together with `ShiftRows`, `MixColumns` provides diffusion in the cipher.

During this operation, each column is transformed using a fixed matrix (matrix multiplied by column gives new value of column in the state):

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Matrix multiplication is composed of multiplication and addition of the entries. Entries are 8 bit bytes treated as coefficients of polynomial of order x^7. Addition is simply XOR. Multiplication is modulo irreducible polynomial $x^8+x^4+x^3+x+1$. If processed bit by bit then after shifting a conditional XOR with 0x1B should be performed if the shifted value is larger than 0xFF (overflow must be corrected by subtraction of generating polynomial). These are special cases of the usual multiplication in **GF** (2^8).
In more general sense, each column is treated as a polynomial over **GF**(2^8) and is then multiplied modulo x^4+1 with a fixed polynomial $c(x) = 0x03 \cdot x^3 + x^2 + x + 0x02$. The coefficients are displayed in their hexadecimal equivalent of the binary representation of bit polynomials from **GF**(2)[x]. The `MixColumns` step can also be viewed as a multiplication by the shown particular MDS matrix in the finite field **GF**(2^8).
This process is described further in the article Rijndael mix columns.

The `AddRoundKey` step



In the `AddRoundKey` step, each byte of the state is combined with a byte of the round subkey using the XORoperation (⊕).

In the `AddRoundKey` step, the subkey is combined with the state. For each round, a subkey is derived from the main key using Rijndael's key schedule; each subkey is the same size as the state.

The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.

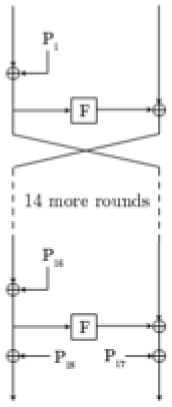
Optimization of the cipher

On systems with 32-bit or larger words, it is possible to speed up execution of this cipher by combining the `SubBytes` and `ShiftRows` steps with the `MixColumns` step by transforming them into a sequence of table lookups. This requires four 256-entry 32-bit tables, and utilizes a total of four kilobytes (4096 bytes) of memory — one kilobyte for each table. A round can then be done with 16 table lookups and 12 32-bit exclusive-or operations, followed by four 32-bit exclusive-or operations in the `AddRoundKey` step.
If the resulting four-kilobyte table size is too large for a given target platform, the table lookup operation can be performed with a single 256-entry 32-bit (i.e. 1 kilobyte) table by the use of circular rotates.
Using a byte-oriented approach, it is possible to combine the `SubBytes`, `ShiftRows`, and `MixColumns` steps into a single round operation.

Blowfish Algorithm:

The Algorithm

Blowfish has a 64-bit block size and a variable key length from 32 bits up to 448 bits. It is a 16-round Feistel cipher and uses large key-dependent S-boxes. In structure it resembles CAST-128, which uses fixed S-boxes.



The Feistel structure of Blowfish

The diagram shows the action of Blowfish. Each line represents 32 bits. The algorithm keeps two subkey arrays: the 18-entry P-array and four 256-entry S-boxes. The S-boxes accept 8-bit input and produce 32-bit output. One entry of the P-array is used every round, and after the final round, each half of the data block is XORed with one of the two remaining unused P-entries. The diagram to the upper right shows Blowfish's F-function. The function splits the 32-bit input into four eight-bit quarters, and uses the quarters as input to the S-boxes. The outputs are added modulo 2^{32} and XORed to produce the final 32-bit output.

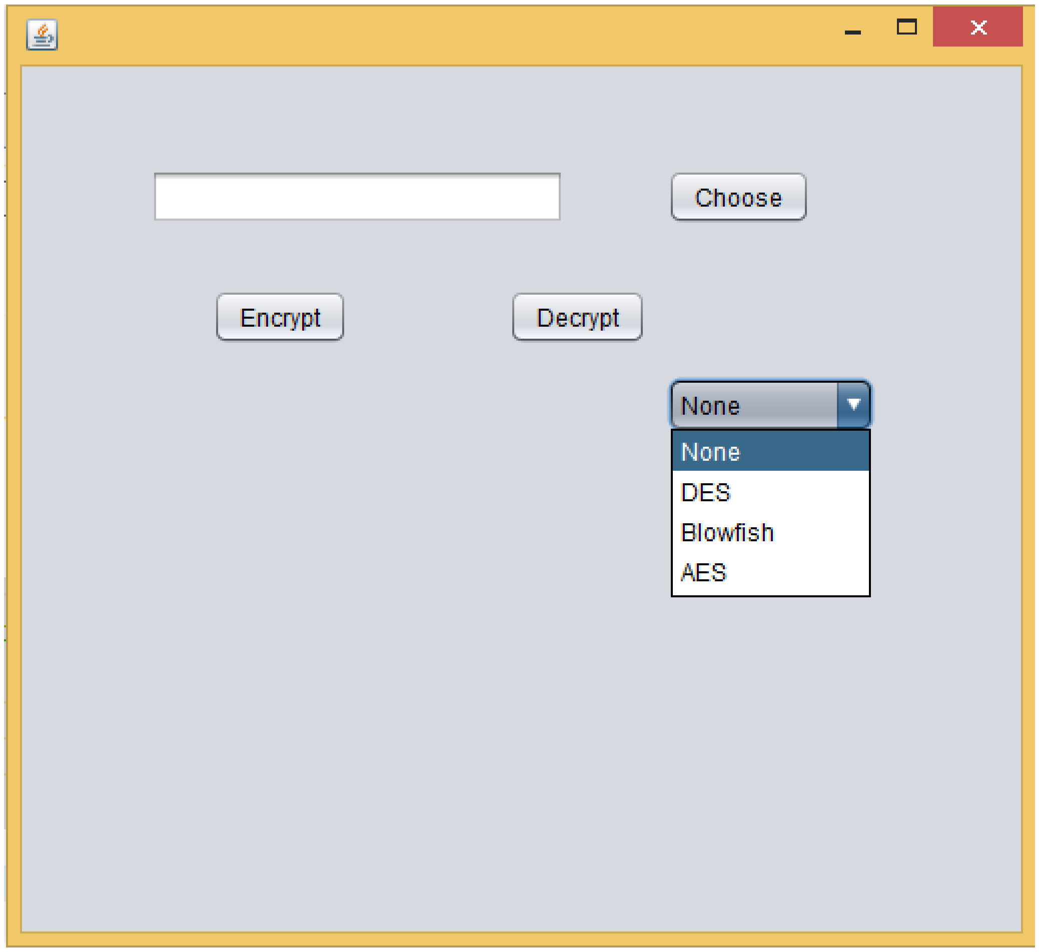
Decryption is exactly the same as encryption, except that P1, P2,….. P18 are used in the reverse order. This is not so obvious because xor is commutative and associative. A common misconception is to use inverse order of encryption as decryption algorithm (i.e. first XORing P17 and P18 to the ciphertext block, then using the P-entries in reverse order).
Blowfish's key schedule starts by initializing the P-array and S-boxes with values derived from the hexadecimal digits of pi, which contain no obvious pattern (see nothing up my sleeve number). The secret key is then, byte by byte, cycling the key if necessary, XORed with all the P-entries in order. A 64-bit all-zero block is then encrypted with the algorithm as it stands. The resultant ciphertext replaces P_1 and P_2. The same ciphertext is then encrypted again with the new

subkeys, and the new ciphertext replaces P_3 and P_4. This continues, replacing the entire P-array and all the S-box entries. In all, the Blowfish encryption algorithm will run 521 times to generate all the subkeys - about 4KB of data is processed.

Because the P-array is 576 bits long, and the key bytes are XORed through all these 576 bits during the initialization, many implementations support key sizes up to 576 bits. While this is certainly possible, the 448 bits limit is here to ensure that every bit of every subkey depends on every bit of the key, as the last four values of the P-array don't affect every bit of the ciphertext. This point should be taken in consideration for implementations with a different number of rounds, as even though it increases security against an exhaustive attack, it weakens the security guaranteed by the algorithm. And given the slow initialization of the cipher with each change of key, it is granted a natural protection against brute-force attacks, which doesn't really justify key sizes longer than 448 bits.

Implementation

The implementation of the logic is done in two parts, Three algorithms are made in Java names DES (Data Encryption Standard), AES (Advanced Encryption Standard), and Blowfish.



The java code includes numerous library files which are dedicated only to image encryption. The used ones is javax.crypto.*;
It has three modules for each DES, AES and Blowfish.
Chaotic Algorithm:

TESTS

NPCR and UACI Randomness Tests for Image Encryption

The number of changing pixel rate (NPCR) and the unified averaged changed intensity (UACI) are two most common quantities used to evaluate the strength of image encryption algorithms/ciphers with respect to differential attacks. Conventionally, a high NPCR/UACI score is usually interpreted as a high resistance to differential attacks. However, it is not clear how high NPCR/UACI is such that the image cipher indeed has a high security level. In this paper, we approach this problem by establishing a mathematical model for ideally encrypted images and then derive expectations and variances of NPCR and UACI under this model. Further, these theoretical values are used to form statistical hypothesis NPCR and UACI tests. Critical values of tests are consequently derived and calculated both symbolically and numerically. As a result, the question of whether a given NPCR/UACI score is sufficiently high such that it is not discernible from ideally encrypted images is answered by comparing actual NPCR/UACI scores with corresponding critical values. Experimental results using the NPCR and UACI randomness tests show that many existing image encryption methods are actually not as good as they are purported, although some methods do pass these randomness tests

NPCR & UACI quantitative and qualitative scores for the strength against possible differential attacks of image ciphers.

RESULTS

NPCR and UACI Randomness Tests

FOR AES

```
NPCR_Score: 0.142857142857143
NPCR pVal: 3.486500682384463e-272
NPCR_dist : [0.991803278688525 5.806810765155296e-04]
uaci_score: 0.005903187721370
uaci_pVal: 2.014111622292874e-07
uaci_dist: [0.336065573770492 0.004034386980687]
```

For DES

```
NPCR_Score: 0.142857142857143
npcr_pVal: 3.486500682384463e-272
npcr_dist: [0.991803278688525 5.806810765155296e-04]
uaci_score: 0.005903187721370
uaci_pVal: 2.014111622292874e-07
uaci_dist: [0.336065573770492 0.004034386980687]
```

For BlowFish

```
NPCR_Score: 0.105263157894737
npcr_pVal: 3.24567 0634384463e-311
npcr_dist: [1x2 double]
uaci_score: 0.004349717268378
uaci_pVal: 1.172100249417747e-09
uaci_dist: [1x2 double]
```

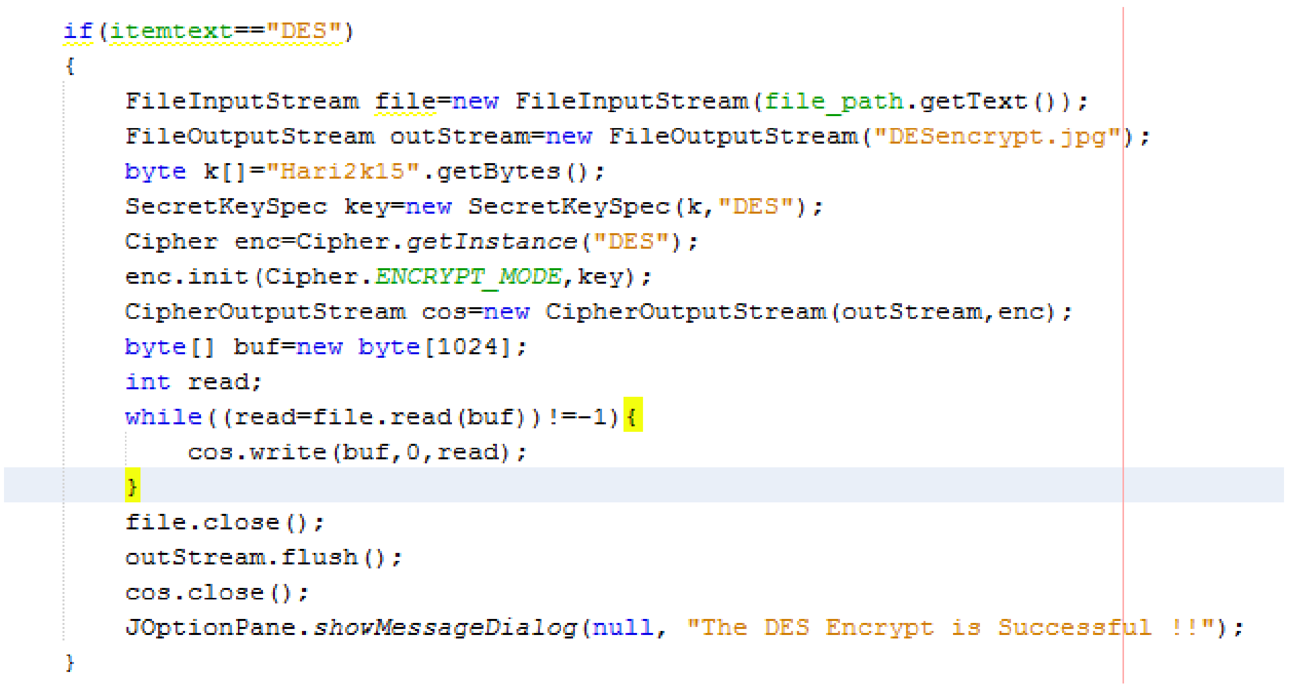
For Chaotic

```
NPCR_Score: 0.065268310856737
npcr_pVal: 5.270287176004828e-318
npcr_dist: [0.991803278688525 6.774612559347845e-04]
uaci_score: 0.002372115268771
uaci_pVal: 9.657801139329518e-07
uaci_dist: [0.336065573770492 0.004706784810801]
```

Algorithm

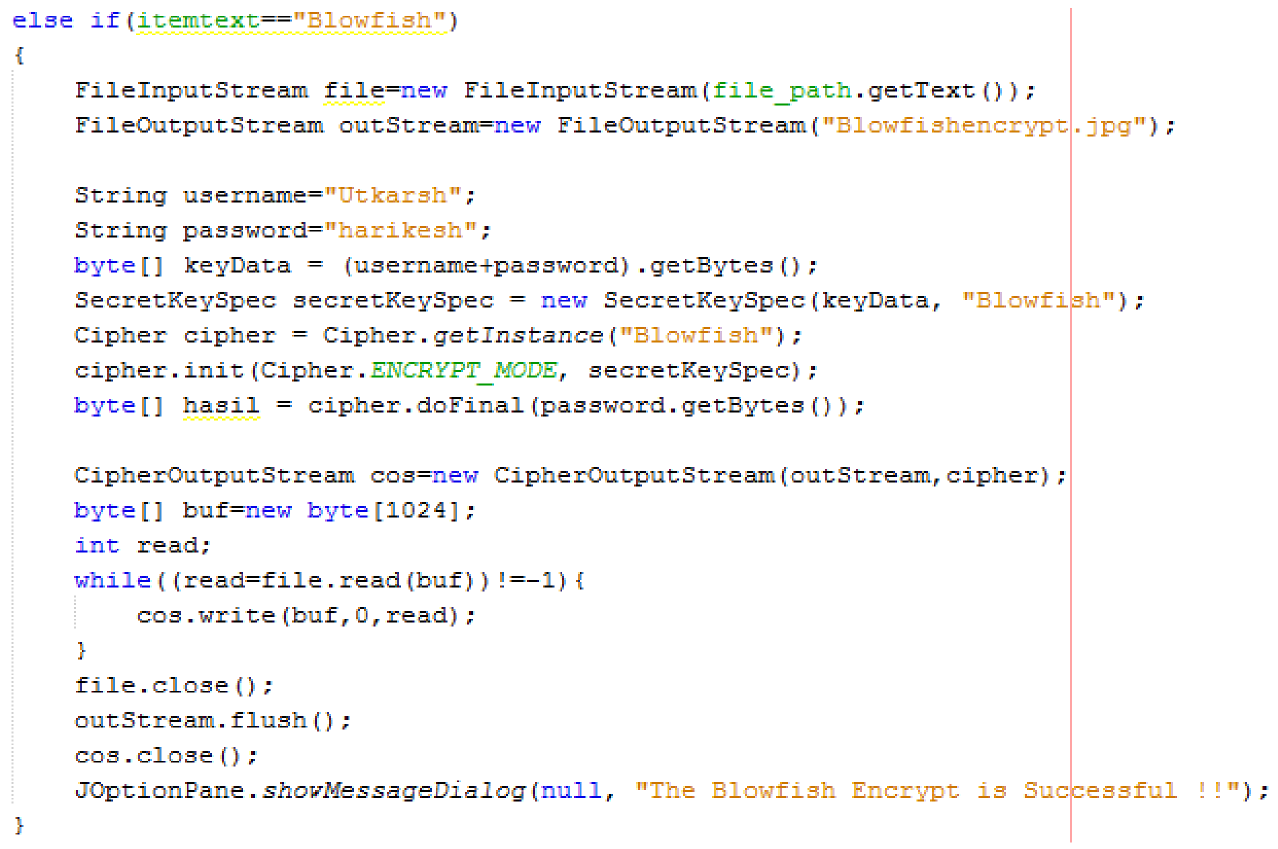
The code in Java is divided into three parts, The first one is DES its Algorithm is as follows:

DES Code for encryption



```
if(itemtext=="DES")
{
    FileInputStream file=new FileInputStream(file_path.getText());
    FileOutputStream outStream=new FileOutputStream("DESencrypt.jpg");
    byte k[]="Hari2k15".getBytes();
    SecretKeySpec key=new SecretKeySpec(k,"DES");
    Cipher enc=Cipher.getInstance("DES");
    enc.init(Cipher.ENCRYPT_MODE,key);
    CipherOutputStream cos=new CipherOutputStream(outStream,enc);
    byte[] buf=new byte[1024];
    int read;
    while((read=file.read(buf))!=-1){
        cos.write(buf,0,read);
    }
    file.close();
    outStream.flush();
    cos.close();
    JOptionPane.showMessageDialog(null, "The DES Encrypt is Successful !!");
}
```

Blowfish Code for encryption

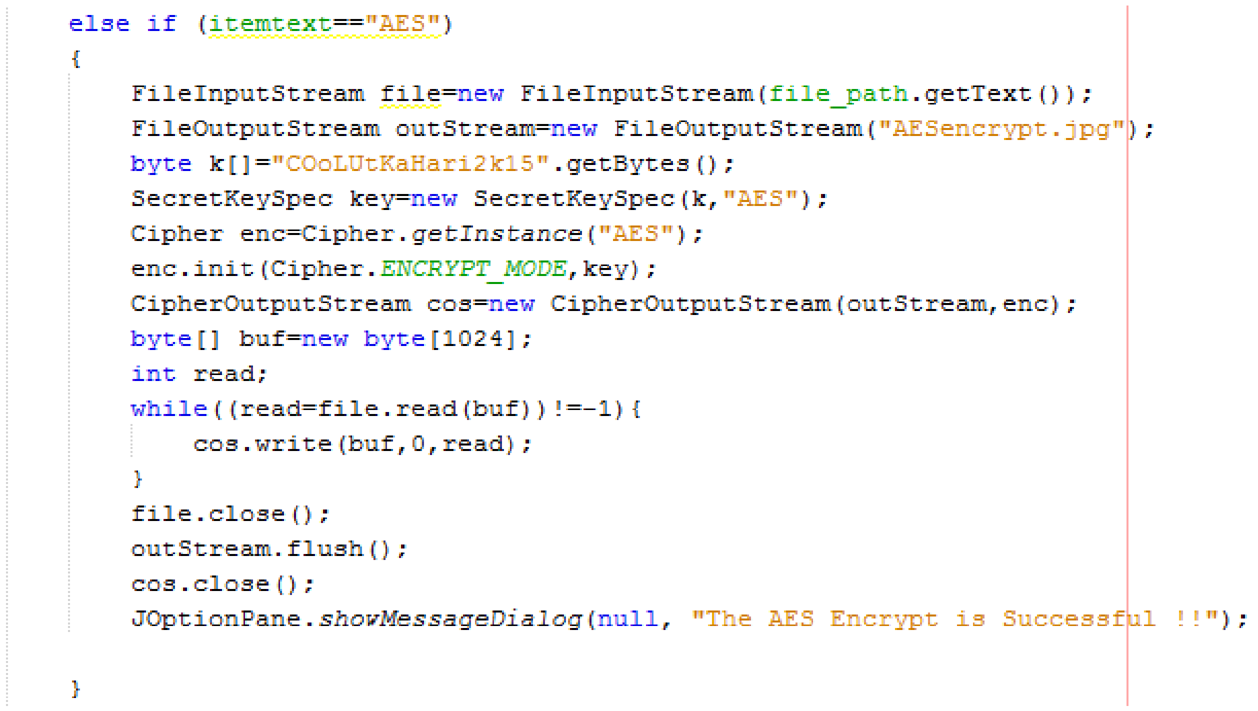


```
else if(itemtext=="Blowfish")
{
    FileInputStream file=new FileInputStream(file_path.getText());
    FileOutputStream outStream=new FileOutputStream("Blowfishencrypt.jpg");

    String username="Utkarsh";
    String password="harikesh";
    byte[] keyData = (username+password).getBytes();
    SecretKeySpec secretKeySpec = new SecretKeySpec(keyData, "Blowfish");
    Cipher cipher = Cipher.getInstance("Blowfish");
    cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec);
    byte[] hasil = cipher.doFinal(password.getBytes());

    CipherOutputStream cos=new CipherOutputStream(outStream,cipher);
    byte[] buf=new byte[1024];
    int read;
    while((read=file.read(buf))!=-1){
        cos.write(buf,0,read);
    }
    file.close();
    outStream.flush();
    cos.close();
    JOptionPane.showMessageDialog(null, "The Blowfish Encrypt is Successful !!");
}
```

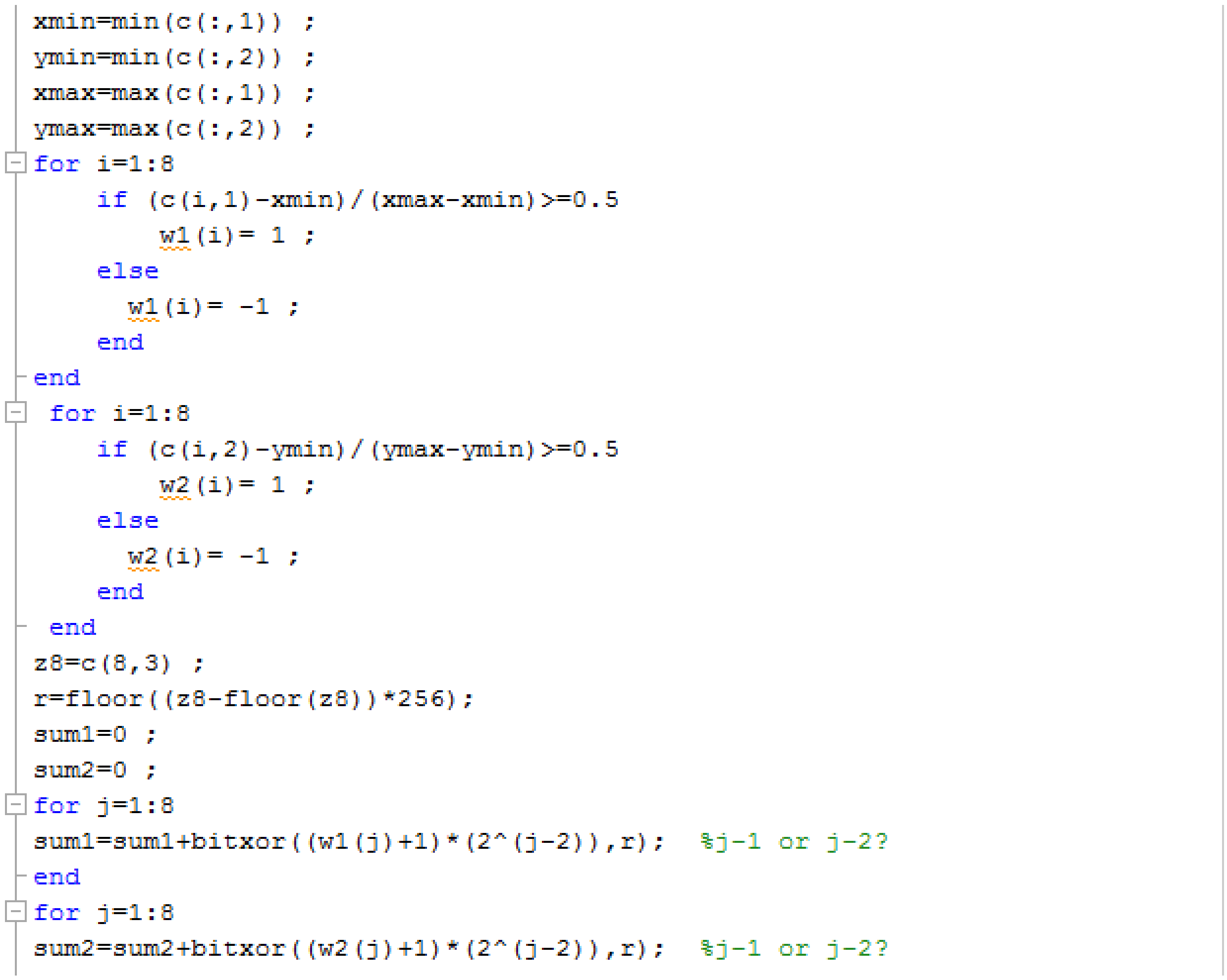
AES Code for encryption



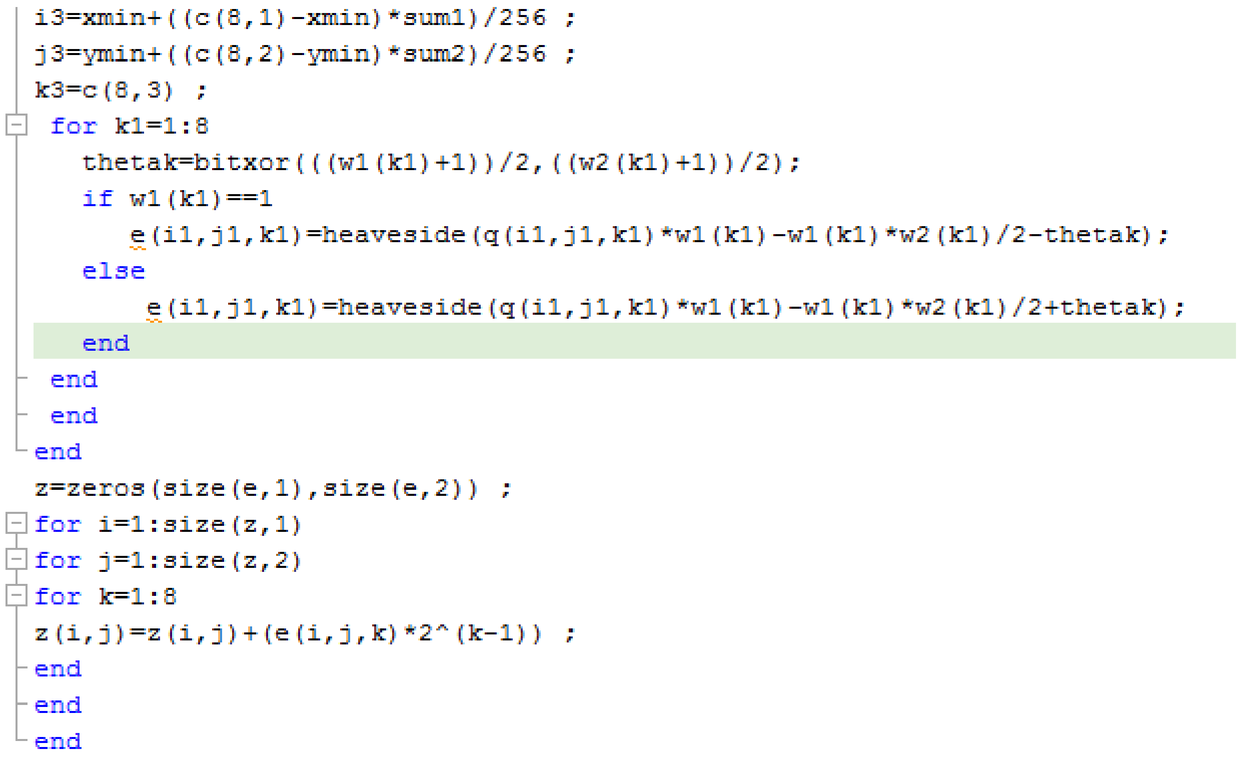
```java
else if (itemtext=="AES")
{
    FileInputStream file=new FileInputStream(file_path.getText());
    FileOutputStream outStream=new FileOutputStream("AESencrypt.jpg");
    byte k[]="COoLUtKaHari2k15".getBytes();
    SecretKeySpec key=new SecretKeySpec(k,"AES");
    Cipher enc=Cipher.getInstance("AES");
    enc.init(Cipher.ENCRYPT_MODE,key);
    CipherOutputStream cos=new CipherOutputStream(outStream,enc);
    byte[] buf=new byte[1024];
    int read;
    while((read=file.read(buf))!=-1){
        cos.write(buf,0,read);
    }
    file.close();
    outStream.flush();
    cos.close();
    JOptionPane.showMessageDialog(null, "The AES Encrypt is Successful !!");

}
```

The Fourth algorithm is made in Matlab named
Chaotic image encryption using perceptron model.



```
xmin=min(c(:,1)) ;
ymin=min(c(:,2)) ;
xmax=max(c(:,1)) ;
ymax=max(c(:,2)) ;
for i=1:8
    if (c(i,1)-xmin)/(xmax-xmin)>=0.5
        w1(i)= 1 ;
    else
      w1(i)= -1 ;
    end
end
 for i=1:8
    if (c(i,2)-ymin)/(ymax-ymin)>=0.5
        w2(i)= 1 ;
    else
      w2(i)= -1 ;
    end
 end
z8=c(8,3) ;
r=floor((z8-floor(z8))*256);
sum1=0 ;
sum2=0 ;
for j=1:8
sum1=sum1+bitxor((w1(j)+1)*(2^(j-2)),r);  %j-1 or j-2?
end
for j=1:8
sum2=sum2+bitxor((w2(j)+1)*(2^(j-2)),r);  %j-1 or j-2?
```



```
i3=xmin+((c(8,1)-xmin)*sum1)/256 ;
j3=ymin+((c(8,2)-ymin)*sum2)/256 ;
k3=c(8,3) ;
 for k1=1:8
   thetak=bitxor(((w1(k1)+1))/2,((w2(k1)+1))/2);
   if w1(k1)==1
      e(i1,j1,k1)=heaveside(q(i1,j1,k1)*w1(k1)-w1(k1)*w2(k1)/2-thetak);
   else
       e(i1,j1,k1)=heaveside(q(i1,j1,k1)*w1(k1)-w1(k1)*w2(k1)/2+thetak);
   end
 end
 end
end
z=zeros(size(e,1),size(e,2)) ;
for i=1:size(z,1)
for j=1:size(z,2)
for k=1:8
z(i,j)=z(i,j)+(e(i,j,k)*2^(k-1)) ;
end
end
end
```