# Efficient Encryption Approach Using Discrete-Time Chaotic Maps for Securing the Internet of Robotic Things

*Submitted by*

## AAKASH S

**ARMAAN SAHOO**

**ANIRUDHAN S R**

# List of Figures

|  |  |  |
| --- | --- | --- |
| **Fig No.** | **Title** | **Page No.** |
| 1.1 | Process Flow | 4 |
| 1.2 | Client Server Model | 4 |
| 4.1 | Cubic Map’s Time Series | 20 |
| 4.2 | Cubic Map’s Bifurcation Diagram | 20 |
| 4.3 | Cubic Map’s Map Function | 21 |
| 4.4 | Cubic Map’s Lyapunov Exponent | 21 |
| 4.5 | Ricker’s Population Model’s Time Series | 22 |
| 4.6 | Ricker’s Population Model’s Bifurcation Diagram | 22 |
| 4.7 | Ricker’s Population Model’s Map Function | 23 |
| 4.8 | Ricker’s Population Model’s Lyapunov Exponent | 23 |
| 4.9 | Input Image | 24 |
| 4.10 | Encrypted Cubic Image | 24 |
| 4.11 | Ricker’s Horizontal Encrypted Image | 24 |
| 4.12 | Ricker’s Vertical Encrypted Image | 24 |
| 4.13 | Ricker’s Horizontal Decrypted Image | 24 |
| 4.14 | Ricker’s Vertical Decrypted Image | 24 |

|  |  |  |
| --- | --- | --- |
| 4.15 | Cubic Decrypted Image | 25 |
| 4.16 | Input Image | 25 |
| 4.17 | Raspberry Pi With Webcam | 25 |
| 4.18 | Client Side Setup | 25 |
| 4.19 | Red Channel (Original) | 26 |
| 4.20 | Red Channel(Encrypted) | 26 |
| 4.21 | Blue Channel (Original) | 26 |
| 4.22 | Blue Channel(Encrypted) | 26 |
| 4.23 | Green Channel (Original) | 26 |
| 4.24 | Green Channel(Encrypted) | 26 |

**List of Tables**

|  |  |  |
| --- | --- | --- |
| **Table No.** | **Title** | **Page No.** |
| 4.1 | NPCR,UACI,Entropy Testing | 27 |
| 4.2 | Vertical Correlation | 27 |
| 4.3 | Horizontal Correlation | 27 |

# Abbreviations

|  |  |
| --- | --- |
| IoRT | Internet of Robotic Things |
| NPCR | Number of Pixels Change Rate |
| UACI | Unified Average Changing Intensity |
| PRNG | Pseudo Random Number Generator |
| GPU | Graphical Processing Unit |
| CPU | Central Processing Unit |
| NIST | National Institute of Standards and Technology |
| RGB | Red Green Blue |

**Notations**

## English Symbols (in alphabetical order)

A Initial parameter for sensitiveness

𝑥 Nth element of Pseudo random series

𝑛

e Euler’s Number

# ABSTRACT

Securing data in robots connected to the Internet of Robotic Things (IoRT) is a critical challenge due to the increasing integration of robots into various industries like military,health,education,agriculture, production, surveillance, etc., and everyday life. These robots handle sensitive information, but their limited computational resources and battery life restrict the use of traditional,resource-intensive encryption methods. The existing secure communication protocols for IoRT include lightweight block ciphers and stream ciphers, which still raise concerns about resource consumption and vulnerability to cryptanalysis. Existing methods often lack the adaptability needed for diverse IoRT applications with varying security requirements.This project proposes a novel lightweight encryption mechanism based on discrete-time chaotic maps, specifically the Cubic Map and Ricker's Population Model Map. These maps exhibit a highly sensitive dependence on initial conditions, making them suitable for generating pseudorandom sequences for data encryption. The mechanism incorporates these maps into a confusion-diffusion structure to achieve both masking and scrambling of data.The efficiency of the proposed method is evaluated using Histogram,Entropy ,Correlation,Number of Pixel Change Rate(NPCR) and Unified Average Change Intensity(UACI).

**KEY WORDS**: Discrete-time Chaotic Map, Lightweight Encryption, Internet of Robotic Things (IoRT), Cubic Map, Ricker's Map

# Table of Contents

**Title Page No.**

1. Summary of the base paper 1
2. Merits and Demerits of the base paper 5
3. Source Code 7
4. Output Snapshots 20
5. Conclusion and Future Plans 28
6. References 29
7. Appendix -Base Paper 30

# CHAPTER 1 SUMMARY OF THE BASE PAPER

**Title :** Lightweight encryption with discrete-time chaotic maps for

Internet of Robotic Things.

**Publisher :** ScienceDirect

**Year :** 2023

**Journal name :** Integration

**DOI :** 10.1016/j.vlsi.2023.06.001

**Base paper URL :** “[https://www.sciencedirect.com/science/article/pii/S0167926](https://www.sciencedirect.com/science/article/pii/S0167926023000895)

[023000895](https://www.sciencedirect.com/science/article/pii/S0167926023000895)”

The main contributions of the base paper are:

* Discrete-time chaotic maps are introduced and explored as a foundation for lightweight encryption techniques.
* The proposed encryption technique is designed for real-time, resource-constrained environments found in IoRT applications.
* Performance metrics for the encryption technique, such as measures of memory use, energy efficiency, and encryption/decryption speed, have been presented in the article.

## 1.1 SELECTION OF DISCRETE TIME CHAOTIC SYSTEMS

For data encryption and mixing, two distinct discrete-time chaotic systems in one dimension are selected. The Cubic Map, which encrypts image pixels, is the first of these chaotic systems. After the data is encrypted, the pixels are mixed using Ricker's Population Model, which is the other chaotic system. The robotic processor prefers these one-dimensional models because of their quick processing speeds.

The cubic map equation, represented as equation (1), is a mathematical function that converts input values into output values. The equation for a map based on Ricker's population model, which describes a population's dynamics over time, is shown in Eq. (2). In computer science and mathematics, both equations are used to examine how complex systems will behave.

1. 𝑋

2

= 𝐴𝑋 (1 − 𝑋 )

CUBIC MAP

𝑛+1

𝑛 𝑛

−𝑋

2. 𝑋

𝑛+1

= 𝐴𝑋 𝑒 𝑛

𝑛

RICKER’S POPULATION MODEL

The erratic and non-periodic patterns that these chaotic systems display are shown using time-series graphs. The plots show how the systems are chaotic, with erratic variations and a high sensitivity to initial conditions. The unpredictability of the encrypted data is increased by even slight variations in the initial conditions, which lead to trajectories that are dramatically different over time.

Furthermore, the system's sensitivity to beginning conditions is measured using the Lyapunov exponent. The behavior of these systems is chaotic and is further confirmed to be suitable for encryption by a positive Lyapunov exponent.

Additionally, map function analysis is conducted to explore the relationship between successive values in these chaotic systems. This analysis demonstrates the chaotic nature of the systems, as evidenced by the diverse values observed at each step.

Bifurcation diagrams are employed to further elucidate the chaotic behavior of the systems across parameter variations. For the Cubic Map, chaotic behavior is observed within the parameter range of 2.3 to 3, while for Ricker’s Population Model, chaos occurs between 15 and 22.2.

## PSEUDO RANDOM NUMBER GENERATION

Using the Hem Cubic Map and Ricker's Population Model (A), pseudocode for a chaos-based pseudorandom number generator is provided. For each chaotic system, the number of steps needed to get binary values for encrypting each image is allocated. The 512 × 512 RGB images that are encrypted are used to determine these characteristics. Each pixel in the Cubic Map system requires 24 binary values to be encrypted, for a total of 6,291,456 binary values for the image. The four lowest binary values in floating point are taken from each value obtained, resulting in 1,572,864 values.Following encryption, the pixels are subjected to a coordinate-based exchange mechanism that makes use of Ricker's Population Model results. This procedure enables the simultaneous exchange of 512 distinct pixels when it is carried out first horizontally and subsequently vertically. Nine binary values are needed for each pixel in order to exchange them horizontally and vertically, for a total of 2,359,296 values generated via Ricker's Population Model.

The resulting pseudorandom numbers are tested for randomness using the ENT and NIST-800-22 tests. The NIST-800-22 test assesses randomization using fifteen statistical tests. To be successful, you need to pass all 15 tests. Both sets of binary values used in the study—6,291,456 from the Cubic Map and 7,077,888 from Ricker's Population Model—passed the tests.

## ENCRYPTION

* + 1. **USING CUBIC MAP ALGORITHM:**

for i = startX ; i <= finishX ; i ++ do

for j = startY ; j <= finishY ; j ++ do

image[i,j,channelIndex] = image[i,j,channelIndex] XOR CubicRNG[(imageWidth \* i + j)+(channelIndex \* imageWidth \* imageHeight)];

end

end

## USING RICKER’S MODEL (HORIZONTAL SWAPPING) ALGORITHM:

for i = 0 ; i < imageWidth ; i ++ do temporary=image[imageX,i,channelIndex];

image[imageX,i,channelIndex]=image[imageX,RickersRNG[(imageWidth \* ImageX + i)+(channelIndex \* imageWidth \* imageHeight)],channelIndex];

image[imageX,RickersRNG[(imageWidth \* ImageX + i)+(channelIndex \* imageWidth \* imageHeight)],channelIndex]=temporary;

end

## 1.3.2 USING RICKER’S MODEL (VERTICAL SWAPPING) ALGORITHM:

for i = 0 ; i < imageHeight ; i ++ do temporary=image[i,imageY,channelIndex];

image[i,imageY,channelIndex]=image[RickersRNG[(imageHeight \* ImageX + i)+(channelIndex \* imageWidth \* imageHeight)],imageY,channelIndex];

image[RickersRNG[(imageHeight \* ImageX + i)+(channelIndex \* imageWidth \* imageHeight)],imageY,channelIndex]=temporary;

end

startX and startY denote the top-left coordinates of the image portion to be processed. finishX and finishY represent the bottom-right coordinates of the same portion. imageWidth and imageHeight provide the width and height of the entire image. channelIndex indicates the RGB channel the image belongs to (0 for Red, 1 for Blue, and 2 for Green).

chaoticRNG is an array containing random values, each derived from eight binary values combined to form numbers up to 255

The pictorial representation of the proposed approach is shown in Figures 1.1 and 1.2 as follows:

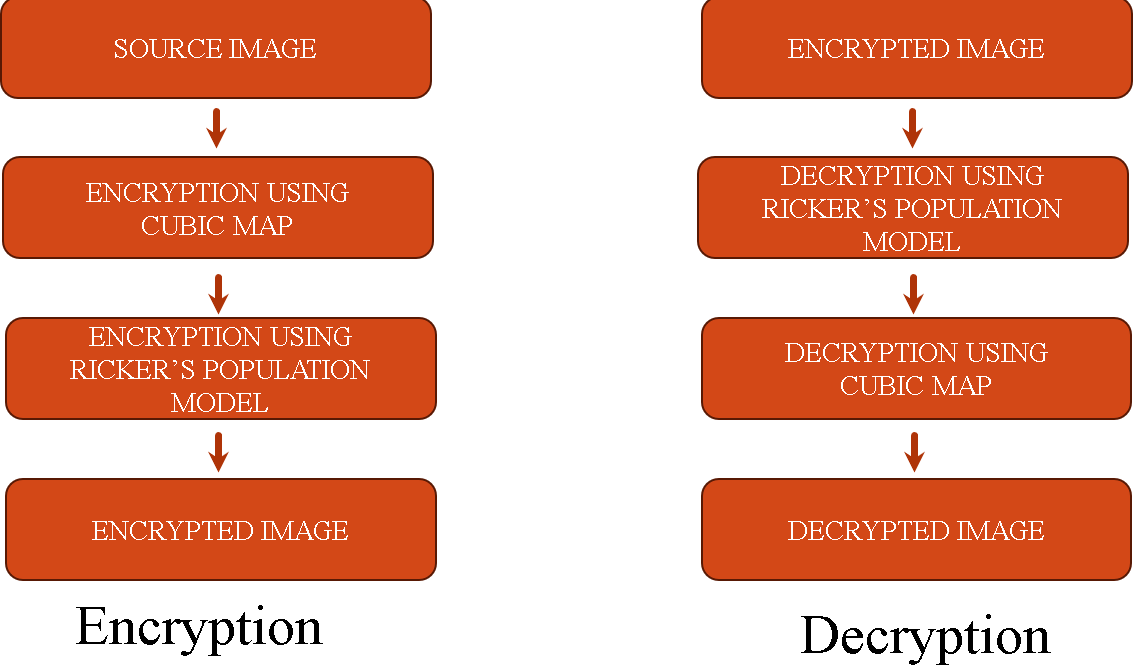


Fig 1.1 Process Flow

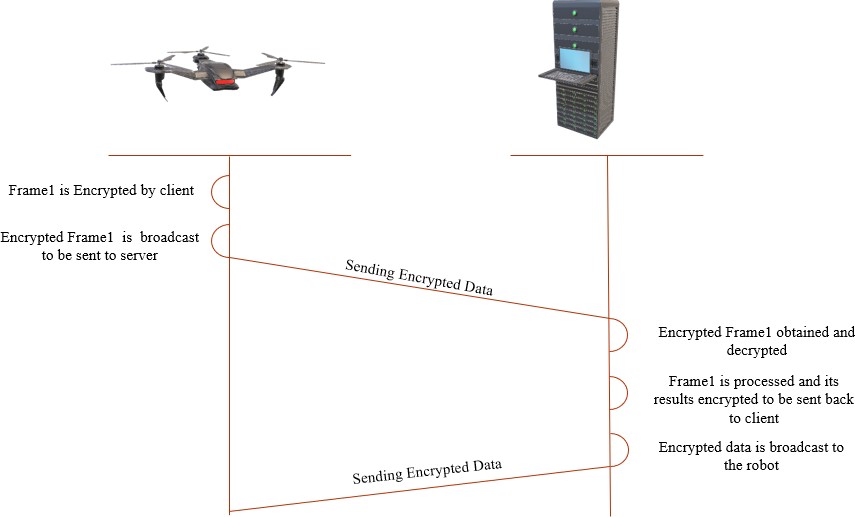


Fig 1.2 Client Server Model

# CHAPTER 2

**MERITS AND DEMERITS OF THE BASE PAPER**

## LITERATURE SURVEY:

There are various encryption techniques available that utilize chaotic maps. Some of them are listed below mentioning the merits and demerits of the proposed method over each of the existing methods.

* **“Chaotic encryption of images in the fractional Fourier transform domain using different modes of operation”** by H.M. Elhoseny, O.S. Faragallah et al. [1]. This paper proposes a novel image encryption method. It scrambles image data in the frequency domain (achieved via Fractional Fourier Transform) using the chaotic Baker map. The scrambling process, controlled by different modes (CBC, CFB, OFB), relies on a secret initialization vector as the key.The disadvantage of this method is that it has higher computational complexity and increased vulnerability to noise.
* **“A new image encryption scheme based on hybrid chaotic maps”** by Pourjabbar Kari et al. [2]. This paper introduces a new grayscale image encryption system that utilizes a combination of chaotic maps. The scrambling and diffusion phases are performed iteratively to encrypt the image. A unique aspect is the selection of a chaotic map based on the correlation coefficient of the original image, enhancing sensitivity to the data. The confusion phase employs Arnold's cat map, while the diffusion phase incorporates extension, XOR, exchange, and transform operations to bolster diffusion speed and sensitivity to the original image.One potential disadvantage is that it does not sufficiently consider the complexity and security needs of specific applications
* **“A new one-dimensional chaotic map and its application in a novel permutation-less image encryption scheme”** by M.Z. Talhaoui et al. [3]. The proposed DCF-IES scheme prioritizes speed for real-time use by adopting a permutation-less architecture. To ensure security without permutation, it utilizes a highly sensitive substitution process that significantly alters pixel values during encryption. This sensitivity is further enhanced by employing a random-like encryption order for rows and columns, generated by a novel one-dimensional chaotic system (1-DCF). But, the disadvantage of this paper is that it suffers from limited randomness due to predictable nature.

## MERITS AND DEMERITS

**Merits:**

* Innovation: Using discrete-time chaotic maps (Cubic Map and Ricker's Population Model Map), the paper presents a novel encryption mechanism for protecting data in the Internet of Robotic Things (IoRT). This is an innovative approach to data security.
* Resource Efficiency: Since chaotic maps are known to generate pseudorandom sequences in an efficient manner, they are a good fit for applications with limited resources, such as the Internet of Robotic Things, where traditional encryption techniques might not be effective.
* Adaptability: The suggested method is made to accommodate various industry demands and scenario variations while meeting the varied security requirements of IoRT applications.
* Evaluation: Using a range of metrics (including histogram, entropy, NPCR, and UACI), the study assesses the effectiveness of the suggested approach and offers a thorough analysis of its results.

## Demerits:

* Real-World Implementation: The paper may not have fully addressed the actual implementation and testing necessary to validate the effectiveness of the suggested method in real-world IoRT environments.
* Limited Key Space: In contrast to conventional cryptography algorithms, chaotic systems could have a smaller key space. The overall security and defense against brute-force assaults may be impacted by this limitation.
* Parameter Selection and Control: To attain desired cryptographic features, chaotic systems need the careful selection and modification of parameters. Inappropriate parameter configurations could endanger the encryption process's efficiency and security.

# CHAPTER 3 SOURCE CODE

## 3.1 SERVER SIDE CODES

* + 1. **IMAGE FORMAT CONVERSION**

import cv2

def convert\_to\_png(img\_path): extension=img\_path[img\_path.rindex('.'):] if(extension!=".png"):

image = cv2.imread(img\_path) path=img\_path[:img\_path.rindex('.')]+".png" cv2.imwrite(path,image)

return path return img\_path

## PSEUDO RANDOM NUMBER GENERATION

import math

def hem\_cubic\_map(A, x0, num\_steps, num\_bits): data = [x0]

for i in range(1, num\_steps + 2):

data.append(A \* data[i - 1] \* (1 - data[i - 1] \* data[i - 1])) data.pop(0) # removing x0

data.pop(0) # removing x1

binary\_data\_array = [] for i in range(num\_steps):

float\_data = '{:.16f}'.format(data[i]).split('.')[1] # Extract decimal part and remove leading zeros

binary\_data = str(bin(int(float\_data)))[-num\_bits:] # Store the lower num\_bits in array

binary\_data\_array.append(binary\_data) return binary\_data\_array

def rickers\_population\_map(A, x0, num\_steps, num\_bits): data = [x0]

for i in range(1, num\_steps + 2):

data.append(A \* data[i - 1] \* (math.e \*\* -data[i - 1]))

data.pop(0) # removing x0 data.pop(0) # removing x1

binary\_data\_array = [] for i in range(num\_steps):

float\_data = '{:.16f}'.format(data[i]).split('.')[1] # Extract decimal part and remove leading zeros

binary\_data = str(bin(int(float\_data)))[-num\_bits:] # Store the lower num\_bits in array

binary\_data\_array.append(binary\_data) return binary\_data\_array

## STORE PSEUDORANDOM NUMBERS

import numpy as np

import PseudoRandomNumberGenerator as PRNG import cv2

import math import random

def store(block\_size):

imgWidth = block\_size[0]#image.shape[1] # must be fixed as 512 imgHeight = block\_size[1]#image.shape[0]

A = 2.3 # 2.3 to 3 for cubic map x0 = 0.1

num\_steps = imgWidth\*imgHeight\*3\*8//8 # 512\*512\*3\*8//8 num\_bits = 8

print("Number of steps in cubic:",num\_steps)

cubicRNGtemp = PRNG.hem\_cubic\_map(A, x0, num\_steps, num\_bits) cubicRNG=[int(cubicRNGtemp[i],2) for i in range(len(cubicRNGtemp))] #print(cubicRNG)

np.save("cubicRNG.npy",cubicRNG)

A = 16 # 15 to 22.2 for rickers map x0 = 0.1

v=math.ceil(math.log(max(imgWidth,imgHeight),2)) num\_steps = imgWidth\*imgHeight\*3\*v//v # 512\*512\*3\*9//9 num\_bits = v

print("Number of steps in rickers:",num\_steps)

rickersRNGtemp = PRNG.rickers\_population\_map(A, x0, num\_steps, num\_bits)

rickersRNG=[int(rickersRNGtemp[i],2) for i in range(len(rickersRNGtemp))] #print(rickersRNG)

np.save("rickersRNG.npy",rickersRNG)

## DECRYPTION

import cv2

import numpy as np import processimage as p

def cubic(source,imgWidth,imgHeight,cubicRNG): for i in range(imgHeight):

for j in range(imgWidth):

for k in range(3): # for each channel (RGB)

source[i][j][k] = source[i][j][k]^cubicRNG[(imgWidth \* i + j + k \* imgWidth \* imgHeight)]

def rickersHorizontalUnique(imgWidth,imgHeight,rickersRNG): d={}

for i in range(imgHeight): for j in range(imgWidth):

for k in range(3): x1=rickersRNG[imgWidth\*i+j+k\*imgWidth\*imgHeight] x1=x1%imgWidth

d[x1]=(i,j,k) D={d[x]:x for x in d} #remove transitivity D2={}

t=list(D.keys()) # keys for i in D:

if (i[0],D[i],i[2]) not in t: D2[i]=D[i]

return D2

def rickersVerticalUnique(imgWidth,imgHeight,rickersRNG): d={}

for i in range(imgWidth): for j in range(imgHeight):

for k in range(3): x1=rickersRNG[imgHeight\*i+j+k\*imgWidth\*imgHeight] x1=x1%imgHeight

d[x1]=(j,i,k) D={d[x]:x for x in d} D2={}

t=list(D.keys()) for i in D:

if (D[i],i[1],i[2]) not in t: D2[i]=D[i]

return D2

def rickersHorizontal(source,D,imgWidth,imgHeight): for i in range(imgHeight):

for j in range(imgWidth):

for k in range(3): if (i,j,k) in D:

x1=D[(i,j,k)] temporary=source[i,j,k] source[i,j,k]=source[i,x1,k] source[i,x1,k]=temporary

def rickersVertical(source,D,imgWidth,imgHeight): for i in range(imgHeight):

for j in range(imgWidth): for k in range(3):

if (i,j,k) in D:

x1=D[(i,j,k)] temporary=source[i,j,k] source[i,j,k]=source[x1,j,k] source[x1,j,k]=temporary

def decryption(block): image=block

# Define image width and height imgWidth = image.shape[1] imgHeight = image.shape[0]

# Load the stored 3D array

cubicRNG = np.load("cubicRNG.npy") rickersRNG = np.load("rickersRNG.npy")

decryptedRickersVertical = np.zeros\_like(image) decryptedRickersHorizontal = np.zeros\_like(image) decryptedCubic = np.zeros\_like(image)

# Perform encryption and decryption

D\_Horizontal=rickersHorizontalUnique(imgWidth,imgHeight,rickersRNG) D\_Vertical=rickersVerticalUnique(imgWidth,imgHeight,rickersRNG)

np.copyto(decryptedRickersVertical,image) rickersVertical(decryptedRickersVertical,D\_Vertical,imgWidth,imgHeight)

np.copyto(decryptedRickersHorizontal,decryptedRickersVertical)

rickersHorizontal(decryptedRickersHorizontal,D\_Horizontal,imgWidth,imgHeigh t)

np.copyto(decryptedCubic,decryptedRickersHorizontal) cubic(decryptedCubic,imgWidth,imgHeight,cubicRNG)

return decryptedCubic

## SPLIT AND PROCESS IMAGE

import cv2

import numpy as np

import LoadCubicRicker as LCR import Decryption as ED

def process\_blockD(block): #rblock=ED.encryption\_decryption(block,x,y) Dblock=ED.decryption(block)

return Dblock

def split\_and\_process\_image(image\_path, block\_size): # Read the image

image = cv2.imread(image\_path) # Get image dimensions

height, width, channels = image.shape

print("Image shape:",image.shape)

#Load Cubic and Ricker LCR.store(block\_size)

# Calculate the number of blocks in both dimensions num\_blocks\_vertical = height // block\_size[1] num\_blocks\_horizontal = width // block\_size[0]

# List to store processed blocks processed\_blocks=[]

# Process each block z=0

for y in range(0, num\_blocks\_vertical): row\_blocks=[]

for x in range(0, num\_blocks\_horizontal):

# Compute the coordinates of the current block start\_x = x \* block\_size[0]

start\_y = y \* block\_size[1] end\_x = start\_x + block\_size[0] end\_y = start\_y + block\_size[1]

# Extract the block from the image

block = image[start\_y:end\_y, start\_x:end\_x] #cv2.imshow('block'+str(z),block)

z+=1

# Process the block

processed\_block = process\_blockD(block) row\_blocks.append(processed\_block)

processed\_blocks.append(row\_blocks) #cv2.waitKey(0)

print("Number of blocks during decryption:",z) return processed\_blocks,(height, width, channels)

## COMBINED MODULE

import cv2

import numpy as np

import SplitAndProcess as SAP import processimage as P

from datetime import \*

def combine\_blocks\_to\_image(processed\_blocks, original\_dims): # Create an empty array for the combined image

combined\_image = np.zeros((original\_dims[0], original\_dims[1], original\_dims[2]), dtype=np.uint8)

block\_height, block\_width = processed\_blocks[0][0].shape[:2] for y, row\_blocks in enumerate(processed\_blocks):

for x, block in enumerate(row\_blocks):

start\_x = x \* block\_width start\_y = y \* block\_height

combined\_image[start\_y:start\_y+block\_height, start\_x:start\_x+block\_width] = block

return combined\_image image\_paths =

["Icon\_Bird\_512x512CombinedEncrypted.png","inputCombinedEncrypted.png"] for image\_path in image\_paths:

image\_path=P.convert\_to\_png(image\_path) block\_size = (128,128)# Define the block size D\_start=datetime.now()

processed\_blocksD,original\_dims = SAP.split\_and\_process\_image(image\_path, block\_size)

combined\_imageD = combine\_blocks\_to\_image(processed\_blocksD, original\_dims)

D\_end=datetime.now() print("Decryption time:",D\_end-D\_start)

image\_pathD=image\_path[:-4]+"CombinedDecrypted"+image\_path[-4:] cv2.imwrite(image\_pathD,combined\_imageD)

cv2.imshow("Combined ImageDecrypted", combined\_imageD) cv2.waitKey(0)

cv2.destroyAllWindows()

12

# Alternatively, save the image using cv2.imwrite('path\_to\_save.jpg', combined\_image)

## SERVER

import socket

# Create a socket object

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# Get the local machine IP address or use a specific IP if available

host = '172.22.11.19' # Replace 'your\_server\_ip' with the actual IP address of the server

port = 12345

# Bind to the port server\_socket.bind((host, port))

# Wait for client connection server\_socket.listen(5)

print("Server listening on {}:{}".format(host, port)) while True:

# Establish connection with client

client\_socket, addr = server\_socket.accept() print('Got connection from', addr)

while True:

# Receive data from client

data = client\_socket.recv(1024) if not data:

break

print("Received from client:", data.decode('utf-8')) # Send a response back to the client

response = input("Enter your response: ") client\_socket.send(response.encode('utf-8'))

# Close the connection with the client client\_socket.close()

## 2.CLIENT SIDE CODES

* + 1. **IMAGE FORMAT CONVERSION**

import cv2

def convert\_to\_png(img\_path): extension=img\_path[img\_path.rindex('.'):] if(extension!=".png"):

image = cv2.imread(img\_path) path=img\_path[:img\_path.rindex('.')]+".png" cv2.imwrite(path,image)

return path return img\_path

## ENCRYPTION

# reversible rickers Horizontal import cv2

import numpy as np import processimage as p

def cubic(source,imgWidth,imgHeight,cubicRNG): for i in range(imgHeight):

for j in range(imgWidth):

for k in range(3): # for each channel (RGB)

source[i][j][k] = source[i][j][k]^cubicRNG[(imgWidth \* i + j + k \* imgWidth \* imgHeight)]

def rickersHorizontalUnique(imgWidth,imgHeight,rickersRNG): d={}

for i in range(imgHeight): for j in range(imgWidth):

for k in range(3): x1=rickersRNG[imgWidth\*i+j+k\*imgWidth\*imgHeight] x1=x1%imgWidth

d[x1]=(i,j,k) D={d[x]:x for x in d} #remove transitivity D2={}

t=list(D.keys()) # keys for i in D:

if (i[0],D[i],i[2]) not in t: D2[i]=D[i]

return D2

def rickersVerticalUnique(imgWidth,imgHeight,rickersRNG): d={}

for i in range(imgWidth): for j in range(imgHeight):

for k in range(3):

x1=rickersRNG[imgHeight\*i+j+k\*imgWidth\*imgHeight] x1=x1%imgHeight

d[x1]=(j,i,k) D={d[x]:x for x in d} D2={}

t=list(D.keys()) for i in D:

if (D[i],i[1],i[2]) not in t: D2[i]=D[i]

return D2

def rickersHorizontal(source,D,imgWidth,imgHeight): for i in range(imgHeight):

for j in range(imgWidth): for k in range(3):

if (i,j,k) in D:

x1=D[(i,j,k)] temporary=source[i,j,k] source[i,j,k]=source[i,x1,k] source[i,x1,k]=temporary

def rickersVertical(source,D,imgWidth,imgHeight): for i in range(imgHeight):

for j in range(imgWidth): for k in range(3):

if (i,j,k) in D:

x1=D[(i,j,k)] temporary=source[i,j,k] source[i,j,k]=source[x1,j,k] source[x1,j,k]=temporary

def encryption(block): image=block

# Define image width and height imgWidth = image.shape[1] imgHeight = image.shape[0]

# Load the stored 3D array

cubicRNG = np.load("cubicRNG.npy") rickersRNG = np.load("rickersRNG.npy")

# Create an empty array to store the encrypted image encryptedCubic = np.zeros\_like(image) encryptedRickersHorizontal = np.zeros\_like(image) encryptedRickersVertical = np.zeros\_like(image)

# Perform encryption and decryption np.copyto(encryptedCubic , image) cubic(encryptedCubic,imgWidth,imgHeight,cubicRNG)

D\_Horizontal=rickersHorizontalUnique(imgWidth,imgHeight,rickersRNG) np.copyto(encryptedRickersHorizontal , encryptedCubic) # b <-- a

rickersHorizontal(encryptedRickersHorizontal,D\_Horizontal,imgWidth,imgHeigh t)

D\_Vertical=rickersVerticalUnique(imgWidth,imgHeight,rickersRNG) np.copyto(encryptedRickersVertical,encryptedRickersHorizontal) rickersVertical(encryptedRickersVertical,D\_Vertical,imgWidth,imgHeight) return encryptedRickersVertical

## SPLIT AND PROCESS IMAGE

import cv2

import numpy as np import Encryption as ED

def process\_blockE(block): #rblock=ED.encryption\_decryption(block,x,y) Eblock=ED.encryption(block)

return Eblock

def split\_and\_process\_image(image\_path, block\_size): # Read the image

image = cv2.imread(image\_path) # Get image dimensions

height, width, channels = image.shape print("Image shape:",image.shape)

#Load Cubic and Ricker

# Calculate the number of blocks in both dimensions num\_blocks\_vertical = height // block\_size[1] num\_blocks\_horizontal = width // block\_size[0]

# List to store processed blocks processed\_blocks=[]

# Process each block z=0

for y in range(0, num\_blocks\_vertical): row\_blocks=[]

for x in range(0, num\_blocks\_horizontal):

# Compute the coordinates of the current block start\_x = x \* block\_size[0]

start\_y = y \* block\_size[1] end\_x = start\_x + block\_size[0] end\_y = start\_y + block\_size[1]

# Extract the block from the image

block = image[start\_y:end\_y, start\_x:end\_x]

#cv2.imshow('block'+str(z),block) z+=1

processed\_block = process\_blockE(block)

row\_blocks.append(processed\_block) processed\_blocks.append(row\_blocks)

#cv2.waitKey(0)

print("Number of blocks during encryption:",z) return processed\_blocks,(height, width, channels)

## COMBINED MODULES

import cv2

import numpy as np

import SplitAndProcess as SAP import processimage as P

from datetime import \*

def combine\_blocks\_to\_image(processed\_blocks, original\_dims): # Create an empty array for the combined image

combined\_image = np.zeros((original\_dims[0], original\_dims[1], original\_dims[2]), dtype=np.uint8)

block\_height, block\_width = processed\_blocks[0][0].shape[:2] for y, row\_blocks in enumerate(processed\_blocks):

for x, block in enumerate(row\_blocks):

start\_x = x \* block\_width start\_y = y \* block\_height

combined\_image[start\_y:start\_y+block\_height, start\_x:start\_x+block\_width] = block

return combined\_image

image\_paths = ['input.png','Ben\_10\_Omnitrix.png'] for image\_path in image\_paths:

image\_path=P.convert\_to\_png(image\_path) block\_size = (128,128)# Define the block size E\_start=datetime.now()

processed\_blocksE,original\_dims = SAP.split\_and\_process\_image(image\_path, block\_size)

combined\_imageE = combine\_blocks\_to\_image(processed\_blocksE, original\_dims)

E\_end=datetime.now()

image\_pathE=image\_path[:-4]+"CombinedEncrypted"+image\_path[-4:] cv2.imwrite(image\_pathE,combined\_imageE)

cv2.imshow("Combined ImageEncrypted", combined\_imageE) cv2.waitKey(0)

print("Encryption time:",E\_end-E\_start)

# Alternatively, save the image using cv2.imwrite('path\_to\_save.jpg', combined\_image)

## CLIENT MODULE

import socket import cv2

import numpy as np

HOST = '127.0.0.1' # The server's hostname or IP address PORT = 65432 # The port used by the server

# Function to send an image and receive confirmation def send\_image(image\_path, sock):

# Load the image

image = cv2.imread(image\_path)

# Check if image loaded successfully if image is None:

print(f"Error: Could not read image '{image\_path}'.") return

# Encode image to bytes (assuming RGB format) image\_data = cv2.imencode('.png', image)[1].tobytes()

# Send image size

image\_size\_str = str(len(image\_data)).encode() sock.sendall(image\_size\_str)

# Send image data in chunks while len(image\_data) > 0:

packet = image\_data[:1024] image\_data = image\_data[1024:] sock.sendall(packet)

# Receive confirmation or error message data = sock.recv(1024).decode()

print(f"Server response for '{image\_path}': {data}")

# Return True if received confirmation, False otherwise return data == 'Image received successfully!'

# Get a list of image paths (modify as needed) image\_paths = ["input.png",

"inputCombinedEncrypted.png","Icon\_Bird\_512x512.png"]

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s: s.connect((HOST, PORT))

# Send image sequence

for image\_path in image\_paths:

# Send image only if previous image received confirmation if send\_image(image\_path, s):

continue # Skip to next iteration if received confirmation else:

print(f"Error sending '{image\_path}'. Stopping transmission.") break

# Close any open windows (optional) cv2.destroyAllWindows()

# CHAPTER 4 OUTPUT SNAPSHOTS

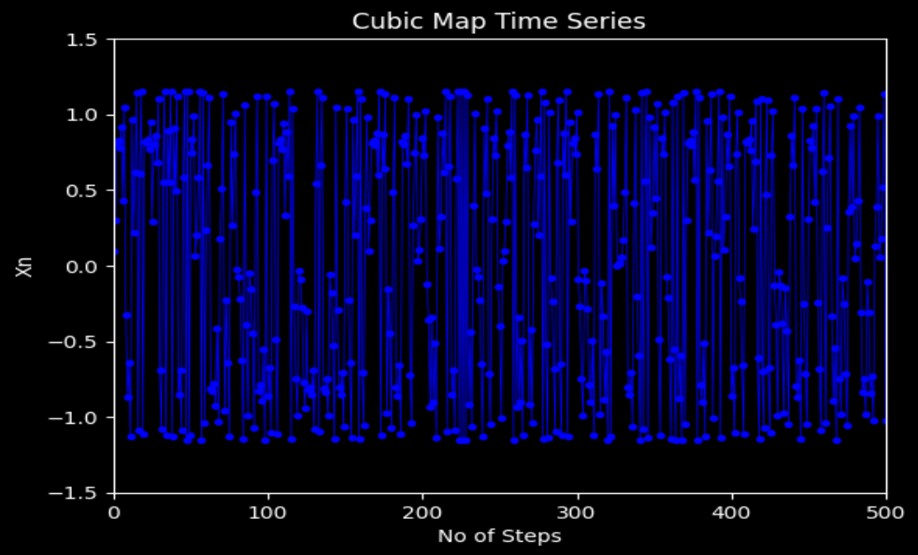


Fig 4.1 Cubic Map’s Time Series

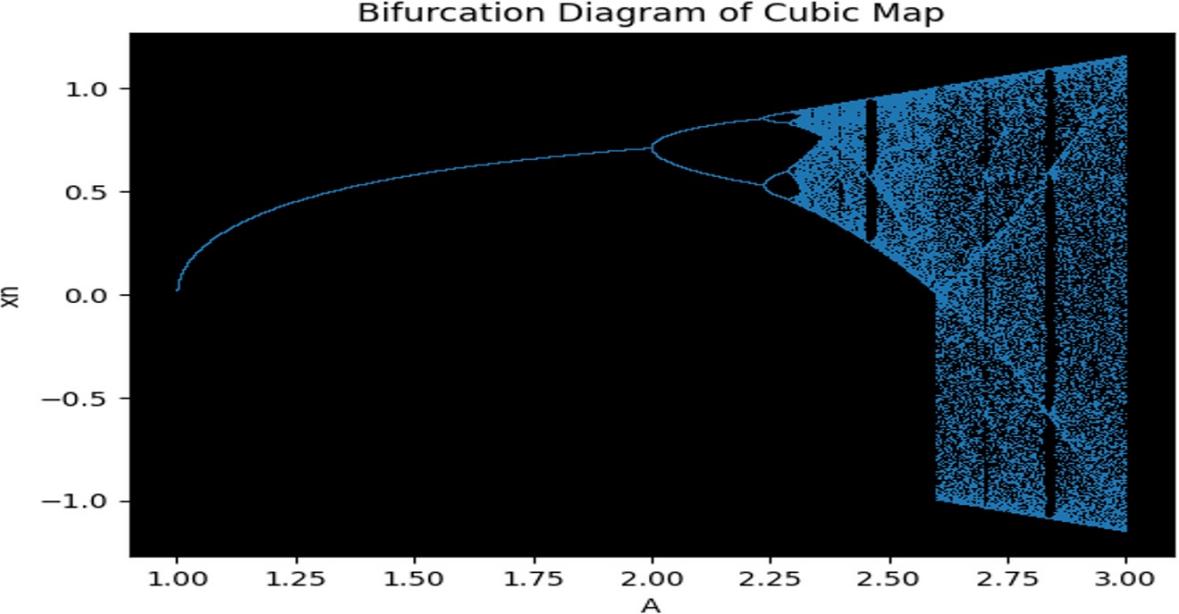


Fig 4.2. Cubic Map’s Bifurcation Diagram

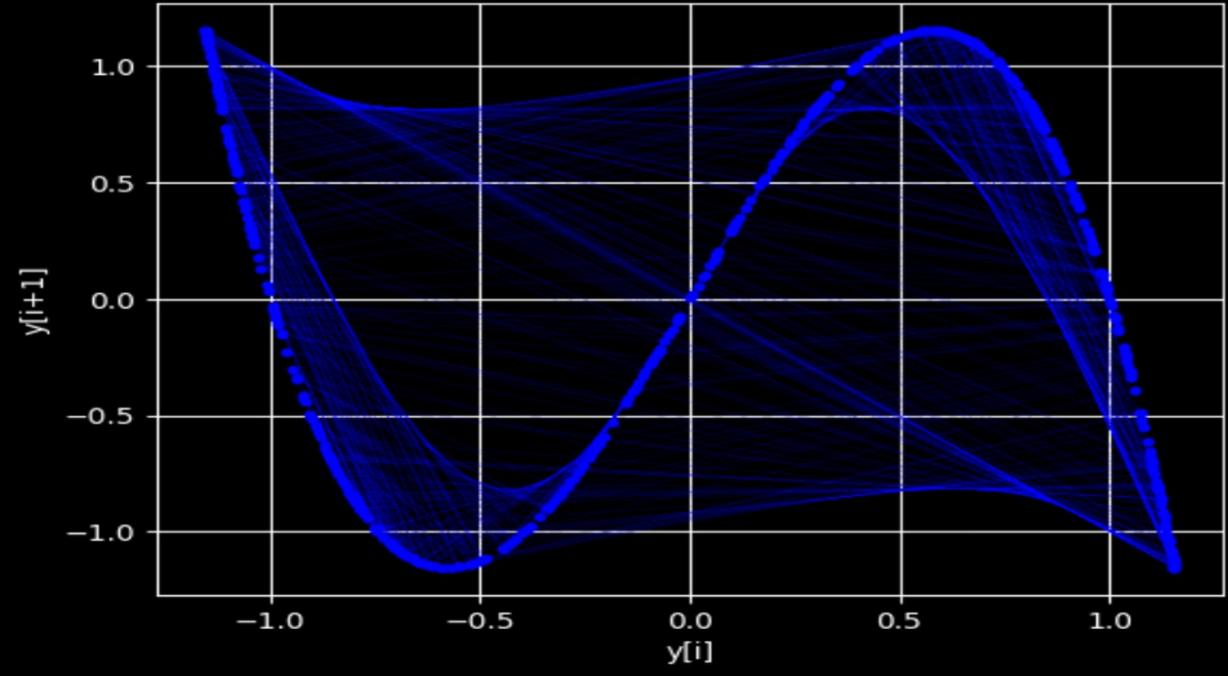


Fig 4.3. Cubic Map’s Map Function

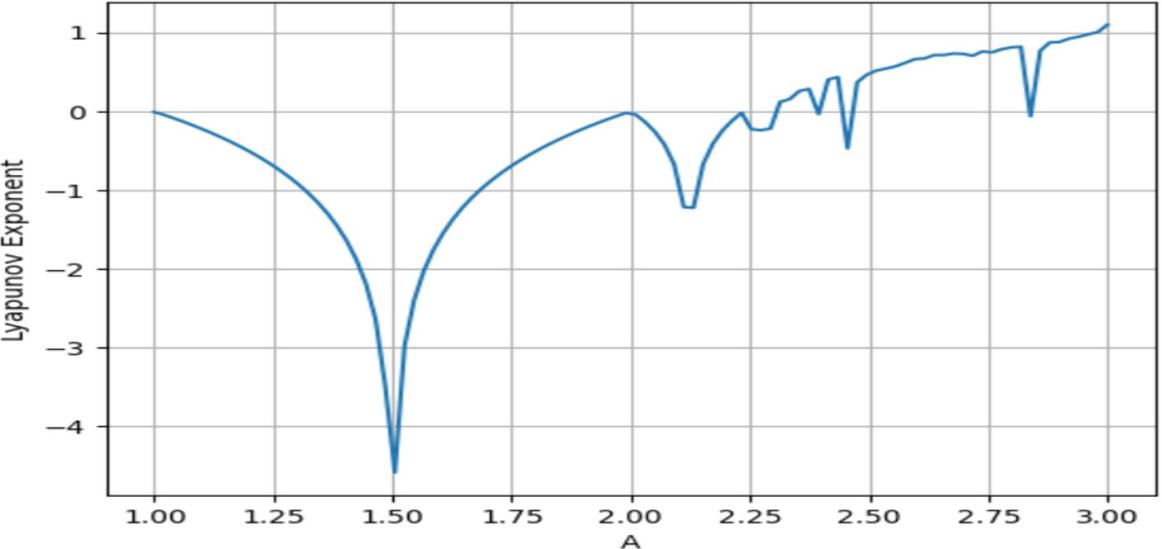


Fig 4.4. Cubic Map’s Lyapunov Exponent

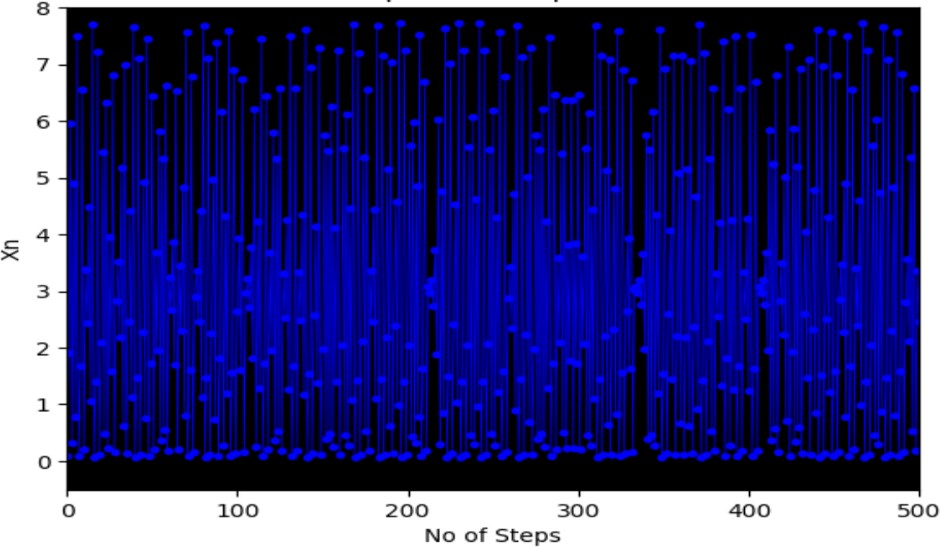


Fig 4.5. Ricker’s Population Model’s Time Series

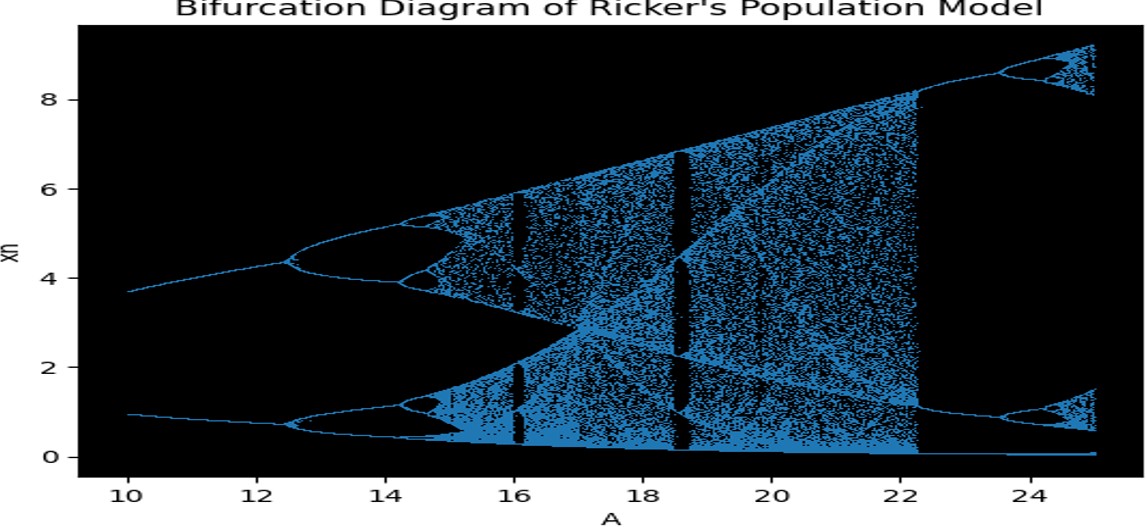


Fig 4.6. Ricker’s Population Model’s Bifurcation Diagram

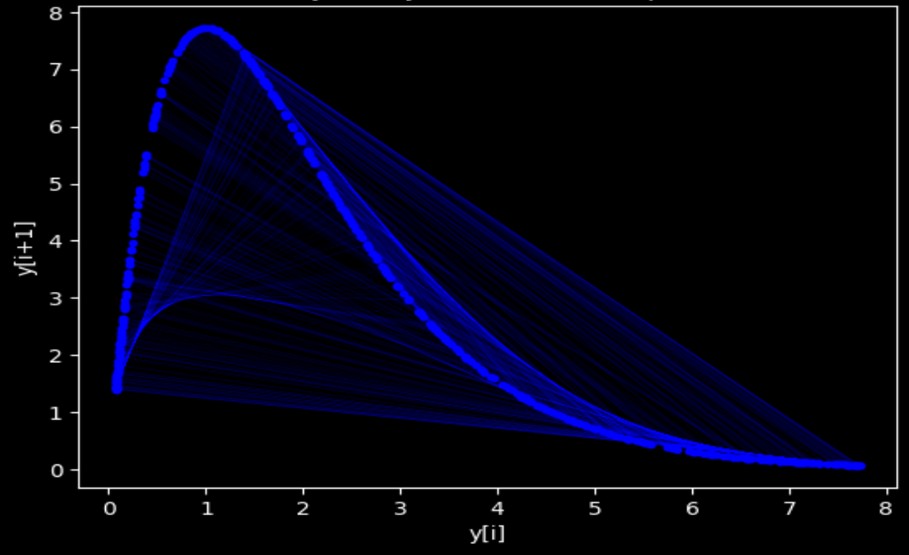


Fig 4.7. Ricker’s Population Model’s Map Function

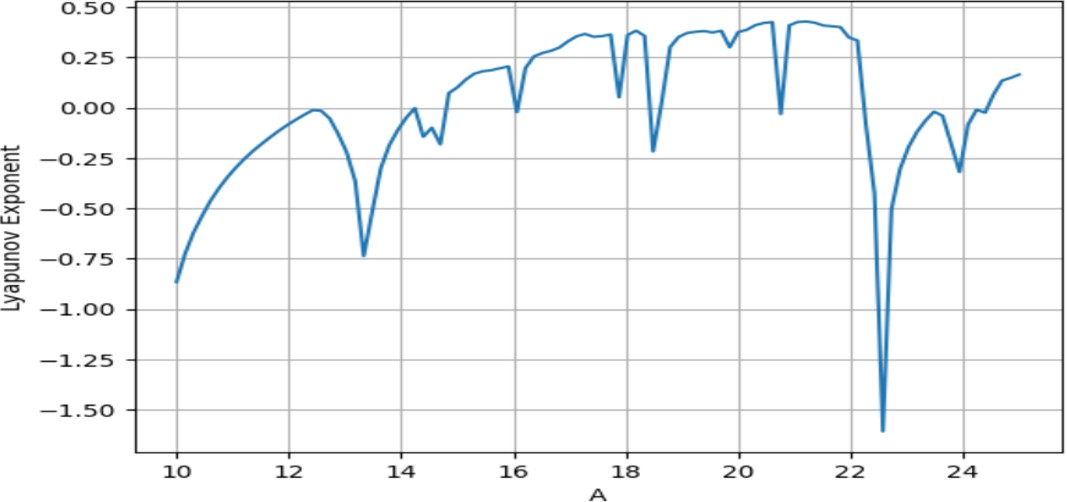


Fig 4.8. Ricker’s Population Model’s Lyapunov Exponent



Fig 4.9 Input Image Fig 4.10. Encrypted Cubic Image

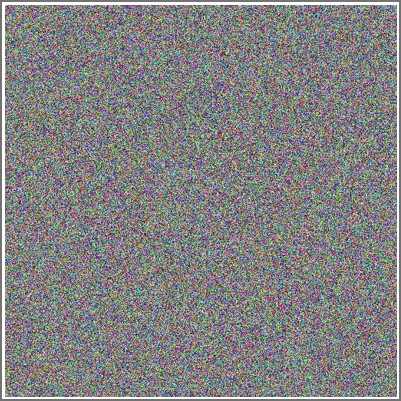


Fig 4.11. Ricker’s Horizontal Fig 4.12. Ricker’s Vertical Encrypted Image Encrypted Image

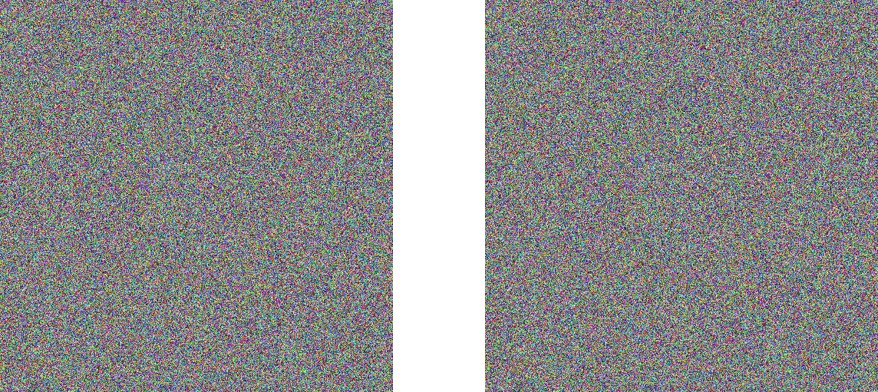


Fig 4.13. Ricker’s Vertical Fig 4.14. Ricker’s Horizontal Decrypted Image Decrypted Image

24

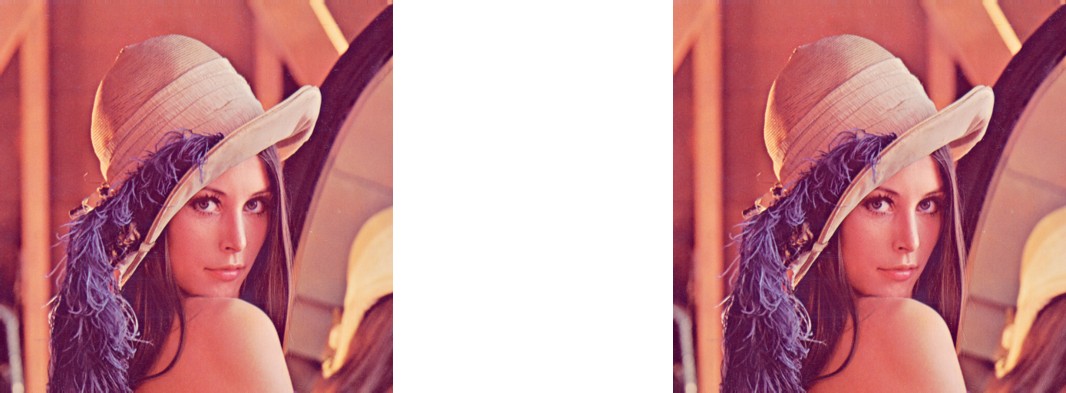


Fig 4.15 Cubic Decrypted Image Fig 4.16 Input Image

# Hardware Implementation



Fig 4.17 Raspberry pi with Webcam



Fig 4.18. Client Side Setup

# HISTOGRAM TESTING

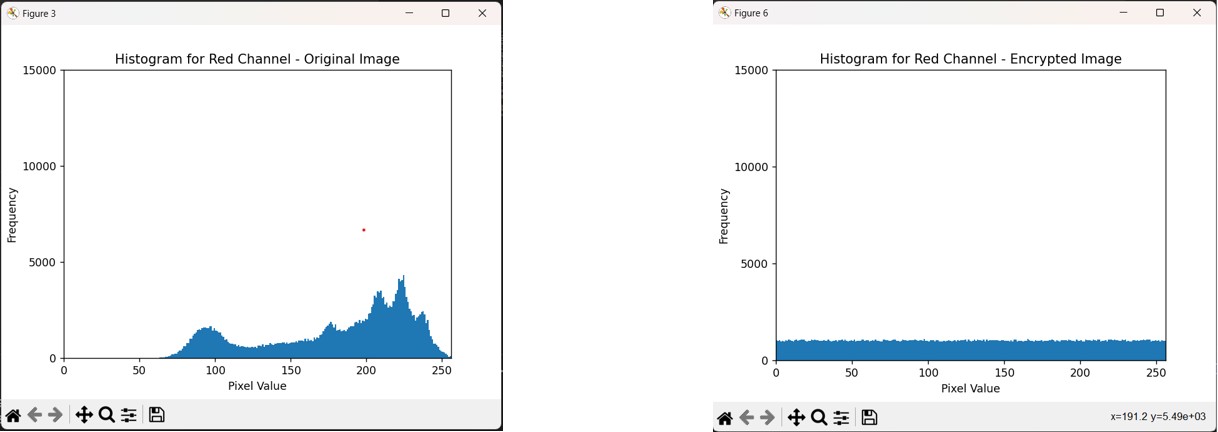
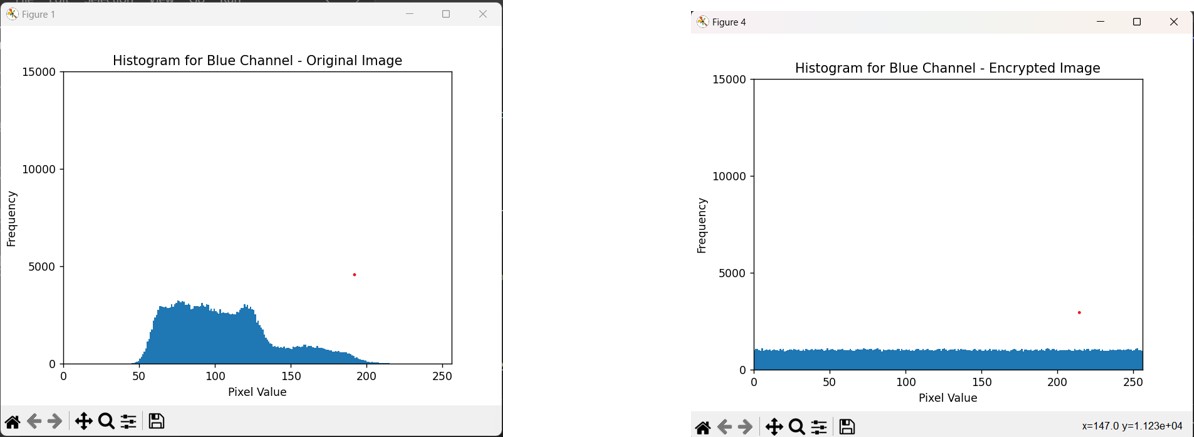


Fig 4.19 Red Channel (Original) Fig 4.20 Red Channel (Encrypted)



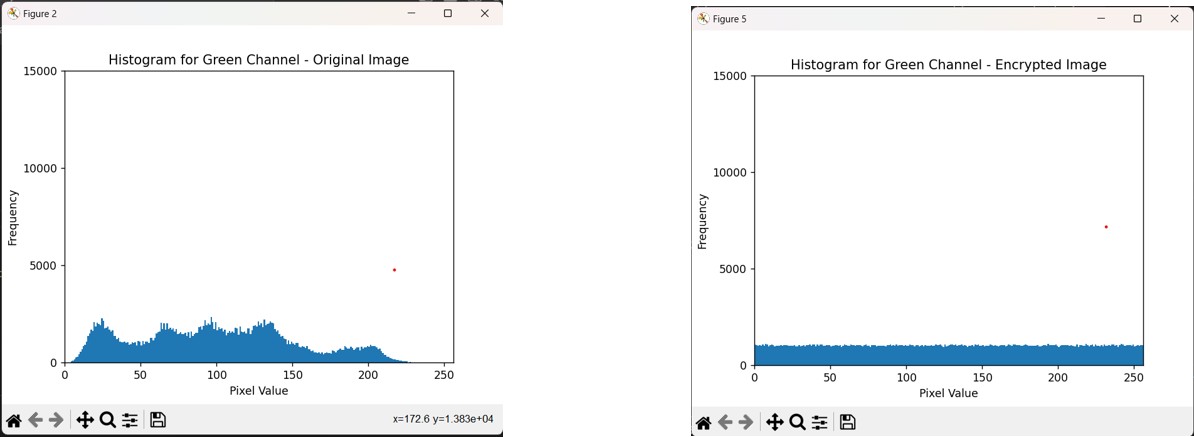
Fig 4.21 Blue Channel (Original) Fig 4.22 Blue Channel (Encrypted)

Fig 4.23. Green Channel (Original) Fig 4.24.Green Channel (Encrypted)

# Testing and Analysis

**Table 4.1 NPCR ,UACI ,ENTROPY TESTING**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SIZE** | **CHANNEL** | **NPCR** | **UACI** | **ENTROPY**  **(Original)** | **ENTROP**  **Y(Encrypt ed)** |
| 512 | Red | 99.6067% | 27.5709% | 6.9684 | 7.99916 |
| 512 | Green | 99.5975% | 30.6384% | 7.5940 | 7.99930 |
| 512 | Blue | 99.5941% | 33.0773% | 7.2531 | 7.99937 |

# Table 4.2 VERTICAL CORRELATION

|  |  |  |  |
| --- | --- | --- | --- |
| **SIZE** | **CHANNEL** | **ORIGINAL** | **ENCRYPTED** |
| 512 | Red | 0.983149 | -0.002356 |
| 512 | Green | 0.982493 | 0.001385 |
| 512 | Blue | 0.957604 | -0.000241 |

**Table 4.3 HORIZONTAL CORRELATION**

|  |  |  |  |
| --- | --- | --- | --- |
| **SIZE** | **CHANNEL** | **ORIGINAL** | **ENCRYPTED** |
| 512 | Red | 0.979773 | 0.0002988 |
| 512 | Green | 0.969065 | 9.3481739 |
| 512 | Blue | 0.932741 | 9.9888888 |

# CHAPTER 5 CONCLUSION AND FUTURE PLANS

For more effective encryption of large-scale data, including pictures and movies of robots, a novel discrete-time chaotic encryption algorithm has been presented in this work.Using Ricker's population model systems and the discrete-time cubic map for encryption, the created method allows for high-rate picture encryption per second and is appropriate for parallel processing, which is the most significant element of this study. The devised mechanism is easily applicable to robot objects based on the acquired image analysis and encryption performance.

Studies on IoRT robot chaotic encryption and robot-specific modules are quite rare. This makes the path for these kinds of investigations very evident. New research on IoRT robots is intended to be presented in subsequent works by parallelizing other chaotic systems.

This project implements encryption and decryption using only a single set of pseudorandom numbers, which are available to both the server and client as secret keys. This can be extended by the server changing the pseudorandom numbers by varying the initial parameter after a random amount of time and passing these values after xoring them with the current version. This increases security but might also increase computational overhead for the client to acquire the new version.

# CHAPTER 6 REFERENCES

**[1] H.M. Elhosany, H.E. Hossin, H.B. Kazemian, O.S. Faragallah,** “Chaotic encryption of images in the fractional Fourier transform domain using different modes of operation”, *in: 2012 29th National Radio Science Conference (NRSC)*, IEEE, 2012, pp. 223–235.

**[2] A. Pourjabbar Kari, A. Habibizad Navin, A.M. Bidgoli, M. Mirnia,** “A new image encryption scheme based on hybrid chaotic maps”, *Multimedia Tools Appl. 80 (2) (2021)* 2753–2772.

**[3] M.Z. Talhaoui, X. Wang, A. Talhaoui,** “A new one-dimensional chaotic map and its application in a novel permutation-less image encryption scheme”, *Visual Computer 37 (7) (2021)* 1757–1768

**[4] Y. Zhou, L. Bao, C.P. Chen,** “A new 1D chaotic system for image encryption”, Signal Process. 97 (2014) 172–182

**[5] W. Cao, Y. Mao, Y. Zhou,** “Designing a 2D infinite collapse map for image

encryption”, Signal Process. 171 (2020) 107457

**[6] D. Prasad, K. Singla, V. Baggan, et al.,** “System model for smart precision farming for high crop yielding”, J. Comput. Theor. Nanosci. 16 (10) (2019) 4406–4411

**[7] M.L. Sahari, I. Boukemara, “**A pseudo-random numbers generator based on a novel 3D chaotic map with an application to color image encryption”, Nonlinear Dynam. 94 (2018) 723–744

# CHAPTER 7 APPENDIX

## Base Paper : Harun Emre Kiran, Akif Akgul, Oktay Yildiz, Emre

**Deniz,**

"Lightweight encryption mechanism with discrete-time chaotic maps for Internet of Robotic Things", ScienceDirect , Integration,Volume 93, ISSN 0167-9260,

102047, 2023.

## DOI : 10.1016/j.vlsi.2023.06.001

**Key Words :** Discrete-time Chaotic Map, Lightweight Encryption,

Internet of Robotic Things (IoRT), Cubic Map, Ricker's Map

**Base Paper URL :** [https://www.sciencedirect.com/science/article/abs/pii/S016](http://www.sciencedirect.com/science/article/abs/pii/S016)

7926023000895

**NIST :** “https://csrc.nist.gov/projects/random-bit-generation/docum entation-and-software”