Indian Institute of Science Education and Research, Mohali

Non-Linear Dynamics

IDC 402

Using Novel Chaotic Oscillators for Image Encryption

Aprameyan Desikan

MS20175

November 2023

1 Introduction to Image Encryption

With advance in technology, information security has started playing a crucial role a variety of fields. The process of encryption requires a random key to be generated using a chaotic oscillator. As a result of the chaotic map being highly sensitive to initial conditions, the key will be hard to generate by a third party even if aware of the flow. Such systems are therefore used to design modern cryptosystems for data security. Furthermore, certain permutations and substitution methods are also carried to increase the randomness in the encryption.
This term paper is specifically trying to replicate and understand the encryption scheme based on the paper "A Novel Chaos-Based Cryptography Algorithm and Its Performance Analysis" published in MDPI, Mathematics (Link).
We start by first analysing our hyper-chaotic oscillator by understanding its evolution, its bifurcation diagram and its Lyapunov exponents. We then start framing out the encryption scheme and apply it on a specific image and trace out its evolution over the process. We will also apply the decryption scheme to complete the process.

2 The Chaotic Oscillator

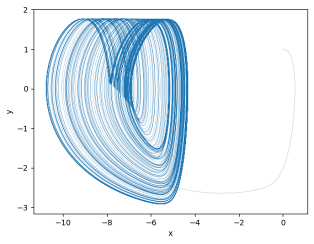
The chaotic oscillator designed by the authors is given by;

$$\begin{aligned} \dot{x} &= y \\ \dot{y} &= x \\ \dot{z} &= ax - y - 0.7z – 0.9y^2 - 0.7z^2 + 0.3xy + cxz + byz \end{aligned} \tag{2.1}$$

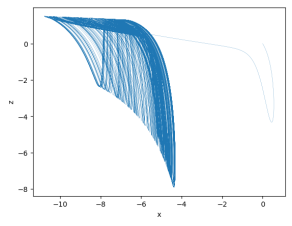
The parameters used by the paper and for the sake of recreation is $a = -0.7$, $a = 2.7$, $c = 0.3$ with an initial condition at $(0, 1, 0)$.

```python
def Novel(x0, y0, z0, a, b, c, num_steps):
    dt= 0.01
    x = np.empty(num_steps+1)
    y = np.empty(num_steps+1)
    z = np.empty(num_steps+1)

    x[0] = x0
    y[0] = y0
    z[0] = z0

    for i in range(num_steps):
        x[i+1] = x[i] + y[i]*dt
        y[i+1] = y[i] + z[i]*dt
        z[i+1] = z[i] + (a*x[i] - y[i] - 0.7*z[i] - 0.9*(y[i])**2 - 0.7*(z[i])**2 + 0.3*x[i
    ]*y[i] + c*x[i]*z[i] + b*y[i]*z[i])*dt

return x, y, z
```

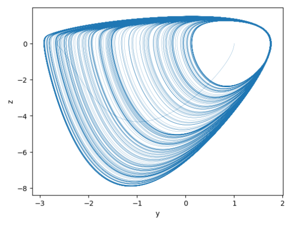
This function implements the flow for the chaotic oscillator and returns the array of x, y and z values that the oscillator has gone through, given the initial conditions and the number of steps for running (iterating over multiple small time steps). With this in hand, we can plot the evolution for the initialisation.



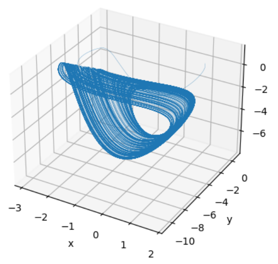
(a) y-x variation



(b) z-x variation



(c) z-y variation

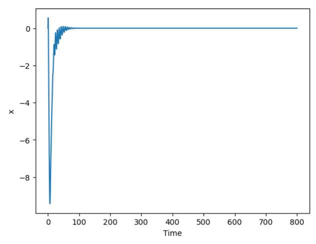


(d) Complete picture of the Chaotic Oscillator

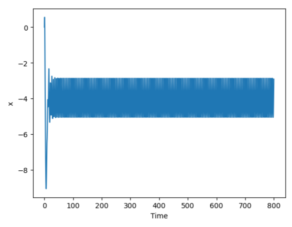
2.1 Analysis

2.1.1 Bifurcation Diagram

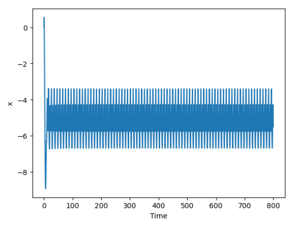
In order to analyse, we plot out the time series to see the stability while changing the parameters, namely, a, b and c. The bifurcation diagram can be understood by observing the variation of the variables as a function of time, and finding their peaks or using a Poincare section.



(e) 1-stable point



(f) 2-stable point



(g) 4-stable point

As seen, by varying 'a' values we see such a trend and it can be used to find the bifurcation diagram as given in the paper.

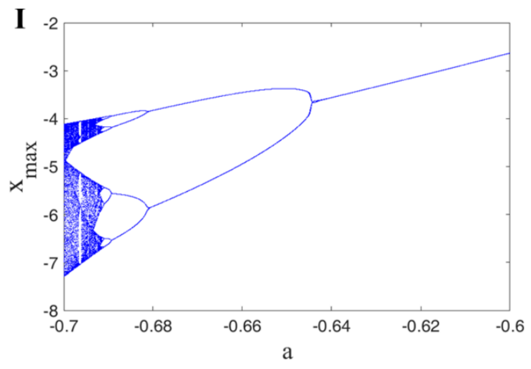


Fig: Taken from the paper. Shows the x-max values for varying 'a' values

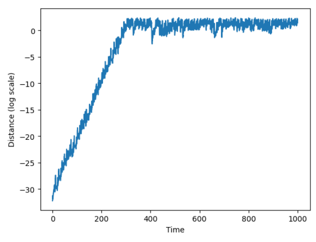
Similar bifurcation diagrams can be drawn for varying 'b' values and 'c' values.

2.1.2 Lyapunov Exponent

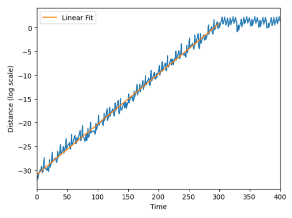
The idea of the Lyapunov exponent is to find the rate of exponential separation as a result of the deviation in initial conditions. For the flow in a chaotic oscillator, this is done by first calculating the logarithm of the Euclidean distances between the chaotic oscillator with certain initial conditions and another path taken by slightly deviated initial conditions. We plot this distance as a function of time and find the slope of the graph before it saturates at the time of oscillation. The slope is given to be the Lyapunov exponent by definition.

```python
x, y, z = Novel(0, 1, 0, -0.702, 2.7, 0.3, 100000) # Original path
x_dev, y_dev, z_dev = Novel(10e-15, 1, 0, -0.702, 2.7, 0.3, 100000) # Deviated Path
delt = [] # To collect the distances
for i in range(100001):
    delt.append(np.log(np.sqrt((x_dev[i]-x[i])**2 + (y_dev[i]-y[i])**2 + (z_dev[i]-z[i])**2)
    )) # Euclidean distance appended at each iteration

```

Using this code to initialise two arrays with varying evolution, we collect the logrithms of the distances and plot it over time.



(h) Evolution of deviation over time



(i) Linear fit made for the deviation

The slope was found to be 0.106 for this value of a = -0.702 One can plot the Lyapunov exponent as a function of the varying values of 'a' to analyse the trend of the Lyapunov exponents. **Note:** One could also analyse the Lyapunov exponents individually for each variable by just calculating the slope of the logarithmic distance over just one of the axes, instead of the total Euclidean distance. The plot on the paper calculates the Lyapunov exponent using the Wolf method and obtains the following.

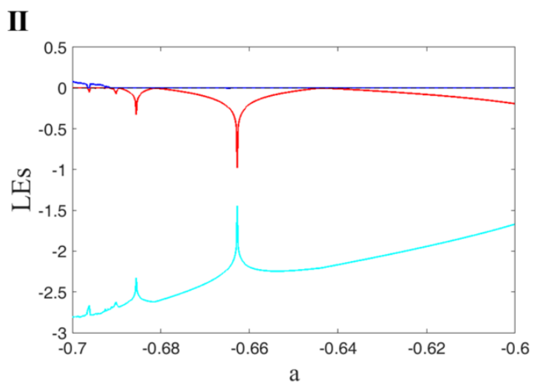


Fig: Taken from the paper. Shows the trend for the three Lyapunov exponents as a function of the 'a' parameter

Similar trends can be obtained for the variation in 'b' and 'c'. With the analysis of the chaotic oscillator, we can now get into the understanding of the encryption scheme.

2.2 The Encryption Scheme

The scheme consists of using Pseudo-Random number generators and S-boxes (substitution-box) along with certain permutation algorithms to make up a robust encryption system. We will first go through the processes one by one and then analyse the encryption scheme along with the evolution of the image.

2.2.1 PRNG Mechanism

The way we obtain the PRNG sequence is by doing the following;

- Run the chaotic oscillator with certain initial values and set of parameters which will return the sequences X, Y, Z (showing the evolution of each variable).
- Transform the signal into integer number in the range of 0 to 255 in the following fashion

$$\begin{aligned} SeqX &= fix(X \times 10^{12} mod 256) \\ SeqY &= fix(Y \times 10^{12} mod 256) \\ SeqZ &= fix(Z \times 10^{12} mod 256) \end{aligned} \tag{2.2}$$

  where fix function takes floors the number to its integer value close to 0, i.e. fix(5.68) = 5
- Now the PRNG sequence is given by;

$$PRNG = SeqX \oplus SeqY \oplus SeqZ \tag{2.3}$$

  where we just XOR the sequences element wise.

The code for the mechanism is given by;

```
def seq(X): # applies the fix function along with the modulus
    return np.floor((X*10e12)%256)
def PRNG(x_0, y_0, z_0, a, b, c, R, C, N): # Returns the the PRNG sequence, for
                                            # a given set of initial conditions
    X, Y, Z = Novel(x_0, y_0, z_0, a, b, c, (R*C*N)-1)  # Created sequence of size = no. of pixels
    Sx = seq(X) # Applies the required function to all sequences
    Sy = seq(Y)
    Sz = seq(Z)
    Sx = Sx.astype(np.int32) # conversion to numpy array to apply XOR in a simpoler manner
    Sy = Sy.astype(np.int32)
    Sz = Sz.astype(np.int32)
    return Sx^Sy^Sz # Returns the PRNG sequence

```

2.2.2 S-Box Mechanism

The mechanism works as folllows;

- Obtain PRNG sequence as proposed before
- Collect the first 256 dissimilar elements from the sequence.

The code for the mehcanism is as follows;

```
def unique(H, R): # Function to pick first R dissimilar elements
    arr = []
    for i in range(len(H)):
        k=0
        for j in range(i):
            if H[i]==H[j]:
                k+=1
        if k==0:
            arr.append(H[i])
        if len(arr) == R:
            break
    return arr
S1 = PRNG(x_n, y_n, z_n, -0.7, 2.7, 0.3, R, C, N) # Creates the PRNG sequence
SB = unique(S1, 256) # Collects the first 256 elements

```

2.2.3 Encryption Algorithm

To give an overview, we first take the plain image and substitute it using the PRNG sequence obtained with certain initial conditions. With some information about the substituted image, we then update the initial conditions we first used. Using the updated initial conditions, we generate 3 sequences and utilize the first and third sequence to shuffle the pixels along the rows, and the second and third sequence to shuffle the pixels along the columns. Finally, the S-box is used to substitute the permuted image to give the final cipher image.
We start by first taking our plain image;

```
import cv2 # package to
img = cv2.imread(r"\Path")
plt.imshow(img)

```

Ouput:

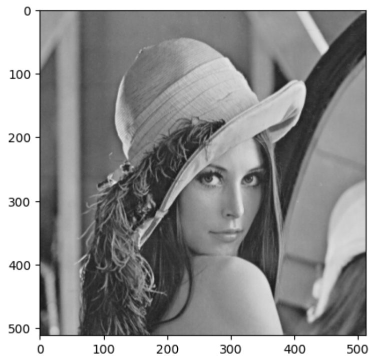
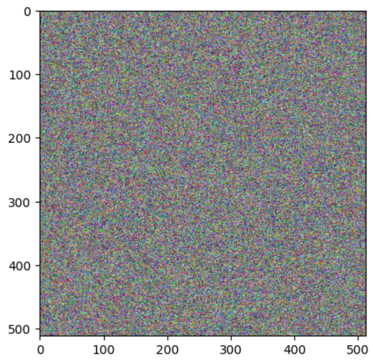


Fig 1: Standard picture of Lena used for testing encryption schemes

```
[R, C, N] = img.shape # saving the shape of the image, row and column pixels along with
    pixel values in terms of R G B
x_0, y_0, z_0, a, b, c = -0.2851, 0.7692, 0.617, -0.7, 2.7, 0.3 # initial condition for the
    chaotic oscillator
# 1. Generating the PRNG sequence to carry out subsitution
K = PRNG(x_0, y_0, z_0, -0.7, 2.7, 0.3, R, C, N) # using the previously defined PRNG
    function to generate sequence
K1 = K.reshape(R, C, N) #  Reshaping to perform XOR on image
SG = img^K1 # Subsitution carried out on the plain image with the K matrix

```

Ouput:



Now, we obtain the **information from the substituted image** to update the initial conditions of our chaotic oscillator for the permutation process.

```
s = 0
for i in range(R):
    for j in range(C):
        for k in range(N):
            n = (SG[i][j][k])
            s += n
beta = (s%512)/512 # beta has extracted some information from the substituted image
x_n = (x_0 + beta)/2 # updated initial values with beta
y_n = (y_0 + beta)/2
z_n = (z_0 + beta)/2

```

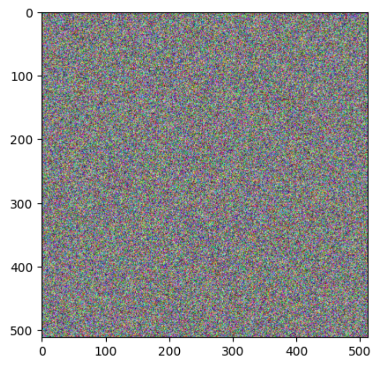
Now, we will carry out the **permutation** process using the sequences generated with the updated initial conditions of the chaotic oscillator.

```
X, Y, Z = Novel(x_n, y_n, z_n, a, b, c, (R*C*N)-1)

# For row shuffling
H = np.floor(((X+Z)*10e12)%R) # Creating a sequence with X and Z obtained from the chaotic
    oscillator
PerH = unique(H, R) # Collecting the first 512 (no. of rows) dissimilar elements from the H
    sequence
PerH = np.array(PerH).astype(np.int32) # converting datatype of array for future purposes of
     image plotting

# For column shuffling
W = np.floor(((Y+Z)*10e12)%C) # Creating a sequence with Y and Z obtained from the chaotic
    oscillator
PerW = unique(W, C) # Collecting the first 512 (no. of columns) dissimilar elements from the
     W sequence
PerW = np.array(PerW).astype(np.int32)

PerG = np.zeros(shape=[R, C, N]) # Setting up the permuted matrix
for t in range(R):
    for u in range(C):
        PerG[t][u][:] = SG[PerH[t]][PerW[u]][:] # Carrying out the permutation
PerG = PerG.astype(np.int32)

```

The outputs at this point don't have any significant differences in terms of their visual appeal.
We finally have to carry out the **S-box mechanism**

```
S1 = PRNG(x_n, y_n, z_n, -0.7, 2.7, 0.3, R, C, N)
SB = unique(S1, 256) # Obtaining the elements of the S-box
CG = np.zeros(shape=[R, C, N]) # Setting up the cipher image array
for t in range(R):
    for u in range(C):
        for v in range(N):
            CG[t, u, v] = SB[PerG[t, u, v]] # carrying out the S-box substitution

```

Final Output:



The final cipher image has been obtained.

2.2.4 The Decryption Scheme

The decryption scheme is essentially the reverse of the encryption scheme. It is to note that before the encryption is carried out, the key parameters used are shared between the sender and receiver through some standard cryptographic channels. After encryption, the beta value is also shared through some secure channel.

2.2.5 Decryption Algorithm

We start be reversing the **S-box process**

```
[R, C, N] = CG_final.shape # Obtain initial values
x_g, y_g, z_g = Novel(x_n, y_n, z_n, a, b, c, (R*C*N)-1) # use shared initial coniditions
    for running the chaotic oscillator
S1_g = PRNG(x_n, y_n, y_n, -0.7, 2.7, 0.3, R, C, N)
SB_g = unique(S1, 256) # Create S-box
PerG_g = np.zeros(shape=[R, C, N], dtype=np.uint64) # Set up the matrix storing the S-box
    undone image
for t in range(R):
    for u in range(C):
        for v in range(N):
            PerG_g[t, u, v] = np.where(SB_g == CG[t, u, v])[0][0] # reverse S-box
    substitution

```

Now, we have to reverse the **permutation** that was carried out. We similarly set up the individual permutation matrices for the rows and columns.

```
    #  Carry out the same process as before
H_g = np.floor(((x_g+z_g)*10e12)%R)
PerH_g = unique(H, R)
PerH_g = np.array(PerH_g).astype(np.int32)
W_g = np.floor(((y_g+z_g)*10e12)%C)
PerW_g = unique(W, C)
PerW_g = np.array(PerW_g).astype(np.int32)

SG_g = np.zeros(shape=[R, C, N], dtype=np.uint64)
for t in range(R):
    for u in range(C):
        SG_g[PerH_g[t], PerW_g[u], :] = PerG_g[t, u, :]

```

Finally, we use the beta value obtained to find the initial conditions to generate the PRNG sequence and **apply a final XOR** to obtain back the plain image.

```
K_g = PRNG(x_0, y_0, z_0, -0.7, 2.7, 0.3, R, C, N)
K1_g = K_g.reshape(R, C, N)
K1_g = K1_g.astype("uint64")
DG = SG_g^K1_g

```

Final Output:

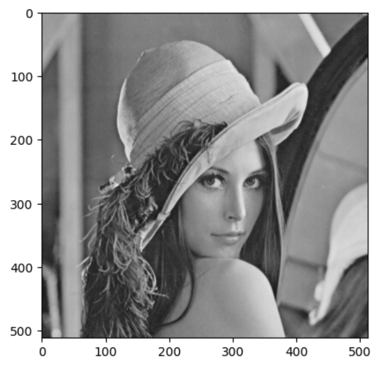


Fig 1: Original plain image obtained

The code, which can be found at GitHub Link can be used by the reader and replicated for further proof.

3 Remarks

The paper has further delved into testing the efficiency of the encryption scheme, namely based on the encryption time and its ability to resist attacks through brute force, statistical cryptanalysis, differential cryptananlysis, etc. Further notes on the performance analysis of the pseudo-random number generator and the S-box scheme was also discussed. The cryptosystem scheme was evaluated with a variety of tests including the histogram test, information entropy and key sensitivity.

Histogram Test: A good encryption scheme must ensure the uniformity for distinct cipher images. Meaning, if you consider different images and perform the said scheme, one must not be able to find patterns based on their histograms (of their pixel values) that could help differentiate them. The plain image would obviously have a higher variance in its histogram plot whereas it is expected that the cipher images must have a lower one, denoting higher uniformity.
Information Entropy: A quantitative method to calculate the randomness in the cipher image and used to evaluate encryption.
Key sensitivity: This is a test to evaluate how slight of a difference could ruin the decryption scheme. By testing out different deviations in the key, the decryption scheme is applied to see how close the cipher image can get to the original. For a good encryption scheme, slight modifications drastically affect the final result.

4 Conclusion

We have thus studied the use of one such hyper-chaotic oscillator in an image encryption scheme. We first analysed the chaotic behavious of the oscillator itself by analysing its bifurcation diagram and Lyapunov exponents for varying parameters. Further, we studied the encryption scheme through the PRNG mechanism, S-box mechanism and permutation methods. We then observed the evolution of the image through the process and finally understood few evaluation processes to test the encryption scheme itself.