
Hamiltonian
Encryption

# Objective :

Privacy is a major concern in this booming world. Every Alice and Bob thinks of "Encryption" and "Decryption" when it comes to security. But the one question that they have in mind is, which scheme would safeguard their privacy. When it comes to pictures, a picture is worth more than thousands of words, and there indeed are thousands of pixels in a single image. After multiple trials with different schemes, permutation schemes are yet the most reliable ones in use. Hamiltonian path is one such application of permutation schemes.
What makes Hamiltonian different from other schemes is that, in other permutation schemes we have to perform sorting operations but in Hamiltonian we don't need to perform sorting operations. This allows us to increase the efficiency as well.
In this project we have tried to implement image encryption using random hamiltonian paths. We have deployed a flask application for the frontend purpose and implemented the algorithm in the backend. During the process of encryption and decryption we have tried to research various mathematical applications such as Bernoulli shift maps, chaotic systems, bit plane slicing, Dirac's theorem which are a part of the process in general.

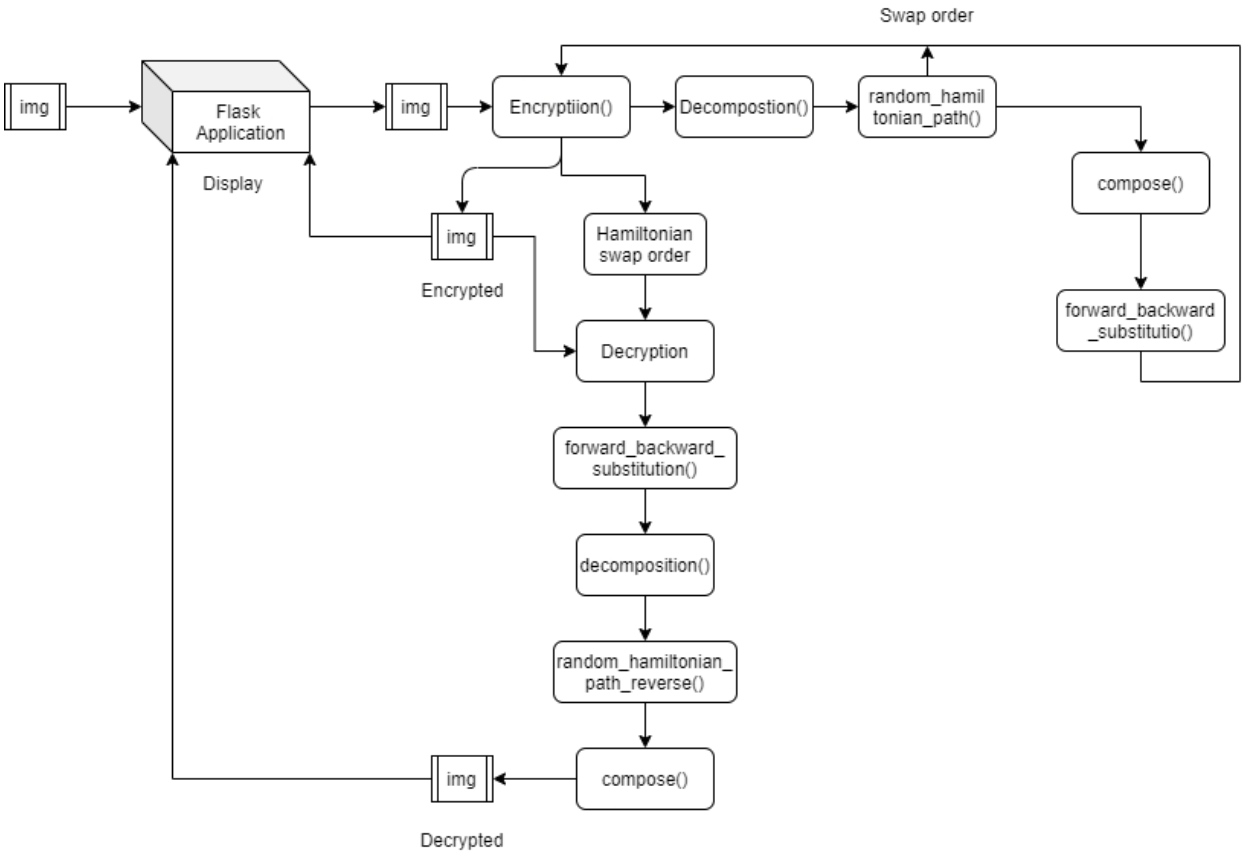
# Motive :

Our motive is to perform encryption and decryption on digital data by building a random Hamiltonian path along the pixels of the digital data, relevant to permutation schemes. Along with Hamiltonian path arrays with gray level (1D) substitution is achieved and ABM's are used to generate pseudo random numbers which play a major role in the encryption process.

# Random Hamiltonian Path :

As per the graph theory, a path which visits each vertex exactly once is referred to as Hamiltonian Path.
An image has pixels, these pixels are considered as vertices and the connections between them are considered to be edges, since Dirac's theorem states that, in a graph G of N vertices , if for each vertex v, the degree of v is greater than or equal to N/2 then at least one Hamiltonian path exists in that graph(G). The pixels of an image are in a similar manner where hamiltonian path encryption can be performed.
Adding randomness to the hamiltonian path for image encryption is done by using Bernoulli's shift maps, chaotic systems and bit plane slicing.

# Project Workflow :




(Created using draw.io)

- **Flask Application :** Flask is a framework written in Python with no requirements of database abstraction layer, form validation, or any other components.

  UI created using flask :

# Encryption Functionality :

An image X (m x n) is received from the flask application. This image is first converted into a gray scale image and then converted in a matrix. This matrix is sent to the encryption function. Firstly the image goes through a bit plane decomposition process.
Bit Plane Decomposition is a process of slicing the image at different planes (bit-planes) plays an important role in image processing. An application of this technique is data compression.
After decomposing the image to bit planes, these bit planes are composed to generate a new image (2m x 2n). Then this new image is used to iterate ABM (Adjusted Bernoulli Map) for generating pseudo random number (r)

# Adjusted Bernoulli Map :

$$x_{n+1} = \beta(\alpha x_n \bmod 1) \bmod 1$$

To generate randomness of the Hamiltonian path, chaotic maps can serve a major purpose by generating pseudo random numbers. As per our research, 1D chaotic maps are much compatible and less complex. In this encryption-decryption process we use ABM's to serve the purpose where random values of $\alpha$, $\beta$ are used to generate chaos.

Then the decomposed bit planes are merged to generate an image Y of size (mxn). Now two 1D arrays are used to generate pseudo random numbers and are quantized using ABM. These arrays are used for swapping purposes
Note : there are 2 swapping operations, one before merging of decomposed bit planes and one after merging.

$$H_{i-1} = H_{i-1} \oplus T_{S_{H_i}} \oplus (round(b_i \times 10^{14}) \bmod 256)$$

$$H_{i+1} = H_{i+1} \oplus S_{T_{H_i}} \oplus (round(a_i \times 10^{14}) \bmod 256)$$

These operations result in a cipher image or an encrypted image

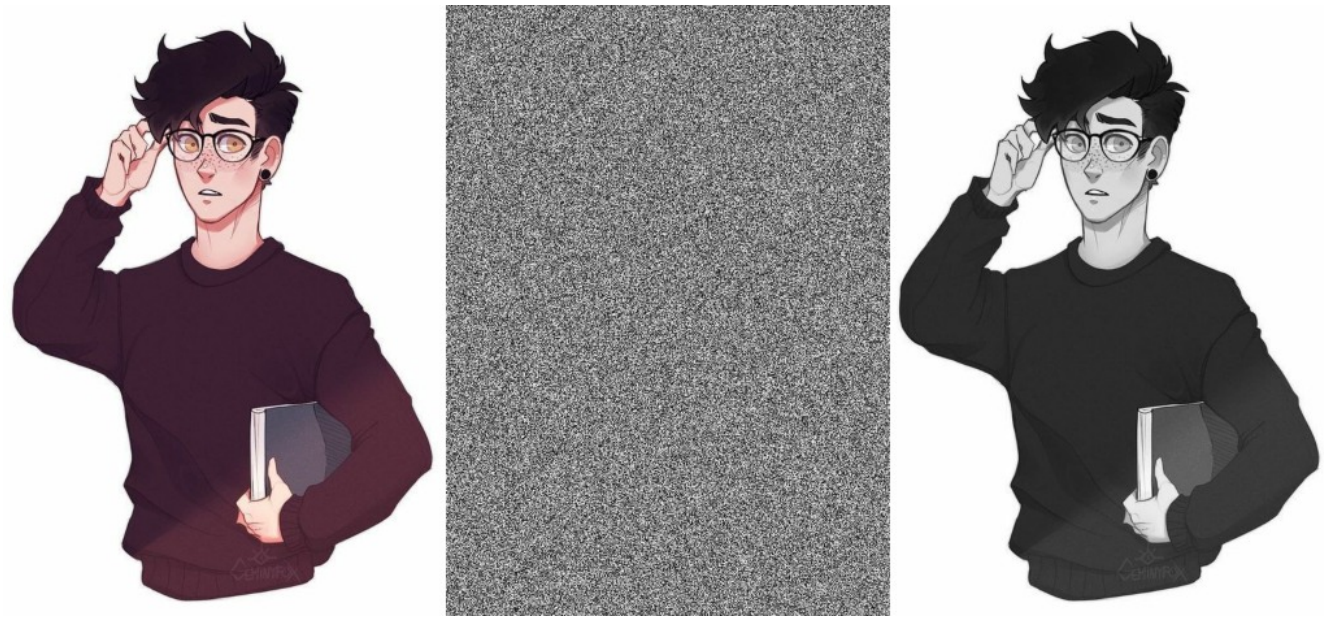
# Decryption Process :

Multiple XOR operations are performed :
- Reversed operation of backward substitution
- Reversed operation of forward substitution

Followed by bit plane decomposition, reverse operations of generation random hamiltonian path and then followed by bit plane mergence.
The output of this process results in a grayscale decrypted image.

## Flask Output :


## Encryption->Decryption :



# Conclusion :

A 1D modified Bernoulli map suitable for encryption systems is proposed in this project. A revolutionary image encryption algorithm was created based on the current chaotic map. The permutation process was accomplished by creating a random Hamiltonian path through multiple bit planes.The random Hamiltonian route was then extended for grey level substitution in the diffusion process. We have also tried to explore various chaotic systems along with Bernoulli shift maps.

# Code :

## Flask Application:

```
UPLOAD_FOLDER = "{{url_for('uploads')}}"
UPLOAD_FOLDER = "static//uploads"
ALLOWED_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'}
app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
if os.path.exists(UPLOAD_FOLDER):
    shutil.rmtree(UPLOAD_FOLDER)  # delete output folder
os.makedirs(UPLOAD_FOLDER)

def allowed_file(filename):
        return '.' in filename and \
                filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
@app.route("/")
def home():
        return render_template("index.html")
@app.route("/predict",methods=['POST','GET'])
def predict():
        if request.method=="POST":
                if 'file' not in request.files:
                        flash('No file part')
                        return redirect(request.url)
                file = request.files['file']
                if file.filename == '':
                        flash('No selected file')
                        return redirect(request.url)
                if file and allowed_file(file.filename):
                        filename = secure_filename(file.filename)
                        file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
                        x_0 = np.double(0.123456789)
                        alpha = np.double(10.45678)
                        beta = np.double(10.123)
                        img = cv2.imread("static//uploads//image.jpg", cv2.IMREAD_GRAYSCALE)
                        encrypted_img, hamiltonian_swap_order = encryption(img, x_0, alpha, beta)
                        cv2.imwrite('static//uploads//encrypt.png', encrypted_img)
```

```
                    encrypted_img = cv2.imread("static//uploads//encrypt.png", cv2.IMREAD_GRAYSCALE)
                    decrypted_img = decryption(encrypted_img, x_0, alpha, beta,
    hamiltonian_swap_order)
                    cv2.imwrite('static//uploads//decrypt.png', decrypted_img)
      return render_template("index.html",img1="static//uploads//encrypt.png", img2="static//uploads//
decrypt.png")
              # return 'ok'
if _name_ == "_main_":
      app.run(debug=True)
```

## Encryption:

```
def encryption(img, x_0, alpha, beta):
      x = x_0
      shuffled_img = decompositon(img)
      print("Decomposed")
      shuffled_img, x, hamiltonian_swap_order = random_hamiltonian_path(shuffled_img, x, alpha, beta)
      print("Shuffled")
      shuffled_img = compose(shuffled_img)
      print("Composed")
      encrypted_img, x = forward_and_backward_substitution(shuffled_img, x, alpha, beta)
      print("Substituded")
      return encrypted_img, hamiltonian_swap_order
```

## Decomposition:

```
def decompositon(img):
    newImg = np.zeros((2 * img.shape[0], 2 * img.shape[1]))
    for row in range(img.shape[0]):
        for col in range(img.shape[1]):
            newImg[row][col] = (img[row][col] >> 6) & 3
            newImg[row][col + img.shape[1]] = (img[row][col] >> 4) & 3
            newImg[row + img.shape[0]][col] = (img[row][col] >> 2) & 3
            newImg[row + img.shape[0]][col + img.shape[1]] = (img[row][col]) & 3

    return newImg
```

## Composition:

```
def compose(img):
    newImg = np.zeros((int(img.shape[0] / 2), int(img.shape[1] / 2)))
    for row in range( int(img.shape[0] / 2)):
        for col in range(int (img.shape[1] / 2)):
            x = int(img[row][col]) << 6
            y = int(img[row][col + int(img.shape[1] / 2) ]) << 4
            z = int(img[row + int(img.shape[0] / 2)][col]) << 2
            u = int(img[row + int(img.shape[0] / 2)][col + int(img.shape[1] / 2)])
            newImg[row][col] =  x + y + z + u

    return newImg
```

## Swapping:

```
def twoD2oneD(row, col, width):
        return row * width + col

def oneD2twoD(ind, width):
        return int(ind // width), int(ind % width)
```

## ABM:

```
def generate_abm(x_0, alpha, beta):
        assert type(x_0) != 'numpy.float64' or type(alpha) != 'numpy.float64' or type(beta) != 'numpy.float64',
'generate_abm(): parameter type error'
        return beta * (alpha * x_0 % 1) % 1
```

## Random Hamiltonian Path:

```
def random_hamiltonian_path(img, x_0, alpha, beta):
        img_size = img.shape
        x = x_0
        swap_order = []
        for row in range(img_size[0]-1, -1, -1):
                for col in range(img_size[1]-1, -1, -1):
                        if row == 0 and col == 0:
                                break
                        x = generate_abm(x, alpha, beta)
                        swap_row, swap_col = oneD2twoD(int(round(x * (10**14))) % twoD2oneD(row, col,
img_size[1]), img_size[1])
                        img[swap_row][swap_col], img[row][col] = img[row][col], img[swap_row][swap_col]
                        swap_order.append([swap_row, swap_col])
        return img, x, swap_order
```

## Forward and Backward Substitution:

```
def forward_and_backward_substitution(img, x_0, alpha, beta):
        x = x_0
        s_list = [i for i in range(256)]
        for i in range(255, 0, -1):
                x = generate_abm(x, alpha, beta)
                swap_index = int(int(round(x * (10**14))) % i)
                s_list[i], s_list[swap_index] = s_list[swap_index], s_list[i]
        t_list = [i for i in range(256)]
        for i in range(255, 0, -1):
                x = generate_abm(x, alpha, beta)
                swap_index = int(int(round(x * (10**14))) % i)
                t_list[i], t_list[swap_index] = t_list[swap_index], t_list[i]

        img_size = img.shape
```

```
        for i in range(1, img_size[0] * img_size[1]):
                row, col = int(i//img_size[1]), int(i%img_size[1])
                prev_row, prev_col = (i-1)//img_size[1], (i-1)%img_size[1]
                x = generate_abm(x, alpha, beta)
                xor_value = int(int(round(x * (10**14))) % 256)
                img[row][col] = float(int(img[row][col]) ^ (s_list[t_list[int(img[prev_row][prev_col])]]) ^
xor_value)
        for i in range(img_size[0] * img_size[1]-1, 0, -1):
                cur_row, cur_col = int(i//img_size[1]), int(i%img_size[1])
                prev_row, prev_col = int((i-1)//img_size[1]), int((i-1)%img_size[1])
                x = generate_abm(x, alpha, beta)
                xor_value = int(int(round(x * (10**14))) % 256)
                img[prev_row][prev_col] = float(int(img[prev_row][prev_col]) ^ (t_list[s_list[int(img[cur_row]
[cur_col])]]) ^ xor_value)

        return img, x
```

## Reverse Forward and Backward Substitution:

```
def forward_and_backward_substitution_reverse(img, x_0, alpha, beta):
    x = x_0
    img_size = img.shape
    # let x can be used correctly
    for row in range(img_size[0]*2-1, -1, -1):
        for col in range(img_size[1]*2-1, -1, -1):
            if row == 0 and col == 0:
                break
            x = generate_abm(x, alpha, beta)

    s_list = [i for i in range(256)]
    for i in range(255, 0, -1):
        x = generate_abm(x, alpha, beta)
        swap_index = int(int(round(x * (10**14))) % i)
        s_list[i], s_list[swap_index] = s_list[swap_index], s_list[i]

    t_list = [i for i in range(256)]
    for i in range(255, 0, -1):
        x = generate_abm(x, alpha, beta)
        swap_index = int(int(round(x * (10**14))) % i)
        t_list[i], t_list[swap_index] = t_list[swap_index], t_list[i]

    x_list_forward = []
    x_list_backward = []
    for i in range(img_size[0] * img_size[1]-1, 0, -1):
        x = generate_abm(x, alpha, beta)
        x_list_forward.append(x)

    for i in range(1, img_size[0] * img_size[1]):
        x = generate_abm(x, alpha, beta)
```

```
        x_list_backward.append(x)

    for i in range(1, img_size[0] * img_size[1]):
        cur_row, cur_col = int(i//img_size[1]), int(i%img_size[1])
        prev_row, prev_col = int((i-1)//img_size[1]), int((i-1)%img_size[1])
        xor_value = int(int(round(x_list_backward[len(x_list_backward)-i] * (10**14))) % 256)
        img[prev_row][prev_col] = float(int(img[prev_row][prev_col]) ^ (t_list[s_list[int(img[cur_row][cur_col])]])
^ xor_value)

    for i in range(img_size[0] * img_size[1]-1, 0, -1):
        row, col = int(i//img_size[1]), int(i%img_size[1])
        prev_row, prev_col = (i-1)//img_size[1], (i-1)%img_size[1]
        xor_value = int(int(round(x_list_forward[i-1] * (10**14))) % 256)
        img[row][col] = float(int(img[row][col]) ^ (s_list[t_list[int(img[prev_row][prev_col])]]) ^ xor_value)

    return img, x
```

## Reverse Random Hamiltonian Path:

```
def random_hamiltonian_path_reverse(img, swap_order):
    img_size = img.shape
    count = len(swap_order)
    for row in range(0, img_size[0]):
        for col in range(0, img_size[1]):
            if(row == 0 and col == 0):
                continue
            swap_row = swap_order[count-1][0]
            swap_col = swap_order[count-1][1]
            img[swap_row][swap_col], img[row][col] = img[row][col], img[swap_row][swap_col]
            count -= 1

    return img
```

## Decryption:

```
def decryption(img, x_0, alpha, beta, hamiltonian_swap_order):
        x = x_0
        decryption_img, x = forward_and_backward_substitution_reverse(img, x, alpha, beta)
        print('Substituded reverse')
        shuffled_img = decompositon(decryption_img)
        print('Decomposed')
        shuffled_img = random_hamiltonian_path_reverse(shuffled_img, hamiltonian_swap_order)
        print('Shuffled reverse')
        original = compose(shuffled_img)
        print(''Composed'')
        return original
```

## Future Scope :

To understand the dynamics of chaotic systems and work on 3 DIMENSIONAL Digital data