STRING AND IMAGE ENCRYPTION USING DNA ENCRYPTION, RANDOMLY GENERATED MOORE MACHINE AND HYPERCHAOTIC SYSTEM

Report submitted to the SASTRA Deemed to be University
as the requirement for the course

CSE300/INT300/ICT300 - MINI PROJECT

Submitted By

Saai Sudarsanan D
(123003212, B.Tech CSE)
Aravind M
(123003022, B.Tech CSE)

June 2022

SASTRA
ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION
DEEMED TO BE UNIVERSITY
(U/S 3 of the UGC Act,1956)
THINK MERIT | THINK TRANSPARENCY | THINK SASTRA
THANJAVUR | KUMBAKONAM | CHENNAI

SCHOOL OF COMPUTING

Thanjavur, Tamilnadu, India - 613 401

SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

T H A N J A V U R | K U M B A K O N A M | C H E N N A I

SCHOOL OF COMPUTING

THANJAVUR - 613 401

Bonafide Certificate

This is to certify that the report titled "**String and Image Encryption Using DNA Encryption, Randomly Generated Moore Machine and Hyperchaotic System**" submitted as a requirement for the course, CSE300 / INT300 / ICT300: MINI PROJECT for B.Tech. is a bonafide record of the work done by Mr Aravind M(Reg. No. 123003022, B.Tech CSE) and Mr Saai Sudarsanan D (Reg. No.123003212, B.Tech CSE) during the academic year 2021-22, in the School of Computing, under my supervision.

Signature of Project Supervisor:

Name with Affiliation: Dr Kannan Balasubramanian, Professor, SOC, SASTRA Deemed University, Thanjavur

Date: 24 - 06 - 2022

Miniproject *Viva voce* Held on : ____________

Examiner 1 **Examiner 2**

ACKNOWLEDGEMENTS

We would like to thank our Honorable Chancellor **Prof. R. Sethuraman** for providing us with an opportunity and the necessary infrastructure for carrying out this project as a part of our curriculum.

We would like to thank our Honorable Vice-Chancellor **Dr S. Vaidhyasubramaniam** and **Dr S. Swaminathan**, Dean, Planning & Development, for the encouragement and strategic support at every step of our college life.

We extend our sincere thanks to **Dr R. Chandramouli**, Registrar, SASTRA Deemed to be University for providing the opportunity to pursue this project.

We extend our heartfelt thanks to **Dr A. Umamakeswari**, Dean, School of Computing, **Dr B.Santhi**, Associate Dean, Department of Computer Application, **Dr R. Muthaiah**, Associate Dean, Department of Information Technology and Information & Communication Technology and **Dr V. S. Shankar Sriram**, Associate Dean, Department of Computer Science and Engineering for their motivation and support offered in materialising this project.

Our guide **Dr Kannan Balasubramanian**, Professor, School of Computing was the driving force behind this whole idea from the start. His deep insight into the field and invaluable suggestions helped us in making progress throughout our project work. We also thank the project review panel members for their valuable comments and insights which made this project better.

We would like to extend our gratitude to all the teaching and non-teaching faculties of the School of Computing who have either directly or indirectly helped us in the completion of the project.

We gratefully acknowledge all the contributions and encouragement from my family and friends resulting in the successful completion of this project. We thank you all for providing me with an opportunity to showcase my skills through the project.

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

DNA	Deoxyribonucleic Acid
OTP	One Time Password
CPS	Cyber Physical Systems
IoT	Internet of Things
DCR	Dynamic Coding Rule
MSE	Mean Squared Error

ABSTRACT

A large amount of data is being transmitted over the internet and in cyber-physical systems, such data also includes highly confidential data like bank account details, credit card details, OTPs and other sensitive data. We must use cryptography to encrypt such data when using untrusted channels of communications, to ensure secure transmission of data. The paper proposes a novel encryption algorithm based on DNA cryptography, the plain text or the image is encrypted using DNA cryptography and is converted into a DNA sequence like 'ATCGGAT'. Many cryptographic operations are performed on this sequence. The hyperchaotic system generates pseudorandom numbers that are normalized and converted into binary strings and then into DNA sequences which are then used to perform further operations on the DNA from the plaintext. The randomness in the hyperchaotic system helps improve the confidentiality of the encrypted message. The Moore Machine is used to perform substitutions in the given DNA sequence, making it more secure. Different cryptographic attacks were performed to analyze the strength of the algorithm and the proposed scheme is more secure and efficient than existing schemes.

Keywords: *Image Encryption, Moore Machine, Hyperchaotic System, DNA Encoding*

TABLE OF CONTENTS

CHAPTER 1
SUMMARY OF THE BASE PAPER

The base paper is titled A novel cryptosystem based on DNA cryptography, hyperchaotic systems and a randomly generated Moore machine for cyber-physical systems. The Journal is Computer Communications and the paper was published on 15th April 2022. The Journal has been indexed in SCIE, Scopus and SJR.

Securing data in Cyberphysical Systems is important due to the sensitivity of the data concerned. In a Cyberphysical system, a lot of confidential data is stored and transmitted, especially when it comes to the health care sector. Only authorized users must be allowed to use and access the data. To ensure this, there is a need for fast and highly efficient cryptosystems to perform secure encryption of the data.

The objective of the paper is to propose a cryptosystem that is fast, highly efficient and also secure at the same time. The security of the cryptosystem is measured by performing various kinds of attacks. The authors have analysed it with other similar algorithms already present and also under different conditions. The avalanche effect of the algorithms is measured. The Moore machine and the input to the hyperchaotic system are randomly generated and transferred according to the requirements.

The paper has novelly implemented hyperchaotic systems, randomly generated Moore machines and DNA Cryptography.

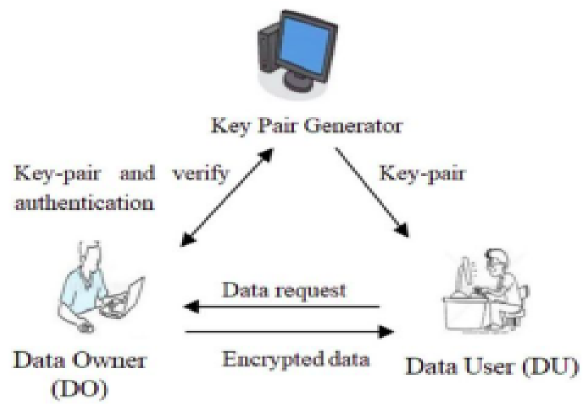
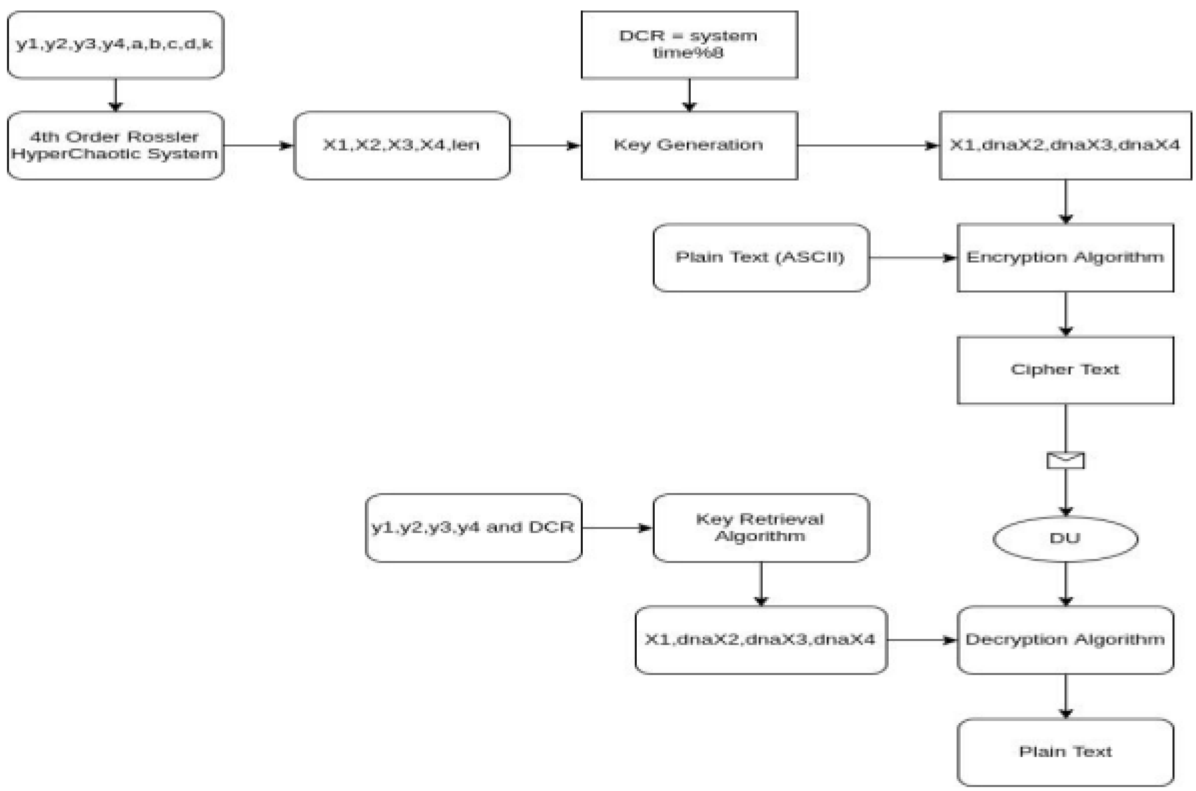
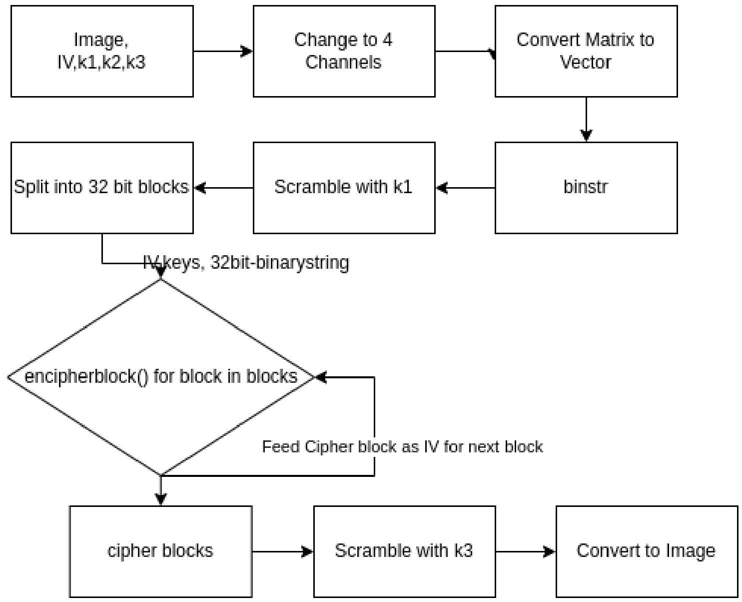


Fig. 2. Proposed scheme system model.



Architecture for String Encryption



Architecture for Image Encryption

Algorithms in String and File Encryption

Algorithm 1: Key Generation Algorithm
Input: X1, X2, X3, X4 and len
Output: X1, dnaX2, dnaX3 and dnaX4
1 **START**
2 **CONVERT** X1, X2, X3, X4 into values from {0,1,2,...,len-1}
3 **CONVERT** X2, X3, X4 into binary values binX2, binX3 and binX4.
4 **GENERATE** DCR from the system time
5 **PERFORM** dynamic DNA coding on binX2 to generate dnaX2
6 **PERFORM** dynamic DNA coding on binX3 to generate dnaX3
7 **PERFORM** dynamic DNA coding on binX4 to generate dnaX4
8 **STOP**

Algorithm 2: Encryption Algorithm
Input: ASCII values, X1, dnaX2, dnaX3 and dnaX4
Output: Ciphertext (dna4)
1 **START**
2 **SCRAMBLE** ASCII values of plaintext using index values of X1

REPLACE ASCII values with their binary values to generate string binPT
CONVERT X1 into its equivalent binary string to generate binX1
EXECUTE EXOR between binPT and binX1
CONVERT the binary string into DNA sequence dna
EXECUTE DNA ADD between dna and dnaX2 to generate dna1
EXECUTE DNA SUB between dna1 and dnaX3 to dna2
EXECUTE DNA EXOR between dna2 and dnaX4 to generate dna3
INPUT dna3 to Moore Machine
SAVE output of Moore Machine in dna4
IF DCR < 3
    COMPLEMENT DNA bases in dna4 having an even index
IF DCR >= 3
    COMPLEMENT DNA bases in dna4 having an odd index
STOP

Algorithm 3: Key Retrieval Algorithm
Input: y1, y2, y3, y4 and DCR
Output: X1, dnaX2, dnaX3 and dnaX4
START
CONVERT X1, X2, X3, X4 into values from {0,1,2,...,len-1}
CONVERT X2, X3, X4 into binary values binX2, binX3 and binX4.
GENERATE DCR from the system time
PERFORM dynamic DNA coding on binX2 to generate dnaX2
PERFORM dynamic DNA coding on binX3 to generate dnaX3
PERFORM dynamic DNA coding on binX4 to generate dnaX4
STOP

Algorithm 4: Decryption Algorithm
Input: Ciphertext, DCR, X1, dnaX2, dnaX3 and dnaX4
Output: Plaintext
START
DECODE DCR to generate Moore Machine
IF DCR < 3
    COMPLEMENT the DNA bases of ciphertext with even index
IF DCR >= 3

6 COMPLEMENT the DNA bases of ciphertext with odd index
7 **SAVE** complemented ciphertext as dna4
8 **INPUT** dna4 into the Moore Machine
9 **SAVE** output of the Moore Machine as dna3
10 **EXECUTE** DNA EXOR between dna3 and dnaX4 to generate dna2
11 **EXECUTE** DNA ADD between dna2 and dnaX3 to generate dna1
12 **EXECUTE** DNA SUB between dna1 and dnaX2 to generate dna
13 **CONVERT** dna and XI into binary strings dnabin and binX1
14 **EXECUTE** EXOR between dnabin and binX1 to generate binPT
15 **CONVERT** binPT into its equivalent ASCII values
16 **UNSCRAMBLE** ASCII values using the index values of X1
17 **CONVERT** ASCII values into plaintext
18 **STOP**

Algorithms in Image Encryption

Algorithm 1: Key Generation Algorithm
Input: None
Output: [SR,y1,y2,y3],[IV, k1, k2, k3]
1 **GENERATE** 4 Random 32-bit numbers S4, y1, y2, y3
2 **EXECUTE** SCRAMBLE(SR,(y1 XOR y2 XOR y3)) to generate IV
3 **EXECUTE** SCRAMBLE(y1,(SR XOR y2 XOR y3)) to generate k1
4 **EXECUTE** SCRAMBLE(y2,(y1 XOR SR XOR y3)) to generate k2
5 **EXECUTE** SCRAMBLE(y3,(y1 XOR y2 XOR SR)) to generate k3
6 **RETURN**

Algorithm 2: Encryption Algorithm
Input: IV, k1, k2, k3, 4-channels Image
Output: Encrypted Image Array
1 **SAVE** Image shape into tshape
2 **CONVERT** Image Matrix to Vector
3 **CONVERT** image vector into binary string binimg
4 **SCRAMBLE** binimg with k1
5 **SAVE** length of binimg into l

6 **SPLIT** binimg into 32-bit blocks
7 **FOR** block in blocks
8   cblock = ENCIPHERBLOCK(IV, [k1,k2,k3], block)
9   IV = cblock
10   **APPEND** cblock to cblocks
11 **SCRAMBLE** cblocks with k3
12 **MAKEIMG** using cblocks and set shape = tshape
13 **STOP**

Algorithm 3: Key Retrieval Algorithm
INPUT: SR, y1, y2, y3
OUTPUT: IV, k1, k2, k3
1 **EXECUTE** SCRAMBLE(SR,(y1 XOR y2 XOR y3)) to generate IV
2 **EXECUTE** SCRAMBLE(y1,(SR XOR y2 XOR y3)) to generate k1
3 **EXECUTE** SCRAMBLE(y2,(y1 XOR SR XOR y3)) to generate k2
4 **EXECUTE** SCRAMBLE(y3,(y1 XOR y2 XOR SR)) to generate k3
5 **RETURN**

Algorithm 4: Decryption Algorithm
INPUT: IV, k1, k2, k3 and Encrypted Image
OUTPUT: Decrypted Image
1 **SAVE** image shape in tshape
2 **CONVERT** Image Matrix to Vector
3 **CONVERT** Image Vector to binary string binimg
4 **SAVE** length on binimg in l
5 **UNSCRAMBLE** binimg with k3
6 **SPLIT** binimg into 32-bit blocks
7 **FOR** cblock in cblocks
8   block = ENCIPHERBLOCK(IV, [k1,k2,k3], block)
9   IV = cblock
10   **APPEND** block to blocks
11 **UNSCRAMBLE** blocks with k1
12 **MAKEIMG** with blocks and set shape = tshape
13 **STOP**

Algorithm 5: Encipher Block Algorithm

INPUT: IV, k1, k2, k3, block
OUTPUT: cblock
1 **EXOR** IV and block to get cblock
2 **SCRAMBLE** cblock with k3
3 **SBOX** cblock
4 **EXOR** cblock with k2
5 **CONVERT** cblock to DNA Sequence dna with DCR = IV
6 **CONVERT** k1 to DNA Sequence dnak1 with DCR = IV
7 **CONVERT** k2 to DNA Sequence dnak2 with DCR = IV
8 **CONVERT** k3 to DNA Sequence dnak3 with DCR = IV
9 **EXECUTE** DNA ADD dnak1 with dna to get dna1
10 **EXECUTE** DNA SUB dnak2 with dna1 to get dna2
11 **EXECUTE** DNA EXOR dnak3 with dna2 to get dna3
12 **INPUT** dna3 to MOORE Machine-generated using k2 to get dna4
13 **IF** k3 < 3
14 COMPLEMENT DNA bases in dna4 having an even index
15 **IF** k3 >= 3
16 COMPLEMENT DNA bases in dna4 having an odd index
17 **DECODE** dna4 to get binary string cblock
18 **RETURN**

Algorithm 6: Decipher Block Algorithm
Input: IV, k1, k2, k3 and cblock
Output: block
1 **CONVERT** cblock to dna
2 **CONVERT** k1 to DNA Sequence dnak1
3 **CONVERT** k2 to DNA Sequence dnak2
4 **CONVERT** k3 to DNA Sequence dnak3
5 **CONVERT** cblock to DNA Sequence dna4
5 **IF** k3 < 3
6 COMPLEMENT DNA bases in dna4 having an even index
7 **IF** k3 >= 3
8 COMPLEMENT DNA bases in dna4 having an odd index
9 **INPUT** dna4 to MOORE MACHINE generated with k2 and get dna3
10 **EXECUTE** DNA EXOR dna3 with dnak3 to get dna2

11 **EXECUTE** DNA ADD dna2 with dnak2 to get dna1
12 **EXECUTE** DNA SUB dna1 with dnak1 to get dna
13 **DECODE** dna with DCR to get cblock
14 **ISBOX** cblock
15 **EXOR** cblock with k2
16 **UNSCRAMBLE** cblock with k3
16 **EXOR** cblock with IV to get block
17 **RETURN**

CHAPTER 2
MERITS AND DEMERITS

Existing Methods

Thangavel and Varalaxmi have proposed a scheme that helps improve data security by encrypting using 16 x 16 matrix operations and EXOR operations in their paper *Enhanced DNA and ElGamal cryptosystem for secure data storage and retrieval in cloud* published in the year 2018. The technique proposed by them is Sate of the Art and provided better data security. It was also a novel method, but the proposed scheme and the algorithms involves were computationally intensive.

A.K Kaundal and A Verma have proposed a scheme which involves using an OTP and has a Feistel-inspired structure in their paper *Extending Feistel structure to DNA cryptography*. The technique proposed provided good data security and was inspired by the Feistel Structure and also implemented the OTP but the main disadvantage was that the DNA Sequence used for encryption was too short and hence it faces serious security issues.

A. Majumdar, A. Biswas, A. Majumder, S.K. Sood, K.L. Baishnab in their paper *A novel DNA-inspired encryption strategy for concealing cloud storage* have proposed a scheme that involves the extensive use of 2D Matrix operations and complex S-Box Operations. The merits of this scheme were the extensive use of the S-Box Algorithm, but the algorithm uses a 2D matrix and hence the computations involved were of time complexity O(n^2) and also there were many complex operations in this scheme making it computationally intensive.

Comparing with Existing Schemes

The proposed scheme takes lesser encryption and decryption time when compared to the above-mentioned state-of-the-art methods, the scheme uses only simple operations like addition, multiplication, subtraction and division, unlike the other schemes that use complex 2D Matrix operations or other computationally intensive algorithms. All work products of the proposed algorithm are one-dimensional vectors which make the algorithm less computationally intensive.

The time complexity of the algorithm's operations on the one-dimensional DNA Sequences is also equal to O(n). The Moore machine's operations are also performed using an O(n) algorithm with the help of a hash-table and the overall time complexity of the algorithm is O(n).

The throughput of the algorithm is decided by the encryption time and plaintext size and it was calculated by using the below algorithm.

Throughput = (size of plaintext)/(encryption time)

The proposed algorithm showed greater throughput than the other algorithms.

CHAPTER 3
SOURCE CODE

DNA STRING AND FILE ENCRYPTION

```
# Import Required Modules
import time
import random

# Tables Involved
CODES={
    0:{"00":"T","01":"A","10":"G","11":"C"},
    1:{"00":"A","01":"T","10":"C","11":"G"},
    2:{"00":"A","01":"C","10":"T","11":"G"},
    3:{"00":"C","01":"A","10":"G","11":"T"},
    4:{"00":"T","01":"G","10":"A","11":"C"},
    5:{"00":"G","01":"C","10":"T","11":"A"},
    6:{"00":"G","01":"T","10":"C","11":"A"},
    7:{"00":"C","01":"G","10":"A","11":"T"}
}
DECODES = {
    0:{'T':'00','A':'01','G':'10','C':'11'},
    1:{'A':'00','T':'01','C':'10','G':'11'},
    2:{'A':'00','C':'01','T':'10','G':'11'},
    3:{'C':'00','A':'01','G':'10','T':'11'},
    4:{'T':'00','G':'01','A':'10','C':'11'},
    5:{'G':'00','C':'01','T':'10','A':'11'},
    6:{'G':'00','T':'01','C':'10','A':'11'},
    7:{'C':'00','G':'01','A':'10','T':'11'}
}
dna_xor = {
    "G":{"G":"A","A":"G","C":"T","T":"C"},
    "A":{"G":"G","A":"A","C":"C","T":"T"},
    "C":{"G":"T","A":"C","C":"A","T":"G"},
    "T":{"G":"C","A":"T","C":"G","T":"A"}
}
dna_add = {
    "G":{"G":"C","A":"G","C":"T","T":"A"},
    "A":{"G":"G","A":"A","C":"C","T":"T"},
    "C":{"G":"T","A":"C","C":"A","T":"G"},
```

```
    "T":{"G":"A","A":"T","C":"G","T":"C"}
}
dna_sub = {
    "G":{"G":"A","A":"G","C":"T","T":"C"},
    "A":{"G":"T","A":"A","C":"C","T":"G"},
    "C":{"G":"G","A":"C","C":"A","T":"T"},
    "T":{"G":"C","A":"T","C":"G","T":"A"}
}
COMPLEMENT = {"A":"G","G":"A","T":"C","C":"T"}

# Get Binary String from ASCII values
def binstr(x): # Enter ASCII LIST
    binx = ""
    for i in x:
        binx += bin(i)[2:].zfill(8)
    return binx # Returns a string of binary numbers

# Get ASCII from binary string
def ascii(binstr): # Enter a string of binary numbers
    return [int(binstr[i:i+8],2) for i in range(0,len(binstr),8)] # Returns a list of ASCII values

# Get DNA Code
def code(DCR,binstr): # Enter a string of 0's and 1's
    val = DCR%8
     return "".join([CODES[val][binstr[i:i+2]] for i in range(0,len(binstr),2)]) # Returns DNA
Sequence

# Decode DNA Sequence
def decode(DCR,dnastr): # Enter DNA Sequence
    val = DCR%8
    return "".join([DECODES[val][i] for i in dnastr]) # returns a string of binary numbers

# Moore Machine
def moore(DCR,dna): # Enter a DNA Sequence
    rnd = DCR%4
    stateInOut = {rnd : "A",(rnd+1)%4 : "T",(rnd+2)%4 : "C",(rnd+3)%4 : "G"}
    ttable = {
        "G":{0:(rnd+1)%4,1:rnd,2:(rnd+2)%4,3:(rnd+3)%4},
        "A":{0:(rnd+2)%4,1:(rnd+1)%4,2:(rnd+3)%4,3:rnd},
        "C":{0:rnd,1:(rnd+3)%4,2:(rnd+1)%4,3:(rnd+2)%4},
```

```
        "T":{0:(rnd+3)%4,1:(rnd+2)%4,2:rnd,3:(rnd+1)%4}
    }
    state = rnd
    for i in range(len(dna)):
        state = ttable[dna[i]][state]
        dna[i] = stateInOut[state]
    return dna # Returns a different DNA Sequence

# Reverse Moore Machine
def revMoore(DCR,dna): # Enter a DNA Sequence
    rnd = DCR%4
    stateOutIn = {"A":rnd,"T":(rnd+1)%4,"C":(rnd+2)%4,"G":(rnd+3)%4}
    rttable = {
        0:{(rnd+1)%4:"G",(rnd+2)%4:"A",rnd:"C",(rnd+3)%4:"T"},
        1:{rnd:"G",(rnd+1)%4:"A",(rnd+3)%4:"C",(rnd+2)%4:"T"},
        2:{(rnd+2)%4:"G",(rnd+3)%4:"A",(rnd+1)%4:"C",rnd:"T"},
        3:{(rnd+3)%4:"G",rnd:"A",(rnd+2)%4:"C",(rnd+1)%4:"T"}
    }
    state = rnd
    ndna = ""
    for i in range(len(dna)):
        fstate = stateOutIn[dna[i]]
        ndna += rttable[state][fstate]
        state = fstate
    return ndna # Returns a different DNA Sequence

# Key Generation Algorithm & Hyperchaotic System
def keygen(pt): # Enter the ASCII values in plain text
    DCR = round(time.time())
    y1 = random.getrandbits(10)
    y2 = random.getrandbits(10)
    y3 = random.getrandbits(10)
    y4 = random.getrandbits(10)
    a = 36
    b = 36
    c = 28
    d = -16
    k = 0.5
    X1 = []
    X2 = []
```

```
    X3 = []
    X4 = []
    SUM = sum(pt)
    LEN = len(pt)
    y1 = abs(y1 - (float(SUM%100)/1000))
    y2 = abs(y2 - (float(SUM%100)/1000))
    y3 = abs(y3 - (float(SUM%100)/1000))
    y4 = abs(y4 - (float(SUM%100)/1000))
    ini = (y1,y2,y3,y4)

    for _ in range(LEN):
        y1 = a*(y2-y1)
        y2 = -1*(y1*y3) + (d*y1) + (c*y2) - y4
        y3 = (y1*y2)-(b*y3)
        y4 = y1+k
        y1 = abs(y1 - round(y1))
        y2 = abs(y2 - round(y2))
        y3 = abs(y3 - round(y3))
        y4 = abs(y4 - round(y4))
        X1.append(y1)
        X2.append(y2)
        X3.append(y3)
        X4.append(y4)
    keys = []
    for key in [X1,X2,X3,X4]:
        tmp = sorted(key)
        nkey = []
        for e in key:
            nkey.append(tmp.index(e))
        keys.append(nkey)
    for e in range(1,len(keys)):
        keys[e] = code(DCR,binstr(keys[e]))
        return DCR,keys,ini # Return DCR, keys -> [X1,dnaX2,dnaX3,dnaX4] and ini -> [y1,y2,y3,y4]

# Key Retreival Algorithm
def keyretr(ct,DCR,ini): # Enter Cipher Text, DCR,and ini - > [y1,y2,y3,y4] (Obtained from the Key Generation Process)
    y1,y2,y3,y4 = ini[0],ini[1],ini[2],ini[3]
    a = 36
```

```
    b = 36
    c = 28
    d = -16
    k = 0.5
    X1 = []
    X2 = []
    X3 = []
    X4 = []
    for _ in range(len(ct)//4):
        y1 = a*(y2-y1)
        y2 = -1*(y1*y3) + (d*y1) + (c*y2) - y4
        y3 = (y1*y2)-(b*y3)
        y4 = y1+k
        y1 = abs(y1 - round(y1))
        y2 = abs(y2 - round(y2))
        y3 = abs(y3 - round(y3))
        y4 = abs(y4 - round(y4))
        X1.append(y1)
        X2.append(y2)
        X3.append(y3)
        X4.append(y4)
    keys = []
    for key in [X1,X2,X3,X4]:
        tmp = sorted(key)
        nkey = []
        for e in key:
            nkey.append(tmp.index(e))
        keys.append(nkey)
    for e in range(1,len(keys)):
        keys[e] = code(DCR,binstr(keys[e]))
    return keys # Return keys -> [X1,dnaX2,dnaX3,dnaX4]

def encrypt(pt,DCR,keys): # Enter ASCII Values, DCR, keys -> [X1,dnaX2,dnaX3,dnaX4]
    # Scramble Plain Text with index values of X1
    scrpt = []
    for i in keys[0]:
        scrpt.append(pt[i])
    # Convert X1 and Scrambled Plaintext into binary strings
    binx1 = binstr(keys[0])
    binpt = binstr(scrpt)
```

```
    binpt = list(binpt)
    # XOR binx1 and binpt
    for i in range(len(binpt)):
        binpt[i] = "0" if binpt[i] == binx1[i] else "1"
    binpt = "".join(binpt)
    # DNA Coding on binpt
    dna = code(DCR,binpt)
    # ADD dna , dnaX2
    dna1 = [dna_add[i][j] for i,j in zip(dna,keys[1])]
    # SUBTRACT dna1 , dnaX3
    dna2 = [dna_sub[i][j] for i,j in zip(dna1,keys[2])]
    # XOR dna2 , dnaX4
    dna3 = [dna_xor[i][j] for i,j in zip(dna2,keys[3])]
    # Pass dna3 as input to randomly generated moore machine
    dna4 = moore(DCR,dna3)

    # If DCR < 3 complement even indexes, else complement odd indexes
    if DCR%8 < 3:
        for i in range(len(dna4)):
            if i%2 == 0:
                dna4[i] = COMPLEMENT[dna4[i]]
    else:
        for i in range(len(dna4)):
            if i%2 != 0:
                dna4[i] = COMPLEMENT[dna4[i]]
    return "".join(dna4)

# Decryption Algorithm
def decrypt(ct,DCR,keys): # Enter Cipher Text, DCR, keys -> [X1,dnaX2,dnaX3,dnaX4]
    dna4 = list(ct)

    # Reverse Complement
    if DCR%8 < 3:
        for i in range(len(dna4)):
            if i%2 == 0:
                dna4[i] = COMPLEMENT[dna4[i]]
    else:
        for i in range(len(dna4)):
            if i%2 != 0:
                dna4[i] = COMPLEMENT[dna4[i]]
```

```
    # Reverse Moore Machine
    dna3 = revMoore(DCR,dna4)

    # Reverse DNA Operations
    dna2 = [dna_xor[i][j] for i,j in zip(dna3,keys[3])]
    dna1 = [dna_add[i][j] for i,j in zip(dna2,keys[2])]
    dna = [dna_sub[i][j] for i,j in zip(dna1,keys[1])]
    binx1 = binstr(keys[0])
    dnabin = list(decode(DCR,dna))
    # Reverse XOR between binx1 and binpt
    for i in range(len(dnabin)):
        dnabin[i] = "0" if dnabin[i] == binx1[i] else "1"
    binpt = "".join(dnabin)
    # Scrambled List of ASCII Values
    scrpt = ascii(binpt)
    # Unscramble using X1 indexes
    pt = []
    for i in range(len(scrpt)):
        pt.append(scrpt[keys[0].index(i)])
    return pt # Return ASCII in list of Plain Text Characters

# String Encryption Algorithm
def strenc(pt:str):
    inp = [ord(i) for i in pt]
    DCR,keys,ini = keygen(inp)
    ct = encrypt(inp,DCR,keys)
    return ct, DCR, ini

# String Decryption Algorithm
def strdec(ct:str,DCR,ini):
    keys = keyretr(ct,DCR,ini)
    pt = decrypt(ct,DCR,keys)
    pt = "".join([chr(i) for i in pt])
    return pt

# File Encryption Algorithm
def fileenc(fname:str):
    inp = open(fname,"rb").read()
    DCR,keys,ini = keygen(inp)
```

```
    out = open(f"{fname}.enc","w")
    ct = encrypt(inp,DCR,keys)
    out.write(ct)
    out.close()
    return DCR, ini

# File Decryption Algorithm
def filedec(cf:str,DCR,ini):
    ct = open(cf,"r").readlines()[0]
    fname = cf.split(".")[0] + ".dec"
    keys = keyretr(ct,DCR,ini)
    pt = decrypt(ct,DCR,keys)
    pt = bytearray(pt)
    pt = bytes(pt)
    f = open(fname,"wb")
    f.write(pt)
    f.close()
```

DNA IMAGE ENCRYPTION

```
# Import Required Modules
from PIL import Image
import random
import numpy as np

# Making the CODES List
cdna = []
cbin = []
count = 0
for i in "ATGC":
    for j in "ATGC":
        for k in "ATGC":
            for l in "ATGC":
                cdna.append(i+j+k+l)
for i in range(256):
    cbin.append(bin(i)[2:].zfill(8))
DECODES = {}
for i in range(256):
    DECODES[i] = {}
    cdnal = cdna[i:] + cdna[:i]
    for j in range(256):
```

```
        DECODES[i][cdnal[j]] = cbin[j]

# Making the DECODES List
CODES = {}
for i in range(256):
    CODES[i] = {v:k for k,v in DECODES[i].items()}

Sbox = [
0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16 ]

InvSbox = [
0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb,
0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb,
0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e,
0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25,
0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92,
0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84,
0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06,
0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b,
0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73,
0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e,
0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b,
0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4,
0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f,
0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef,
```

```
0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61,
0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d ]

dna_xor = {
   "G":{"G":"A","A":"G","C":"T","T":"C"},
   "A":{"G":"G","A":"A","C":"C","T":"T"},
   "C":{"G":"T","A":"C","C":"A","T":"G"},
   "T":{"G":"C","A":"T","C":"G","T":"A"}
}
dna_add = {
   "G":{"G":"C","A":"G","C":"T","T":"A"},
   "A":{"G":"G","A":"A","C":"C","T":"T"},
   "C":{"G":"T","A":"C","C":"A","T":"G"},
   "T":{"G":"A","A":"T","C":"G","T":"C"}
}
dna_sub = {
   "G":{"G":"A","A":"G","C":"T","T":"C"},
   "A":{"G":"T","A":"A","C":"C","T":"G"},
   "C":{"G":"G","A":"C","C":"A","T":"T"},
   "T":{"G":"C","A":"T","C":"G","T":"A"}
}

COMPLEMENT = {"A":"G","G":"A","T":"C","C":"T"}

# Get Binary String from ASCII values
def binstr(x): # Enter ASCII LIST
   binx = ""
   for i in x:
      binx += bin(i)[2:].zfill(8)
   return binx # Returns a string of binary numbers

# Get ASCII from binary string
def ascii(binstr): # Enter a string of binary numbers
   return [int(binstr[i:i+8],2) for i in range(0,len(binstr),8)] # Returns a list of ASCII values

# Get DNA Code
def code(DCR,binstr): # Enter a string of 0's and 1's
   val = DCR%256
   return "".join([CODES[val][binstr[i:i+8]] for i in range(0,len(binstr),8)]) # Returns DNA
Sequence
```

```
# Decode DNA Sequence
def decode(DCR,dnastr): # Enter DNA Sequence
    val = DCR%256
     return "".join([DECODES[val][dnastr[i:i+4]] for i in range(0,len(dnastr),4)]) # returns a
string of binary numbers

# Moore Machine
def moore(DCR,dna): # Enter a DNA Sequence
    rnd = DCR%4
    stateInOut = {rnd : "A",(rnd+1)%4 : "T",(rnd+2)%4 : "C",(rnd+3)%4 : "G"}
    ttable = {
        "G":{0:(rnd+1)%4,1:rnd,2:(rnd+2)%4,3:(rnd+3)%4},
        "A":{0:(rnd+2)%4,1:(rnd+1)%4,2:(rnd+3)%4,3:rnd},
        "C":{0:rnd,1:(rnd+3)%4,2:(rnd+1)%4,3:(rnd+2)%4},
        "T":{0:(rnd+3)%4,1:(rnd+2)%4,2:rnd,3:(rnd+1)%4}
    }
    state = rnd
    for i in range(len(dna)):
        state = ttable[dna[i]][state]
        dna[i] = stateInOut[state]
    return dna # Returns a different DNA Sequence

# Reverse Moore Machine
def revMoore(DCR,dna): # Enter a DNA Sequence
    rnd = DCR%4
    stateOutIn = {"A":rnd,"T":(rnd+1)%4,"C":(rnd+2)%4,"G":(rnd+3)%4}
    rttable = {
        0:{(rnd+1)%4:"G",(rnd+2)%4:"A",rnd:"C",(rnd+3)%4:"T"},
        1:{rnd:"G",(rnd+1)%4:"A",(rnd+3)%4:"C",(rnd+2)%4:"T"},
        2:{(rnd+2)%4:"G",(rnd+3)%4:"A",(rnd+1)%4:"C",rnd:"T"},
        3:{(rnd+3)%4:"G",rnd:"A",(rnd+2)%4:"C",(rnd+1)%4:"T"}
    }
    state = rnd
    ndna = ""
    for i in range(len(dna)):
        fstate = stateOutIn[dna[i]]
        ndna += rttable[state][fstate]
        state = fstate
```

```
    return ndna # Returns a different DNA Sequence
def unscramble(k,b): # Key -> binary string, b -> list of integers
    l = len(b)
    ptr1 = 0
    ptr0 = l//32*(k.count('1'))
    n = []
    for i in range(l):
        if k[i%32] == '1':
            n.append(b[ptr1])
            ptr1+=1
        else:
            n.append(b[ptr0])
            ptr0+=1
    return n # -> list of integers

def scramble(k,b):  # Key -> binary string, b -> list of integers
    zero = []
    one = []
    for i in range(len(b)):
        if k[i%32] == '0':
            zero.append(b[i])
        else:
            one.append(b[i])
    return one + zero # -> list of values

def keygen():
    SR = random.getrandbits(32)
    y1 = random.getrandbits(32)
    y2 = random.getrandbits(32)
    y3 = random.getrandbits(32)
        t1  =  "".join(scramble(bin(SR)[2:].zfill(32),bin(y1^y2^y3)[2:].zfill(32)))  #  IV  and  for
Coding Rule
    t2 = "".join(scramble(bin(y1)[2:].zfill(32),bin(y2^y3^SR)[2:].zfill(32))) # Scramble
    t3 = "".join(scramble(bin(y2)[2:].zfill(32),bin(y1^y3^SR)[2:].zfill(32))) # Key
    t4 = "".join(scramble(bin(y3)[2:].zfill(32),bin(y1^y2^SR)[2:].zfill(32))) # Key
    return [SR,y1,y2,y3],[t1,t2,t3,t4]

def keyretr(ini):
    SR = ini[0]
    y1 = ini[1]
```

```
    y2 = ini[2]
    y3 = ini[3]
    t1 = "".join(scramble(bin(SR)[2:].zfill(32),bin(y1^y2^y3)[2:].zfill(32))) # IV and for Coding Rule
    t2 = "".join(scramble(bin(y1)[2:].zfill(32),bin(y2^y3^SR)[2:].zfill(32))) # Scramble
    t3 = "".join(scramble(bin(y2)[2:].zfill(32),bin(y1^y3^SR)[2:].zfill(32))) # Key
    t4 = "".join(scramble(bin(y3)[2:].zfill(32),bin(y1^y2^SR)[2:].zfill(32))) # Key
    return [t1,t2,t3,t4]

def sbox(key,pt):
    words = [pt[i:i+8] for i in range(0,32,8)]
    words = "".join([bin(int(Sbox[int(i,2)]))[2:].zfill(8) for i in words])
    cblock = "".join(['0' if x == y else '1' for x,y in zip(key,words)])
    return cblock

def isbox(key,ct):
    words = "".join(['0' if x == y else '1' for x,y in zip(key,ct)])
    words = [words[i:i+8] for i in range(0,32,8)]
    cblock = "".join([bin(int(InvSbox[int(i,2)]))[2:].zfill(8) for i in words])
    return cblock

def decipherBlock(IV,keys,cblock):
    DCR = int(IV,2)
    k1 = int(keys[0],2)
    k2 = int(keys[1],2)
    k3 = int(keys[2],2)
    # Code with k1
    dna4 = list(code(k1,cblock))

    # Reverse Complement with k3
    if k3%8 < 3:
        for i in range(len(dna4)):
            if i%2 == 0:
                dna4[i] = COMPLEMENT[dna4[i]]
    else:
        for i in range(len(dna4)):
            if i%2 != 0:
                dna4[i] = COMPLEMENT[dna4[i]]

    # Reverse Moore
```

```
    dna3 = revMoore(k2,dna4)
    # Reverse DNA Operations
    dnak1 = code(DCR,keys[0])
    dnak2 = code(DCR,keys[1])
    dnak3 = code(DCR,keys[2])
    dna2 = [dna_xor[i][j] for i,j in zip(dna3,dnak3)]
    dna1 = [dna_add[i][j] for i,j in zip(dna2,dnak2)]
    bdna = [dna_sub[i][j] for i,j in zip(dna1,dnak1)]
    bdna = "".join(bdna)
    cblock = decode(DCR,bdna)

    # Using k2 pass it through inverse SBox
    cblock = isbox(keys[1],cblock)
    # Using k3 unscramble the cblock
    cblock = "".join(unscramble(keys[2],cblock))
    # IV xor cblock = block
    block = "".join(['0' if x == y else '1' for x,y in zip(IV,cblock)])
    return block

def encipherBlock(IV,keys,block): # 32 bits
    # IV xor Block
    cblock = "".join(['0' if x == y else '1' for x,y in zip(IV,block)])
    # Using k3 scramble each cblock
    cblock = "".join(scramble(keys[2],cblock))
    # Using k2 pass it through  SBox
    cblock = sbox(keys[1],cblock)

    # DNA Operations Begin
    DCR = int(IV,2)
    k1 = int(keys[0],2)
    k2 = int(keys[1],2)
    k3 = int(keys[2],2)
    cblock = code(DCR,cblock)
    dnak1 = code(DCR,keys[0])
    dnak2 = code(DCR,keys[1])
    dnak3 = code(DCR,keys[2])
    # ADD dna , dnaX2
    dna1 = [dna_add[i][j] for i,j in zip(cblock,dnak1)]
    # SUBTRACT dna1 , dnaX3
    dna2 = [dna_sub[i][j] for i,j in zip(dna1,dnak2)]
```

```
    # XOR dna2 , dnaX4
    dna3 = [dna_xor[i][j] for i,j in zip(dna2,dnak3)]
    # Moore Machine using k2
    dna4 = moore(k2,dna3)

    # If k3%8 < 3 complement even indexes, else complement odd indexes
    if k3%8 < 3:
        for i in range(len(dna4)):
            if i%2 == 0:
                dna4[i] = COMPLEMENT[dna4[i]]
    else:
        for i in range(len(dna4)):
            if i%2 != 0:
                dna4[i] = COMPLEMENT[dna4[i]]

    # Decode the cblock with k1 and return it.
    cblock = decode(k1,"".join(dna4))
    return cblock

def makeimg(img,tshape,out="imgsave.png"): # List of Integers
    pixels = [int(img[i:i+8],2) for i in range(0,len(img),8)]
    img = np.array(pixels)
    img = np.reshape(img,tshape)
    image = Image.fromarray(img.astype(np.uint8))
    image.save(out,bitmap_format='png')
    return img

def encrypt(keys,image):
    # Get the Keys
    IV = keys[0]
    k1 = keys[1]
    k2 = keys[2]
    k3 = keys[3]
    img = np.array(Image.open(image))
    if img.shape[-1] != 4:
        img = np.dstack((img, np.ones(img.shape[:-1])*255)).astype(np.uint8)
    tshape = img.shape
    img = img.flatten()

    # Make it a binary string and scramble it.
```

```
    img = binstr(img)
    img = "".join(scramble(k1,img))

    # Get the length of the binary string
    l = len(img)

    # Block Operation
    blocks = [img[i:i+32] for i in range(0,l,32)]
    print("No of blocks (Encryption): ", len(blocks))
    cblocks = []
    IV = keys[0]
    for block in blocks:
        cblock = encipherBlock(IV,[k1,k2,k3],block)
        IV = cblock
        cblocks.append(cblock)

    # Converge all blocks and get final binary to convert into image
    cimg = "".join(cblocks)
    cimg = "".join(scramble(k3,cimg))
    makeimg(cimg,tshape,out=f"{''.join(image.split('.')[:-1])}.enc.png")

def decrypt(keys,image,out="retr.png"): # add paramns ,shape,csp
    # Get the keys
    IV = keys[0]
    k1 = keys[1]
    k2 = keys[2]
    k3 = keys[3]

    # Open Encrypted Image
    img = np.array(Image.open(image))
    tshape = img.shape
    img = img.flatten()
    img = binstr(img)
    l = len(img)

    # Unscramble using k1
    img = "".join(unscramble(k3,img))
    # Block Operations
    cblocks = [img[i:i+32] for i in range(0,l,32)]
    print("No of blocks (Decryption): ", len(cblocks))
```

```
    blocks = []
    for cblock in cblocks:
        blocks.append(decipherBlock(IV,[k1,k2,k3],cblock))
        IV = cblock

    # Convert it into binary string
    pimg = "".join(blocks)

    # Unscramble using k1
    pimg = "".join(unscramble(k1,pimg))

    # Retrieve final image
    return makeimg(pimg,tshape,out)

def imgenc(imfile):
    ini,keys = keygen()
    encrypt(keys,imfile)
    return ini

def imgdec(ini,imfile,out="retr.png"):
    keys = keyretr(ini)
    return decrypt(keys,imfile,out)
```

ANALYSIS

```
import time
from dnaimage import imgdec
from PIL import Image
import numpy as np
from sewar.full_ref import mse, rmse, psnr, rmse_sw, uqi, ssim, ergas, scc, rase, sam,
msssim, vifp, psnrb
import warnings
warnings.filterwarnings("ignore")

imfile = "lena.enc.png"
original = np.array(Image.open("lena.png"),dtype=np.uint8)
if original.shape[-1] != 4:
        original = np.dstack((original, np.ones(original.shape[:-1])*255)).astype(np.uint8)
def changeb(x:int,pos):
    c={"1":"0","0":"1"}
    b=list(bin(x)[2:].zfill(32))
```

```
    b[pos] = c[b[pos]]
    b = int("".join(b),2)
    return b

keystring = "769296855 3392518851 935530902 3545687538"
k1,k2,k3,k4 = list(map(int,keystring.split(" ")))
testcases = {
    "proper-decryption": [k1,k2,k3,k4],
    "1-missing": [0,k2,k3,k4],
    "2-missing": [k1,0,k3,k4],
    "3-missing": [k1,k2,0,k4],
    "4-missing": [k1,k2,k3,0],
    "1-2-missing": [0,0,k3,k4],
    "1-3-missing": [0,k2,0,k4],
    "1-4-missing": [0,k2,k3,0],
    "2-3-missing": [k1,0,0,k4],
    "2-4-missing": [k1,0,k3,0],
    "3-4-missing": [k1,k2,0,0],
    "1-msb":[changeb(k1,0),k2,k3,k4],
    "2-msb":[k1,changeb(k2,0),k3,k4],
    "3-msb":[k1,k2,changeb(k3,0),k4],
    "4-msb":[k1,k2,k3,changeb(k4,0)],
    "1-lsb":[changeb(k1,-1),k2,k3,k4],
    "2-lsb":[k1,changeb(k2,-1),k3,k4],
    "3-lsb":[k1,k2,changeb(k3,-1),k4],
    "4-lsb":[k1,k2,k3,changeb(k4,-1)]
}

import pandas as pd
test = pd.DataFrame(columns=["test","mse", "rmse", "psnr", "rmse_sw", "uqi", "ssim",
"ergas", "scc", "rase", "sam", "msssim", "vifp", "psnrb","dectime"])

funcnames = ["mse", "rmse", "psnr", "rmse_sw", "uqi", "ssim", "ergas", "scc", "rase", "sam",
"msssim", "vifp", "psnrb"]
funcs = [mse, rmse, psnr, rmse_sw, uqi, ssim, ergas, scc, rase, sam, msssim, vifp, psnrb]

for k,v in testcases.items():
    tic = time.time()
    print(f"Testing --{k}--")
    decrypted = imgdec(v,imfile,k+".png")
```

```
    toc = time.time()
    casedict = {funcname:func(original,decrypted) for funcname,func in zip(funcnames,funcs)}
    casedict["test"] = k
    casedict["dectime"] = toc-tic
    test = test.append(casedict,ignore_index=True)
    print(f"Time Taken {toc - tic}")
    print("")
test.to_csv("analysis"+"_"+imfile+".csv")
```

CHAPTER 4
SNAPSHOTS

STRING ENCRYPTION

```
(ai) saais@pop-os:~/.../str-file$ python string_demo.py
Enter your String : SASTRA
DCR: 1656056389
ini: 892.938 69.938 110.938 747.938
Cipher Text: CCGTTAGAGACGCCAGCGCTTGTA

--------------------------------------------------
---Decryption---
Enter Cipher Text: CCGTTAGAGACGCCAGCGCTTGTA
DCR: 1656056389
Enter y1 y2 y3 y4:892.938 69.938 110.938 747.938
--------------------------------------------------
SASTRA
--------------------------------------------------
Enter your String : SASTRA
DCR: 1656056411
ini: 786.938 142.938 1017.938 897.938
Cipher Text: CATGCATTGCCGCAGGCCCCAGCT

--------------------------------------------------
---Decryption---
Enter Cipher Text: CATGCATTGCCGCAGGCCCCAGCT
DCR: 1656056411
Enter y1 y2 y3 y4:892.938 69.938 110.938 747.938
--------------------------------------------------
VPQPFG
--------------------------------------------------
```

In this string encryption demo, initially, the correct DCR and Keys were given for decryption, note that we get different cipher texts for different encryptions. On decrypting with the wrong keys we get a different random text.

Before Encryption, we have a file *testfile* in the local directory.
We encrypt the file, and we get the output *testfile.enc* and we decrypt the file to the output testfile.dec

```
(ai) saais@pop-os:~/.../str-file$ ls
dna.py  file_demo.py  __pycache__  string_demo.py  testfile  testing.py
(ai) saais@pop-os:~/.../str-file$ python file_demo.py
Enter your Filename : testfile
DCR: 1656056792
ini: 2.913 695.913 326.913 871.913
--------------------------------------------------
---Decryption---
Enter Cipher File: testfile.enc
DCR: 1656056792
Enter y1 y2 y3 y4:2.913 695.913 326.913 871.913
--------------------------------------------------
Check local directory for decrypted file :)
```

```
saais@pop-os:~/Desktop/dna/project/str-file$ cat testfile.enc
ATGCCTCCAATTTCGGTTACCGTCCGGTGCTATGTTGTTGATGCATGTAATCCAAACCCAGCTCGACACATC
TGATGCCTTGTCTGTCAGGCGTCGAACTCCAGCTGGGTGAAGGACAGTAAGCGCACAGAGTAGTATAAAGAG
TCGAGCCAGTTTACCTGGTATGCACGAAGGATCACATAATCAGTCTGCAGGATCCGTGGCACTACAATTTAA
TATATCGTTAATGGAATTGAAATAAAAGATAGAAGGCTCTCTGCGCATGCTGCCTGTAGGCGTAGCACGATT
CTAGTATTGTCTGAAACGACATGTATACAGACAGAGTGATCCCAAAGCAACAATTCCCGGAGCGTCAGTCCA
CTTCACGACTACATAATATCCGATGGACCCGCACGATGTGCGTCAAGACAGTAAATGACCTAGCATTTTAGA
TGACGCTCAAACAACTTTTGGTTCAAACGGAGTGATAACTTTGTCAGCAAATATATGACTAAAATTCCGAGT
TACTAAGAGGGGGGTCATGGCCTTAGAAGGTTATTCCTGGACCCACCTGTAGTTCGGTCTATGCGACCAAGT
GTGGGCTGACGTGTCGTTATTGGATAAGCACTCCAACAAGAAGGGATTAGAAGTAAACAGTGACGCACCACG
TCCTTGTTAGGCAATCGCATCCACGGTGAGAGAAAGGGTGACCATTGCGTGTGTCATGCTTATTTGCACCAG
CATACGGTGTCACCAAACGAATCTCTTCATTACATTAACCTATACGCAGGGGGCAAGTCATGACCCGGCAGA
TTCTCCAATCTCAACTAAGGTCTCGCATGCTAATAGTTCTCAAATTATACGATCTTTGTAGGTAAACGTGGT
TAGATGTCGCCGACGACGGTTGTGAAAGCAACATGTCCTTTGGCCCTAAAAAACGCATACGGTTGGTGGCTA
CGCCTGGCATGAGCGAAGCTGACATCGTTGTGCCCAATTCCATATCCAGTTATCAGCTCAAACAGTGCGACG
GCGGTATAACTGTGTACGTACAGAATCCTCTATTTCTGCCTGCACTCCTTGACAAAAACCTTGGGTTTTTAT
CGACCTATCCCTTCGCACTGCATCGCGCATAGGCGGAGTCCATAGGGGTTTCAGATCACAGTAAAGTTAGCA
CCGCCATGACTCCCCCAGACCGAAGACTTGGGGACCGGTCGTGTTGAACTTCTGAACCCAGCCGGTCGATGT
CACAGGAACTAGAAATCTGATAGTTTGGCCTCGTAACTGAAGCCCCGTTAGCAACACTGTTCACCCTTGAGG
```

We use the *diff* command in Linux with the '-s' tag to make sure we get identical files.

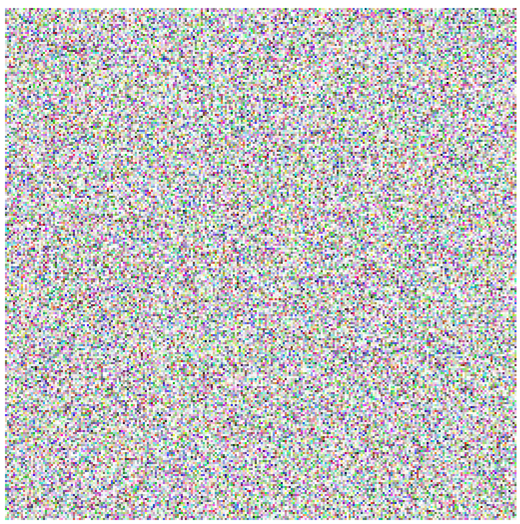
```
saais@pop-os:~/Desktop/dna/project/str-file$ diff -s testfile testfile.dec
Files testfile and testfile.dec are identical
saais@pop-os:~/Desktop/dna/project/str-file$
```

Image Encryption and Decryption

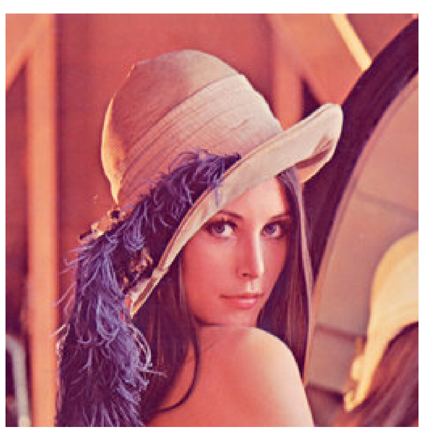
Original Image

Encrypted Image

```
(ai) saais@pop-os:~/.../imageenc$ python img_demo.py
Do you want to encrypt or decrypt?
1)Encrypt
2)Decrypt
Option: 1
---Encryption---
Enter your image name : lena.png
No of blocks (Encryption):  65536
Keys:  2482553995 2156809034 3658582091 897587765
Finished in 2.720594882965088 seconds!
---Done!---
(ai) saais@pop-os:~/.../imageenc$ python img_demo.py
Do you want to encrypt or decrypt?
1)Encrypt
2)Decrypt
Option: 2
---Decryption---
Enter Encrypted Image: lena.enc.png
Enter Keys with ' ' in between : 2482553995 2156809034 3658582091 897587765
Enter Output filename: lena.dec.png
No of blocks (Decryption):  65536
Finished in 2.8929996490478516 seconds!
---Done!---
```



Decrypted Image



Analysis of Image Encryption

Testing the involvement of the different keys in the encryption algorithm was necessary to make sure the loss of one key was not a greater breach when compared to the loss of other keys.

The following tests were performed:

1. Removal of SRand all others intact.
2. Removal of y1 and all others intact.
3. Removal of y2 and all others intact.
4. Removal of y3 and all others intact.
5. Change MSB of SR
6. Change MSB of y1
7. Change MSB of y2
8. Change MSB of y3
9. Change LSB of SR
10. Change LSB of y1
11. Change LSB of y2
12. Change LSB of y3
13. Removal of keys SR,y1 and all other intact
14. Removal of keys SR,y2 and all other intact
15. Removal of keys SR,y3 and all other intact
16. Removal of keys y1.y2 and all other intact
17. Removal of keys y1,y3 and all other intact
18. Removal of keys y2,y3 and all others intact

The Mean Square Error was calculated for all the decrypted images with the original image and only the image decrypted with the proper keys had 0 MSE and all others had an MSE of 11000 or greater.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

```
(ai) saais@pop-os:~/.../imageenc$ python analysis.py
Testing --proper-decryption--
No of blocks (Decryption):  65536
Difference from original MSE = 0.0
Time Taken 2.893301486968994

Testing --1-missing--
No of blocks (Decryption):  65536
Difference from original MSE = 12128.28825378418
Time Taken 2.8858635425567627

Testing --2-missing--
No of blocks (Decryption):  65536
Difference from original MSE = 12165.494045257568
Time Taken 2.869175434112549

Testing --3-missing--
No of blocks (Decryption):  65536
Difference from original MSE = 12134.530906677246
Time Taken 3.2639009952545166

Testing --4-missing--
No of blocks (Decryption):  65536
Difference from original MSE = 12120.782760620117
Time Taken 3.2673087120056152
```

```
Testing --1-2-missing--
No of blocks (Decryption):  65536
Difference from original MSE = 12125.035373687744
Time Taken 3.454700231552124

Testing --1-3-missing--
No of blocks (Decryption):  65536
Difference from original MSE = 12092.988353729248
Time Taken 3.35373592376709

Testing --1-4-missing--
No of blocks (Decryption):  65536
Difference from original MSE = 12110.60580444336
Time Taken 3.581437349319458

Testing --2-3-missing--
No of blocks (Decryption):  65536
Difference from original MSE = 12152.352848052979
Time Taken 3.442662477493286

Testing --2-4-missing--
No of blocks (Decryption):  65536
Difference from original MSE = 12105.640396118164
Time Taken 3.5654592514038086

Testing --3-4-missing--
No of blocks (Decryption):  65536
Difference from original MSE = 12142.216598510742
Time Taken 3.4222779273986816
```

```
Testing --1-msb--
No of blocks (Decryption):  65536
Difference from original MSE = 12142.244102478027
Time Taken 3.5448014736175537

Testing --2-msb--
No of blocks (Decryption):  65536
Difference from original MSE = 12101.699909210205
Time Taken 3.3120009899139404

Testing --3-msb--
No of blocks (Decryption):  65536
Difference from original MSE = 12169.81591796875
Time Taken 3.449822187423706

Testing --4-msb--
No of blocks (Decryption):  65536
Difference from original MSE = 11205.180965423584
Time Taken 3.330674171447754
```

```
Testing --1-lsb--
No of blocks (Decryption):  65536
Difference from original MSE = 12145.683673858643
Time Taken 3.352277994155884

Testing --2-lsb--
No of blocks (Decryption):  65536
Difference from original MSE = 12106.356063842773
Time Taken 3.169725179672241

Testing --3-lsb--
No of blocks (Decryption):  65536
Difference from original MSE = 12122.23091506958
Time Taken 3.3751981258392334

Testing --4-lsb--
No of blocks (Decryption):  65536
Difference from original MSE = 12098.647937774658
Time Taken 3.167532205581665
```

Table of MSE:

Test Case	Mean Squared Error	Time Taken for Decryption
Proper Decryption	0	2.8 seconds
Removal of SR	12128	2.8 seconds
Removal of y1	12165	2.8 seconds
Removal of y2	12134	3.2 seconds
Removal of y3	12120	3.2 seconds
Removal of SR and y1	12125	3.4 seconds
Removal of SR and y2	12092	3.3 seconds
Removal of SR and y3	12110	3.5 seconds
Removal of y1 and y2	12152	3.4 seconds
Removal of y1 and y3	12105	3.5 seconds
Removal of y2 and y3	12142	3.4 seconds
Change MSB of SR	12142	3.5 seconds
Change MSB of y1	12101	3.3 seconds
Change MSB of y2	12169	3.4 seconds
Change MSB of y3	11205	3.3 seconds
Change LSB of SR	12145	3.3 seconds
Change LSB of y1	12106	3.1 seconds
Change LSB of y2	12122	3.3 seconds
Change LSB of y3	12098	3.1 seconds

CHAPTER 5
CONCLUSION AND FUTURE WORKS

String Encryption

In this project, we used DNA encryption to encrypt and decrypt the string and image. DNA encryption uses a Moore machine and a hyperchaotic system. In the key generation, a hyperchaotic system is used to produce four random numbers which are used as keys in DNA operations. A Moore machine is generated randomly to perform substitutions on the DNA sequence and alters it making it difficult to crack. The proposed scheme is immune to various cyber attacks like brute force, ciphertext only attacks, man-in-the-middle attacks, known-plaintext attacks, attacks, and differential cryptanalysis attacks. The time taken for encryption and decryption is low compared to previous schemes. The performance of the system is tested and experimentally analyzed and the results are noted. The analysis is done to find the throughput, avalanche effect and frequency of the DNA bases in the ciphertext. The results show that the proposed scheme outperforms the existing scheme. In the future, we can extend the following algorithm to perform file encryption and image encryption, both of which we have demonstrated in our implementation. We can also try to improve the keyspace of the algorithm.

Image Encryption

The same DNA encryption scheme is used to encrypt and decrypt an image. Here also DNA encryption uses a Moore machine and a scrambling technique is used to increase the randomness of the encryption making it immune to cyber attacks. The image is divided into blocks and the encryption and decryption are done on each block and finally converging all the blocks together to get the encrypted image and original image back. Keys are generated using DCR(Dynamic Coding Rule) making it so random. Using these keys, scrambling is done. The time for encryption and decryption is also less than most image encryption techniques. The analysis is performed for encryption and decryption and found the MSE(Mean Square Error) for all test cases. The image encryption algorithm has a little imbalance in the key contribution, though it is not that obvious, it can become an easy target for experienced cryptanalysts, and we can try to make the key contribution better, by making sure all keys contribute equally to experienced cryptanalysts.

CHAPTER 6
REFERENCES

[1] S. Basu, M. Karuppiah, M. Nasipuri, A.K. Halder, N. Radhakrishnan, Bio-inspired cryptosystem with DNA cryptography and neural networks, J. Syst. Archit. 94 (2019) 24–31.

[2]S.T. Amin, M. Saeb, S. El-Gindi, A DNA-based implementation of YAEA encryption algorithm, in: Computational Intelligence, 2006, pp. 120–125.

[3] R.M. Alguliyev, R.M. Aliguliyev, L.V. Sukhostat, Efficient algorithm for big data clustering on single machine, CAAI Trans. Intell. Technol. 5 (1) (2019) 9–14.

[4] S. Namasudra, P. Roy, P. Vijayakumar, S. Audithan, B. Balusamy, Time efficient secure DNA based access control model for cloud computing environment, Future Gener. Comput. Syst. 73 (2017) 90–105.

[5] Y. Yuan, Y. Mo, Security for cyber-physical systems: Secure control against known-plaintext attack, Sci. China Technol. Sci. 63 (9) (2020) 1637–1646.

[6] S. Jain, V. Bhatnagar, A novel DNA sequence dictionary method for securing data in DNA using spiral approach and framework of DNA cryptography, in: 2014 International Conference on Advances in Engineering & Technology Research, ICAETR-2014, IEEE, 2014, pp. 1–5.

[7] A. Aich, A. Sen, S.R. Dash, S. Dehuri, A symmetric key cryptosystem using DNA sequence with OTP key, in: Information Systems Design and Intelligent Applications, Springer, 2015, pp. 207–215.

[8] H. Zhao, G. Han, L. Wang, W. Wang, MILP-based differential cryptanalysis on round-reduced midori64, IEEE Access 8 (2020) 95888–95896.

[9] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to automata theory, languages, and computation, ACM SIGACT News 32 (1) (2001) 60–65.

[10] S. Namasudra, R. Chakraborty, A. Majumder, N.R. Moparthi, Securing multimedia by using DNA-based encryption in the cloud computing environment, ACM Trans. Multimed. Comput. Commun. Appl. 16 (3s) (2020) 1–19.