Image Steganography Using an Edge Based Embedding Technique

A PROJECT REPORT

Submitted by

18BCI0268 – Nitish Kumar Gupta
20BKT0056 – Shubhlaxh Porwal
20MID0175 – B Lalith Yashasvin

Course Code: BCI3005

Course Title: Digital Watermarking and Steganography

Under the guidance of

Prof. Jasmin T Jose

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

INDEX

Abstract

Steganography is a method of hiding secret data, by embedding it into an audio, video, image or text file, called a carrier or data carrier, more specifically. It is one of the methods employed to protect secret or sensitive data from malicious attacks. Cryptography and steganography are both methods used to hide or protect secret data. However, they differ in the respect that cryptography makes the data unreadable, or hides the meaning of the data, while steganography hides the existence of the data.

The pixels are ever so slightly added with a little bit of information that barely changes the look of the picture. In image Steganography – The image selected for this purpose is called the cover-image and the image obtained after steganography is called the stego-image.

A novel data hiding technique based on the LSB (Least Significant Bit) technique and edge based algorithms of digital images is the aim of the project. Moreover, username psedonymization, age and zip code generalization and traditional AES encryption will be performed as well to preserve user privacy and add another layer of security.

1. Introduction

A secret message is embedded onto an image and sent over a network; the image has no change in itself although but contains the hidden message. In this way the content can be stored in a secure environment. Here, users can create their accounts in which they can add the data to be shared is first compressed, encrypted, and then embedded in a carrier file (an image in our case). It provides a multiple of layers of protection of the data, and not only is the data encrypted, if the attacker manages to break the first layer of encryption, the information won't make any sense, as it'll just be the carrier file. If they manage to somehow retrieve the data, there's still another layer of encryption to break.

The aim of the project is to design an image embedding technique based on LSB technique and edge based embedding that is robust and efficient while also being secure for storing files as images while preserving user privacy. Moreover, another layer of security has to be added to further secure the data.

2. Literature Survey

Title: Hiding Data Using Efficient Combination of RSA Cryptography and Compression Steganography Techniques [1]

Authors: Hesham F. A. Hamed, Aziza I. Hussein, Ashraf A. M. Khalaf, Osama, Fouad Abdel Wahab

The proposed algorithm, in this case, is combining RSA and Huffman coding, or DWT with the intention of reducing the information's bit in steganography. Two key processes are involved, embedding the information and also getting or extracting the message. Key sharing is an essential part in this method. In this paper, a combination of RSA and Huffman coding has been carefully proposed as a method of securing and compressing messages, and even masking messages in the cover image, with the aim of producing a

high quality image with a small size. The experimental results indicate that the proposed mechanism has an effective visual quality and storage capacity.

Title: A comparative study of recent steganography techniques for multiple image formats [2]

Authors: Arshiya Sajid Ansari, Mohammed Sajid Mohammadi, Mohammed Tanvir Parvez

A number of reported works on Image Steganography have been reviewed and the methods are categorized based on cover image formats like JPEG, RGB and PNG and their domain information. This paper reviewed the background details of Steganography algorithms. The technical properties infer that the JPEG (DCT/DWT) algorithms are more immune to attack and provide high resistance to Steganalysis because the coefficients get modified in the transform domain. However, it comes at the price of lossy embedding.

Title: Enhanced Least Significant Bit Replacement Algorithm in Spatial Domain of Steganography Using Character Sequence Optimization [3]

Authors: Jagan Raj, Jayapandiyan, C. Kavitha, K. Sakthivel

All other spatial domain techniques, use one to one byte representation of secret message, which means each byte in the secret message is embedded as a byte in cover image or the change in position of the embedding based on the selected color model. This proposed algorithm works on the spatial domain of image steganography process primarily focuses on optimizing the way that the secret message messages are being embedded. It is evident that the proposed eLSB (enhanced Least Significant Bit) replacement algorithm is giving better PSNR, MSE and RMSE values, which in turn conveys the quality of the cover image is undergoing lesser changes compared to any of the traditional LSB algorithm in steganography.

Title: Hiding Data in Images Using Steganography techniques with compression algorithms [4]

Authors: Abdel Wahab, Aziza I. Hussain, Hesham F.A. Hamed, Hamdy M. Kelash, Ashraf A.M. Khalaf, Hanafy M. Ali

They applied multiple methods to hiding image by applying DCT algorithm to embed the image and encrypt the data via cryptography algorithms, applying DCT compression. It is assumed that the sender as well as the receiver holds the same system of private keys. The performance of a comparison between two different techniques is given. The first technique used LSB with no encryption and no compression. In the second technique, the secret message is encrypted first then LSB technique is applied. It is clear that they can hide the intended data in messages while minimizing its size.

3. Overview of the proposed work

3.1. Objectives of the Project

The aim of the project is to combine traditional encryption with image steganography and designing a robust and efficient embedding algorithm. The user credentials have to be pseudonymized and their details have to be generalized before embedding. Text compression to reduce the number of bits to be processed and embedded has also been planned.

3.2. Software Requirements

Windows OS/ Mac OS/Linux OS

Python 3.6 or above

Python package opencv to read and write image pixels

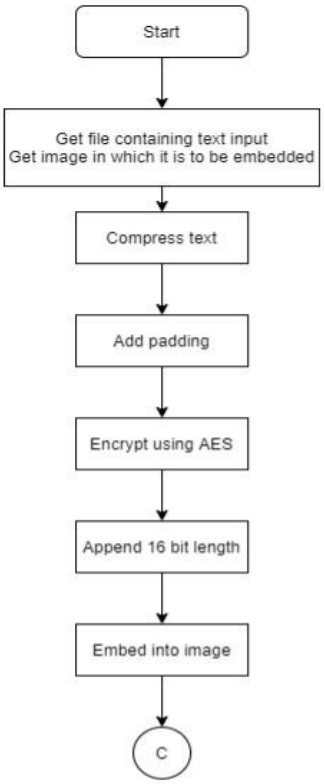
3.3. Hardware Requirements

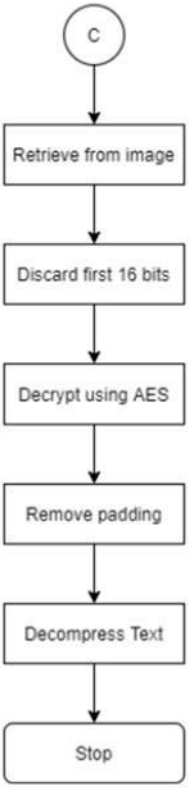
1 GHz CPU, 4 GB RAM

4. System Design

Modules:

1. Text compression
2. AES encryption
3. Proposed text embedding algorithm
4. Proposed text retrieving algorithm
5. AES decryption
6. Text decompression





Algorithms Used:

Huffman Coding for text compression and decompression

AES for traditional encryption and decryption

Custom technique for embedding

5. IMPLEMENTATION

5.1. Description of Modules/Programs

Text Compression:

Huffman coding has been used for this purpose. Frequency of occurrence of all characters in input is calculated and placed in bottom of a tree. Then, the two nodes with lowest frequencies are taken and added to make a node. This process is repeated until only one node is left (root node). From each node, the node to the left is marked 0 and right is marked 1.

Then each character's code is path from root node to the character.

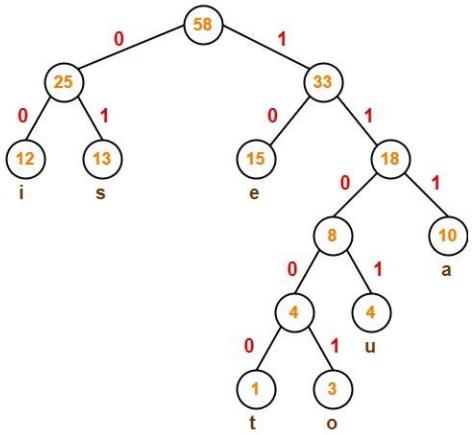


Figure 5a

In the above figure 5a, ‘i’ will be represented by 00, ‘o’ will be represented as 11001.

AES Encryption:

The compressed text is padded so that it is in form of 120 mod 128. The number of bits padded are counted and converted to 8 bit binary. This is appended at beginning of data so that we know how many padding bits are there in the text. The data now becomes a multiple of 128. The text is divided into 128 bit blocks and encrypted with 10 rounds in ECB mode. The results of the encryption are appended to get encrypted text.

Proposed Text Embedding Algorithm:

For an image, some terminology is defined. The top side is side 0 and increments in clockwise direction. Each side also has a left and right indicator that is relative to facing outwards from that side. The sides are also represented in 2 bit binary as 00, 01, 10 and 11. Three corners starting from top left corner are marked as 0, 1 and 2 as seen in figure 5b.

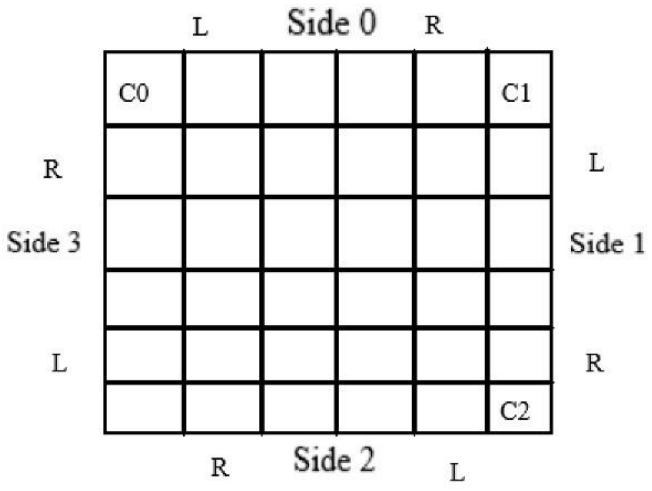


Figure 5b

3 unique integers are randomized between 0 and 3 to get 3 sides. 3 integers between 0 and 1 are randomized to get mode for these 3 sides. Mode = 0 denotes that embedding has to be carried out from left to right indicator for that side. Mode = 1 denotes that embedding has to be carried out from right to left indicator for that side. Three 3 bit binary numbers are generated from this information.

For example: if one result is side = 0, mode = 1, then the binary number corresponding to it is 001

For each of these binary numbers, the 3 bits are embedded in the LSBs of the 3 channels in the corner pixels at C0, C1 and C2 respectively. This is useful later while retrieving. The sides generated denote the order in which data has to be embedded into them.

For example: if 3, 0, 2 are 3 sides generated; then embedding order would be [3, 0, 2, 1]. The final side would just be the remaining one and its mode would be defaulted to 0.

Now, we have the order of sides in which data has to be embedded and operation modes of all of them, so we can start embedding the input. No data has to be embedded in the corner pieces. The length of the input is found, converted to a 16 bit binary and then appended in the beginning. The first bit goes into starting pixel's 1st channel of first side in the order, second into starting pixel's 1st channel of second side in the order and so on.

After 4 bits, we move to 2nd channel, then after another 4 we move to the 3rd channel. After that, we move to next pixel in that side according to operation mode. When all pixels in any side are filled, we move one edge inwards, now we have 4 new corners so we have to generate a new side order and operation modes and repeat the process. This continues until all bits are embedded.

Proposed Text Retrieving Algorithm:

While embedding, we had stored the side order and modes in 3 corners. First, that information in retrieved and 4th side is the side not contained in it and has mode 0.

The bits are extracted similar to the embedding process, going through the sides according to order and moving to next pixels in side according to the operation mode. If we reach the end of any side, then we have to go one edge inwards and get the side order and modes corresponding to that edge again. While embedding, we had padded the length of the message in front of the input by converting it into 16 bit binary. So after first 16 bits are retrieved, convert it to decimal. That will provide how many more bits are left to be retrieved. Once these bits are retrieved, we have successfully recovered the embedded bits. From the recovered data, discard first 16 bits as they are no longer useful. The resultant length would be a multiple of 128.

AES Decryption:

The text is divided into 128 bit blocks and decrypted with 10 rounds in ECB mode. The results of the encryption are appended to get decrypted text.

Text Decompression:

The first 8 bits are converted to decimal to get the number of bits that were padded. These padded bits are removed from the end of the data. Current node is set to root node. The data is traversed, if 0 occurs we go left in the tree otherwise we go right in the tree. If a leaf node occurs we have obtained a character, then we go back to root node and continue.

For example: In figure 5a, if data is 1111101, start traversing from root node. We go right then right again then right again. It is a leaf so ‘a’ is printed and go back to root. Now go right, right, left, right. It is a leaf so ‘u’ is printed.

5.2. Source Code

AES.py Module:

```
from copy import copy

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)

inv_sbox = [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]

Rcon = (0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,

0x9a) def xor(s1,

s2):


    return tuple(a ^ b for a, b in zip(s1, s2)) def

xor_str(s1, s2):


    return tuple(int(a, 16) ^ int(b, 16) for a, b in zip(s1, s2)) def

xor_str_int(s1, s2):

    return tuple(int(a, 16) ^ b for a, b in zip(s1, s2)) def

str_to_hex(st):


    return list(map(lambda x1: int(x1, 16), st)) def

str_to_hex_arr(stx):


    return [list(map(lambda x1: int(x1, 16), st)) for st in stx] def

int_to_hex(st):


    return list(map(lambda x1: format(x1, '02x'), st)) def

split_string(n, st):


    lst = [""]

    for i1 in str(st):

        l = len(lst) - 1

        if len(lst[l]) < n:

            lst[l] += i1

        else:

            lst   +=   [i1]

return       lst       def

left_shift(pos, arr):


    left = arr[pos:]
```

```
    right = arr[:pos]    return left + right

def  right_shift(pos,  arr):          return

left_shift(len(arr)   -   pos,   arr)   def

byte_left_shift(word):               return

left_shift(1,           word)           def

substitute_sbox(word):


    # hex(Sbox[int("20",16)])

    newS = []

    for byt in word:

        newS.append(hex(Sbox[int(byt, 16)])[2:])

    return       newS       def

substitute_inv_sbox(word):


    # hex(Sbox[int("20",16)])
    newS = []

    for byt in word:

        newS.append(hex(inv_sbox[int(byt, 16)])[2:])

    return         newS         def

apply_RoundCONstant(word, i1):


    return int_to_hex(xor_str_int(word, (Rcon[i1], 0, 0, 0))) def

apply_round_key(key, plainT):


    newS = []

    for x1, y in zip(key, plainT):

        newS.append(list(xor_str(x1, y)))

    return newS def

to_matrix(X):


    return [[X[j][i1] for j in range(len(X))] for i1 in range(len(X[0]))] def

to_arr(X):
```

```
    return [[X[i1][j] for j in range(len(X))] for i1 in range(len(X[0]))] def

flatten(state):

    newS = []

    for x1 in state:

        for y in x1:

            newS.append(y)

return       newS       def

unflatten(state):

    newS = []

    for x1 in range(0, 4):

        newS.append(state[x1 * 4:(x1 + 1) * 4])

    return  newS  def

shift_rows(arr):

    for x1 in range(0, 4):

        arr[x1] = left_shift(x1, arr[x1])

    return     arr     def

inv_shift_rows(arr):

    for x1 in range(0, 4):
        arr[x1] = right_shift(x1, arr[x1])

    return arr def

pprint(arr):

    for x1 in arr:

        print(int_to_hex(x1))

#  Galois  Multiplication  #

noinspection    PyUnusedLocal

def galoisMult(a, b):

    p = 0

    hiBitSet = 0
```

```
    for i1 in range(8):

        if b & 1 == 1:

            p ^= a

        hiBitSet = a & 0x80

        a <<= 1

        if hiBitSet == 0x80:

a ^= 0x1b

b >>= 1      return p %

  256               def

  mixColumn(column):


    temp = copy(column)

    column[0]  =  galoisMult(temp[0],  2)  ^  galoisMult(temp[3],  1)  ^  galoisMult(temp[2],  1)  ^
galoisMult(temp[1], 3)

    column[1]  =  galoisMult(temp[1],  2)  ^  galoisMult(temp[0],  1)  ^  galoisMult(temp[3],  1)  ^
galoisMult(temp[2], 3)

    column[2]  =  galoisMult(temp[2],  2)  ^  galoisMult(temp[1],  1)  ^  galoisMult(temp[0],  1)  ^
galoisMult(temp[3], 3)

    column[3] = galoisMult(temp[3], 2) ^ galoisMult(temp[2], 1) ^ galoisMult(temp[1], 1) ^
galoisMult(temp[0], 3) def mixColumnInv(column):

    temp = copy(column)

    column[0]  =  galoisMult(temp[0],  14)  ^  galoisMult(temp[3],  9)  ^  galoisMult(temp[2],  13)  ^
galoisMult(temp[1], 11)
column[1] = galoisMult(temp[1], 14) ^ galoisMult(temp[0], 9) ^ galoisMult(temp[3], 13) ^
galoisMult(temp[2], 11)

    column[2]  =  galoisMult(temp[2],  14)  ^  galoisMult(temp[1],  9)  ^  galoisMult(temp[0],  13)  ^
galoisMult(temp[3], 11)

    column[3] = galoisMult(temp[3], 14) ^ galoisMult(temp[2], 9) ^ galoisMult(temp[1], 13) ^
galoisMult(temp[0], 11) def mixColumns(state):

    for i1 in range(4):

        column = []

        # create the column by taking the same item out of each "virtual" row

        for j in range(4):

            column.append(state[j * 4 + i1])

        # apply mixColumn on our virtual column

        mixColumn(column)
```

```
        # transfer the new values back into the state table
        for j in range(4):
            state[j * 4 + i1] = column[j]
    return    state    def
mixColumnsInv(state):

    for i1 in range(4):
        column = []
        # create the column by taking the same item out of each "virtual" row
        for j in range(4):
            column.append(state[j * 4 + i1])
        mixColumnInv(column)
        for j in range(4):
            state[j * 4 + i1] = column[j]
        return       state       input_key       =
"21a8617473206d79204b756e6720c671"   input_key   =
input_key.rstrip().replace("   ",   "")   words   =
split_string(8, input_key) words = [split_string(2,
word) for word in words] input_plain = ""
input_plain = input_plain.rstrip().replace(" ", "").lower()
plain = split_string(8, input_plain) plain = [split_string(2,
word) for word in plain] def key_expansion_core(word, i1):

    leftShift = byte_left_shift(word)
    applied_sbox = substitute_sbox(leftShift)
    applied_roundConstant = apply_RoundCONstant(applied_sbox, i1)
    return  applied_roundConstant
key_round = 0 for i in range(4,
48):

    if i % 4 != 0:
        generated_word = xor_str(words[i-4], words[i-1])
```

```
        words.append(int_to_hex(generated_word))
    else:
        generated_word = xor_str(words[i-4], key_expansion_core(words[i-1], key_round+1))
        words.append(int_to_hex(generated_word))
        key_round += 1 new =
[]  keys  =  []  for  x  in
range(len(words)):

    if x == 0 or x % 4 == 0:
        count = 0
        # print(new)
        if len(new):
            keys.append(new)
            final = [''.join(x) for x in new]
            final = ''.join(final)
            spaced = split_string(2, final)
            final = ' '.join(spaced)
        new         =         []
new.append(words[x])         def
aes_round(state,      roundKey):
state                          =
substitute_sbox(int_to_hex(flatt
en(state)))

    state = shift_rows(to_matrix(unflatten(state)))
    state = str_to_hex_arr(state)
    state = mixColumns(flatten(state))
    state = unflatten(int_to_hex(state))
    state = apply_round_key(roundKey, state)
    return           state           def
inverse_aes_round(state, roundKey):
```

```
    state = unflatten(int_to_hex(flatten(state)))
    state = apply_round_key(roundKey, state)
    state = flatten(state)
    state = mixColumns(state)
    state = unflatten(state)
    state = shift_rows(to_matrix(state))
    state = substitute_inv_sbox(int_to_hex(flatten(state)))
    return state
```

Main.py:

```
import cv2  # for reading and writing image data
import heapq  # for heap data structure import os
# to work with files import random  # for
randomisation from AES import * compTextRef = ""
side0Mode = 0 side1Mode = 0 side2Mode = 0
side3Mode = 0 embeddedBits = 0  retrievedBits = 0
retrievedTextRef = "" retrievedTextLength = 0
lengthRetrieved = False def
make_frequency_dict(text):

    frequency = {}
    for character in text:
        if character not in frequency:
            frequency[character] = 0
        frequency[character] += 1
return frequency def
pad_encoded_text(encoded_text):

    extra_padding = 120 - len(encoded_text) % 128
    for i1 in range(extra_padding):
        encoded_text += "0"
    padded_info = "{0:08b}".format(extra_padding)
```

```
    encoded_text = padded_info + encoded_text

    return encoded_text
def get_byte_array(padded_encoded_text):

    if len(padded_encoded_text) % 8 != 0:

        print("Error occurred while padding")

        exit(0)

    b = bytearray()

    for i1 in range(0, len(padded_encoded_text), 8):

        byte1 = padded_encoded_text[i1:i1 + 8]

        b.append(int(byte1, 2))

    return b def

remove_padding(padded_encoded_text):


    padded_info = padded_encoded_text[:8]

    extra_padding = int(padded_info, 2)

    padded_encoded_text = padded_encoded_text[8:]

    encoded_text = padded_encoded_text[:-1 * extra_padding]

    return encoded_text class

HuffmanCoding:


    def __init__(self, path):

        self.path = path

        self.heap = []

        self.codes = {}

        self.reverse_mapping = {}

    # noinspection PyTypeChecker

    class HeapNode:

        def __init__(self, char, freq):

            self.char = char

            self.freq = freq

            self.left = None

            self.right = None
```

```
    # defining comparators less_than and equals

    def __lt__(self, other):

        return self.freq < other.freq

    def __eq__(self, other):
        if other is None:

            return False

        if not isinstance(other, self):

            return False

        return self.freq == other

# functions for compression

def make_heap(self, frequency):

    for key in frequency:

        node = self.HeapNode(key, frequency[key])

        heapq.heappush(self.heap, node)

def merge_nodes(self):

    while len(self.heap) > 1:

        node1 = heapq.heappop(self.heap)

        node2 = heapq.heappop(self.heap)

        merged = self.HeapNode(None, node1.freq + node2.freq)

        merged.left = node1

        merged.right = node2

        heapq.heappush(self.heap, merged)

def make_codes_helper(self, root, current_code):

    if root is None:

        return

    if root.char is not None:

        self.codes[root.char] = current_code

        self.reverse_mapping[current_code] = root.char

        return

    self.make_codes_helper(root.left, current_code + "0")

    self.make_codes_helper(root.right, current_code + "1")
```

```
def make_codes(self):

    root = heapq.heappop(self.heap)

    current_code = ""

    self.make_codes_helper(root, current_code)

def get_encoded_text(self, text):
    encoded_text = ""

    for character in text:

        encoded_text += self.codes[character]

    return encoded_text

def compress(self):

    global compTextRef

    filename, file_extension = os.path.splitext(self.path)

    outputPath = filename + ".bin"

    with open(self.path, 'r+') as file1, open(outputPath, 'wb') as output:

        text = file1.read()

        text = text.rstrip()

        print("Bits in input text: ", len(text) * 8, "bits")

        frequency = make_frequency_dict(text)

        self.make_heap(frequency)

        self.merge_nodes()

        self.make_codes()

        encoded_text = self.get_encoded_text(text)

        padded_encoded_text = pad_encoded_text(encoded_text)

        padded_encoded_text = padded_encoded_text

        compTextRef = padded_encoded_text

        b = get_byte_array(padded_encoded_text)

        output.write(bytes(b))

    return outputPath

# functions for decompression

def decode_text(self, encoded_text):

    current_code = ""
```

```
        decoded_text = ""

        for bit in encoded_text:

            current_code += bit

            if current_code in self.reverse_mapping:

                character = self.reverse_mapping[current_code]

                decoded_text += character
                current_code = ""

        return decoded_text

    def decompress(self, input_path):

        filename, file_extension = os.path.splitext(self.path)

        outputPath = filename + "_decompressed" + ".txt"

        with open(input_path, 'rb') as file1, open(outputPath, 'w') as output:

            bit_string = ""

            byte1 = file1.read(1)

            while len(byte1) > 0:

                byte1 = ord(byte1)

                bits1 = bin(byte1)[2:].rjust(8, '0')

                bit_string += bits1

                byte1 = file1.read(1)

            encoded_text = remove_padding(bit_string)

            decompressed_text = self.decode_text(encoded_text)

            output.write(decompressed_text)

        return outputPath def

randomiseSides():


    n = []

    count1 = 0

    keepLooping = True

    sideData = []

    while keepLooping:  # randomise 3 sides

        temp1 = random.randint(0, 3)

        if temp1 not in n:
```

```
            n.append(temp1)

            count1 += 1

        if count1 == 3:

            keepLooping = False

    for i1 in n:

        mode = random.randint(0, 1)

        if i1 == 0:
            sideData.append([0, 0, mode])

        elif i1 == 1:

            sideData.append([0, 1, mode])

elif i1 == 2:


            sideData.append([1, 0, mode])

        else:

            sideData.append([1, 1, mode])

    return sideData def

getBinary(x1):


    s = bin(x1)[2:]

    while len(s) < 8:

        s = '0' + s

return s def

getDecimal(s):


    a = (int(s[0], 2), int(s[1], 2), int(s[2], 2))

    return a def changePixel(row,

col, data):


    global img

    (b, g, r) = img[row][col]

    s = [getBinary(b), getBinary(g), getBinary(r)]

    for i1 in range(3):

        temp1 = list(s[i1])
```

```
        temp1[7] = str(data[i1])

        s[i1] = "".join(temp1)

img[row][col] = getDecimal(s) def

assignSideInfo(currRow, sideData):


    global rows, columns

    changePixel(currRow, currRow, sideData[0])

    changePixel(currRow, columns - currRow - 1, sideData[1])

changePixel(rows - currRow - 1, columns - currRow - 1, sideData[2]) def

assignMode(side, mode):


    global side0Mode, side1Mode, side2Mode, side3Mode
    if side == 0:

        side0Mode = mode

elif side == 1:


        side1Mode = mode

elif side == 2:


        side2Mode = mode     else:

side3Mode = mode def side0(pixel, channel,

data, currRow):


    global columns, img

    tRow = currRow

    if side0Mode == 0:

        tCol = currRow + 1 + pixel

    else:

        tCol = columns - currRow - 2 - pixel

    temp1 = img[tRow][tCol][channel] % 2

    if temp1 != int(data):

        if temp1 == 0:

            img[tRow][tCol][channel] += 1

        else:
```

```
            img[tRow][tCol][channel] -= 1 def
side1(pixel, channel, data, currRow):

    global rows, columns, img
    tCol = columns - currRow - 1
    if side1Mode == 0:
        tRow = currRow + 1 + pixel
    else:
        tRow = rows - currRow - 2 - pixel
    temp1 = img[tRow][tCol][channel] % 2
    if temp1 != int(data):
        if temp1 == 0:
            img[tRow][tCol][channel] += 1         else:
            img[tRow][tCol][channel] -= 1 def
side2(pixel, channel, data, currRow):

    global rows, columns, img
    tRow = rows - currRow - 1
    if side2Mode == 0:
        tCol = columns - currRow - 2 - pixel
    else:
        tCol = currRow + 1 + pixel
    temp1 = img[tRow][tCol][channel] % 2
    if temp1 != int(data):
        if temp1 == 0:
            img[tRow][tCol][channel] += 1
        else:
            img[tRow][tCol][channel] -= 1 def
side3(pixel, channel, data, currRow):

    global rows, img
    tCol = currRow
```

```
    if side3Mode == 0:
        tRow = rows - currRow - 2 - pixel
    else:
        tRow = currRow + 1 + pixel
    temp1 = img[tRow][tCol][channel] % 2
    if temp1 != int(data):
        if temp1 == 0:
            img[tRow][tCol][channel] += 1
else:

            img[tRow][tCol][channel] -= 1 def
embedEdge(currRow):

    global compTextRef, embeddedBits, rows, img
    sides = []
    bitsThisIter = 0
    (b, g, r) = img[currRow, currRow]
    s = [getBinary(b), getBinary(g), getBinary(r)]
    side = int(s[0][7] + s[1][7], 2)
    sides.append(side)
    assignMode(side, int(s[2][7]))
    (b, g, r) = img[currRow, columns - currRow - 1]
    s = [getBinary(b), getBinary(g), getBinary(r)]
    side = int(s[0][7] + s[1][7], 2)
    sides.append(side)
    assignMode(side, int(s[2][7]))
    (b, g, r) = img[rows - currRow - 1, columns - currRow - 1]
    s = [getBinary(b), getBinary(g), getBinary(r)]
    side = int(s[0][7] + s[1][7], 2)
    sides.append(side)
    assignMode(side, int(s[2][7]))
    for i1 in range(4):
```

```
        if i1 not in sides:
            sides.append(i1)
            break
    print("Sides embedding order:", sides)
    print("Modes of side 0 to 3:", side0Mode, side1Mode, side2Mode, side3Mode)
    spaceAvailable = (rows - (currRow + 1) * 2) * 3 * 4
    while True:
        if spaceAvailable < 64:
            if spaceAvailable >= len(compTextRef) - embeddedBits:
                n = spaceAvailable // 4
            else:
                return True
        else:
            n = 16
        for i1 in range(n):
            for j1 in range(4):
                if embeddedBits < len(compTextRef):
                    temp1 = bitsThisIter // 4
                    if sides[j1] == 0:
                        side0(temp1 // 3, temp1 % 3, compTextRef[embeddedBits], currRow)
elif sides[j1] == 1:
                        side1(temp1 // 3, temp1 % 3, compTextRef[embeddedBits], currRow)
elif sides[j1] == 2:
                        side2(temp1 // 3, temp1 % 3, compTextRef[embeddedBits], currRow)
else:
                        side3(temp1 // 3, temp1 % 3, compTextRef[embeddedBits], currRow)
                    embeddedBits += 1
                    bitsThisIter += 1
                    spaceAvailable -= 1
```

```
                else:
return False def embed():

    currRow = 0
    keepLooping = True
    while keepLooping:
        sideData = randomiseSides()
        assignSideInfo(currRow, sideData)
        keepLooping = embedEdge(currRow)
        if embeddedBits >= len(compTextRef):
            keepLooping = False
        currRow += 1
        print('Edge', currRow, 'filled')
print('Bits embedded:', embeddedBits, 'bits') def
getSide(a, b):     if a == 0:

        if b == 0:
            return 0
        else:
            return 1
    else:
        if b == 0:
            return 2
else:             return 3
def getSideData(currRow):

    global img, side0Mode, side1Mode, side2Mode, side3Mode, columns, rows
    sides = []
    (b, g, r) = img[currRow][currRow]
    side = getSide(b % 2, g % 2)
    sides.append(side)
    assignMode(side, r % 2)
    (b, g, r) = img[currRow][columns - currRow - 1]
```

```
    side = getSide(b % 2, g % 2)
    sides.append(side)
    assignMode(side, r % 2)
    (b, g, r) = img[rows - currRow - 1][columns - currRow - 1]
    side = getSide(b % 2, g % 2)
    sides.append(side)
    assignMode(side, r % 2)
    for i1 in range(4):
        if i1 not in sides:
            sides.append(i1)
            assignMode(i1, 0)
return sides def getSide0(currRow,
pixel, channel):

    global img, columns
    tRow = currRow
    if side0Mode == 0:
        tCol = currRow + pixel + 1
    else:
        tCol = columns - currRow - 2 - pixel
return img[tRow][tCol][channel] % 2 def
getSide1(currRow, pixel, channel):

    global img, columns, rows
    tCol = columns - currRow - 1
    if side1Mode == 0:
        tRow = currRow + 1 + pixel
    else:
        tRow = rows - currRow - 2 - pixel
return img[tRow][tCol][channel] % 2 def
getSide2(currRow, pixel, channel):

    global img, rows, columns
```

```
    tRow = rows - currRow - 1
    if side2Mode == 0:
        tCol = columns - currRow - 2 - pixel
    else:
        tCol = currRow + 1 + pixel
return img[tRow][tCol][channel] % 2 def
getSide3(currRow, pixel, channel):

    global img, rows
    tCol = currRow
    if side3Mode == 0:
        tRow = rows - currRow - 2 - pixel
    else:
        tRow = currRow + 1 + pixel
return img[tRow][tCol][channel] % 2 def
retrieveData(currRow, sides):

    global img, side0Mode, side1Mode, side2Mode, side3Mode,\
        retrievedTextRef, lengthRetrieved, retrievedTextLength, retrievedBits
    totalBits = (rows - (currRow + 1) * 2) * 3 * 4
    bitsThisIter = 0
    while True:
        for i1 in range(4):
            for j1 in range(4):
                temp1 = bitsThisIter // 4
                if sides[j1] == 0:
                    retrievedTextRef += str(getSide0(currRow, temp1 // 3, temp1 % 3))
                elif sides[j1] == 1:
                    retrievedTextRef += str(getSide1(currRow, temp1 // 3, temp1 % 3))
                elif sides[j1] == 2:
                    retrievedTextRef += str(getSide2(currRow, temp1 // 3, temp1 % 3))
                else:
```

```
                    retrievedTextRef += str(getSide3(currRow, temp1 // 3, temp1 % 3))
                retrievedBits += 1
                bitsThisIter += 1
                totalBits -= 1
                if retrievedBits == 16 and retrievedTextLength == 0:
                    retrievedTextLength = int(retrievedTextRef, 2)
                    retrievedTextRef = ""
                if retrievedTextLength != 0 and retrievedBits - 16 == retrievedTextLength:
                    return False
                if totalBits == 0:
return True def retrieve():
global img

    currRow = 0
    keepLooping = True
    while keepLooping:
        sides = getSideData(currRow)
        print("Sides retrieval order:", sides)
        print("Modes of side 0 to 3:", side0Mode, side1Mode, side2Mode, side3Mode)
        keepLooping = retrieveData(currRow, sides)
        currRow += 1
        print('Edge', currRow, 'retrieved')
    print("Retrieved bits:", retrievedBits, "bits") print("CSE3005 Digital Watermarking ang
Steganography Project") print("Image Steganography Using an Edge Based Embedding
Technique") print("Members:\n18BCI0268\n20BKT0056\n20MID0175 ") ipPath =
input("\nEnter the name of file containing input text: ") imPath =
input("Enter the name of file containing the image: ") img =
cv2.imread(imPath, 1) print("\nIn image:\nRows:", len(img), "\nColumns:",
len(img[0])) print("\nStep 1: Text compression") h = HuffmanCoding(ipPath)
output_path = h.compress() print("Compressed file path: " + output_path)
print("Compressed text in binary: " + compTextRef) print("Bits in compressed
text:", len(compTextRef), "bits") print("Text compression completed")
```

```
print("\nStep 2: Encrypting using AES") encrypted = "" plainValues = [] with

open(output_path, 'rb') as file:


    encryptionIp = ""

    byte = file.read(1)

    while len(byte) > 0:

        byte = ord(byte)

        bits = bin(byte)[2:].rjust(8, '0')

        encryptionIp += bits         byte = file.read(1)

file.close() len1 = len(encryptionIp) / 4 encryptionIpCopy =

encryptionIp encryptionIp = hex(int(encryptionIp,

2))[2:].zfill(int(len1)) for j in range(int(len(encryptionIp)

/ 32)):


    input_plain = encryptionIp[j:j+32]
    input_plain = input_plain.rstrip().replace(" ", "").lower()

    plain = split_string(8, input_plain)

    plain = [split_string(2, word) for word in plain]

    plainValues.append(plain)

    rounds = [apply_round_key(keys[0], plain)]

    for i in range(1, 10 + 1):

        rounds.append(aes_round(rounds[i - 1], keys[i]))

    paddingLen = len("{}".format(''.join(int_to_hex(flatten(rounds[10])))) * 4

    temp = bin(int("{}".format(''.join(int_to_hex(flatten(rounds[10])))), 16))[2:].zfill(paddingLen)

    encrypted += temp print("After encryption (in binary): " + encrypted) print("Encryption

completed") print("\nStep 3: Embedding data into image") rows = len(img) columns = len(img[0])

compTextRef = encrypted compTextLen = bin(len(compTextRef)) compTextLen = compTextLen[2:]

while len(compTextLen) < 16:     compTextLen = '0' + compTextLen compTextRef = compTextLen +

compTextRef print("Size of data to be embedded after padding 16 bit length:",

len(compTextRef), "bits") embed() cv2.imwrite('encrypted.PNG', img) cv2.imshow('encrypted

image', img) print("Data embedded successfully") print("\nStep 4: Retrieving data from image")

img = cv2.imread('encrypted.PNG', 1) rows = len(img) columns = len(img[0]) retrieve()

print("Bits in retrieved text after removing 16 bit padding:", len(retrievedTextRef), "bits")
```

```
print("Retrieved text: " + retrievedTextRef) print("Successfully retrieved") print("\nStep 5:
Decrypting retrieved data") decrypted = "" len1 = len(retrievedTextRef) / 4 retrievedTextRef =
hex(int(retrievedTextRef, 2))[2:].zfill(int(len1)) for j in range(int(len(retrievedTextRef) /
32)):

    plain = plainValues[j]
    for i in range(10, 0, -1):
        rounds.append(inverse_aes_round(rounds[i - 1], keys[i]))
    paddingLen = len("{}".format(''.join(flatten(plainValues[j])))) * 4
    temp = bin(int("{}".format(''.join(flatten(plainValues[j]))), 16))[2:].zfill(paddingLen)
    decrypted += temp print("Number of bits in decrypted data:",
len(decrypted), "bits") print("Decrypted data: " + decrypted)
print("Decryption completed") print("\nStep 6: Decompression of
data") print("Decompressed file path: " +
h.decompress(output_path)) print("Decompression completed")
cv2.waitKey(0)
```

6. Output and Performance Analysis

6.1. Execution snapshots

Input Image:

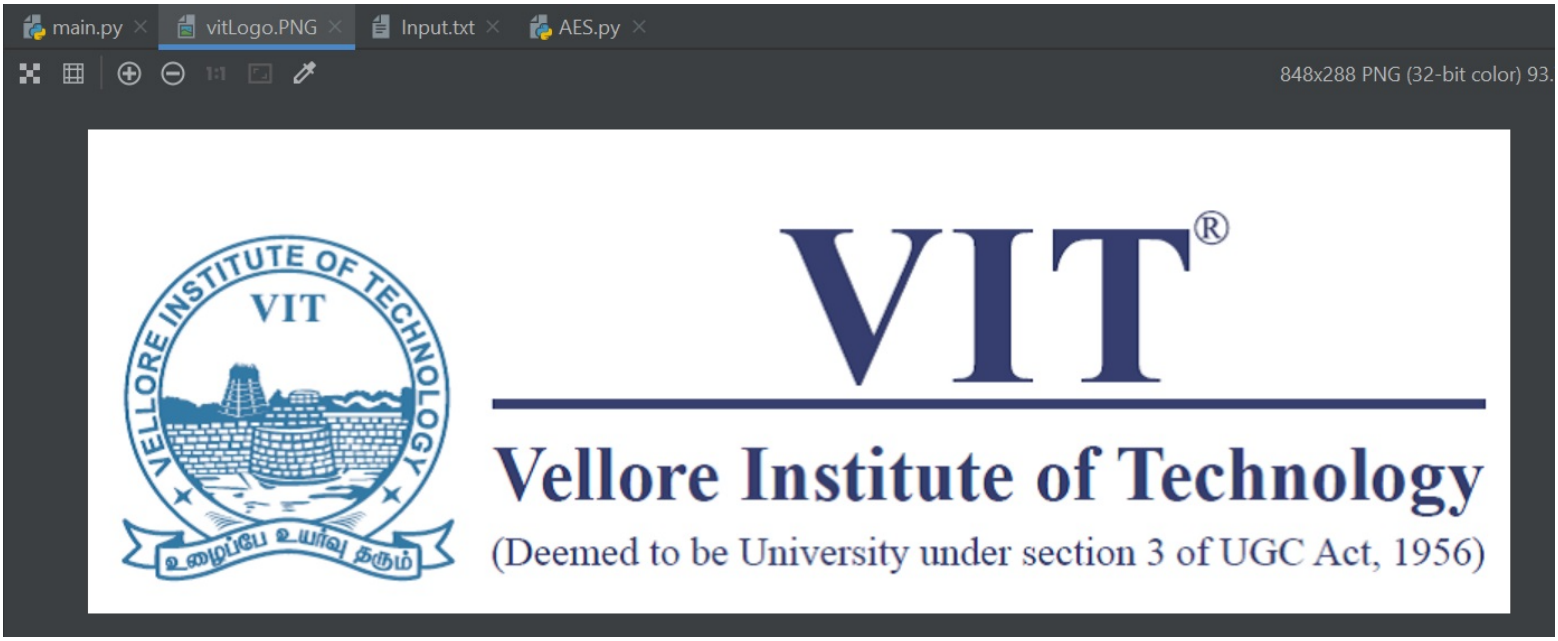
Execution:

```
C:\Users\NiTiSSH\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:/Users/NiTiSSH/PycharmProjects/pythonProject1/main.py
BCI3005 Digital Watermarking and Steganography Project
Title: Image Steganography Using a LSB Based Embedding Technique
Members:
18BCI0268
20MID0175
20BKT0056

Enter the name of file containing input text: Input.txt
Enter the name of file containing the image: vitLogo.PNG

In image:
Rows: 288
Columns: 848

Step 1: Text compression
Bits in input text:  6456 bits
Compressed file path: Input.bin
Compressed text in binary: 00111000100101110011100011100110100011100110000100000001111101110111010111011101110111001010001111110100101110100101
Bits in compressed text: 3712 bits
Text compression completed
```

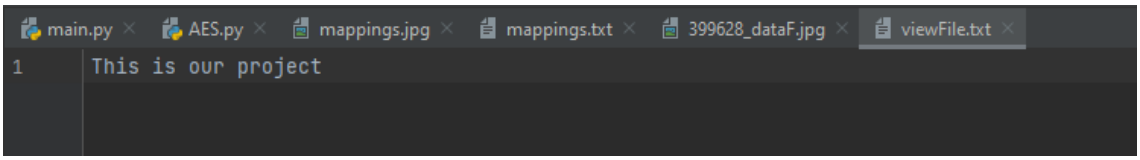
```
Step 2: Encrypting using AES
After encryption (in binary): 0101110010100011110000000111111100101101011111011110011000101100011000010000101110001000110100010011101010101101001
Encryption completed

Step 3: Embedding data into image
Size of data to be embedded after padding 16 bit length: 3728 bits
Sides embedding order: [2, 0, 1, 3]
Modes of side 0 to 3: 1 1 1 0
Edge 1 filled
Sides embedding order: [1, 2, 0, 3]
Modes of side 0 to 3: 1 1 1 0
Edge 2 filled
Bits embedded: 3728 bits
Data embedded successfully

Step 4: Retrieving data from image
Sides retrieval order: [2, 0, 1, 3]
Modes of side 0 to 3: 1 1 1 0
Edge 1 retrieved
Sides retrieval order: [1, 2, 0, 3]
Modes of side 0 to 3: 1 1 1 0
```

```
Edge 2 retrieved
Retrieved bits: 3728 bits
Bits in retrieved text after removing 16 bit padding: 3712 bits
Retrieved text: 0101110010100011110000000111111100101101011111011110011000101100011000010000101110001000110100010011101010101101001011101000010110
Successfully retrieved

Step 5: Decrypting retrieved data
Number of bits in decrypted data: 3712 bits
Decrypted data: 0011100010010111001110001110011010001110011000010000000111110111011101011101111011110111001010001111110100101110100101110111011110
Decryption completed

Step 6: Decompression of data
Decompressed file path: Input_decompressed.txt
Decompression completed
```

Encrypted image:



Decompressed final file:



6.2. Output – in terms of performance metrics

Hiding Capacity:

For an image with r rows and c columns: top = ceil(r/2) - 1

Number of bits that can be embedded = 6*top*(r + c) – 12*(top)*(top + 1)

Ideally a 1920*1080 image or higher should be used that would allow 758 Kilo Bytes at 1920*1080 resolution. Another thing to be noted is that this size is of the compressed text that is allowed, so the original text size is approximately 1.8 times the given size.

Distortion Measurement:

This would vary based on the bit that is being embedded. Worst case scenario is that all bits embedded ended up changing the last bit. For n bits embedded in an image with r rows and c columns:

Worst case scenario distortion = 100*n / (24*r*c) %

6.3. Performance comparison with existing works

Attempts were done to reduce the distortion as well as increasing the hiding capacity by using compression. Using a combination of traditional encryption also lead to a very secure result with very less distortion. This is because:

1. With AES encryption, the original input is very highly distorted.

2. However, when we embed this data into an image all that distortion created is greatly reduced as it is only a portion of the image itself.

3. Furthermore, if the bit that is embedded in LSB of a channel ended up being the same as the one in the image, this also reduces the distortion.

7. Conclusion

A system that combines username pseudonymization, user data anonymization, compression, traditional encryption and image steganography was constructed. Efforts were made to maximize hiding capacity, reduce distortion but simultaneously increase security and efficiency of the system. Further improvements can be made into the compression techniques used to reduce the distortion and increase capacity. The mapping technique for the pseudonyms can also be further improved. Efficiency of the embedding algorithm could also be improved to require fewer iterations with a lower time complexity. With a proper direction, cryptographic and steganographic techniques can be combined with anonymization techniques to create systems that provide high privacy, security and hide it efficiently in multimedia files.

8. References

[1] Arshiya Sajid Ansari, Mohammed Sajid Mohammadi, Mohammed Tanvir Parvez, 'A comparative study of recent steganography techniques for multiple image formats', International Journal of Computer Network and Information Security, 2019.

[2] Hesham F. A. Hamed, Aziza I. Hussein, Ashraf A. M. Khalaf, Osama, Fouad Abdel Wahab, 'Hiding Data Using Efficient Combination of RSA Cryptography and Compression Steganography Techniques', IEEE Access Vol 9, 2021.

[3] Jagan Raj, Jayapandiyan, C. Kavitha, K. Sakthivel, 'Enhanced Least Significant Bit Replacement Algorithm in Spatial Domain of Steganography Using Character Sequence Optimization', IEEE Access Vol 8, 2020.

[4] Abdel Wahab, Aziza I. Hussain, Hesham F.A. Hamed, Hamdy M. Kelash, Ashraf A.M. Khalaf, Hanafy M. Ali, 'Hiding Data in Images Using Steganography techniques with compression algorithms', Telkomnika Vol 17, 2019.

[5] Text book: Cryptography and Network Security Sixth Edition by William Stallings.