



SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Fall Semester 2021-22

Course Code : CSE3501

Programme : B.Tech

Course Name : Information Security Analysis and Audit

J – Component Report

Team Details:

Name	Reg. Number	Slot
Shashwat Sinha	19BCE0684	F1 L55+L56
Utkarsh Deep	19BCE2696	F2 L9+L10

Image Steganography using an Edge Based Embedding Technique

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.

Problem Statement and Objectives:

Internet usage has grown exponentially in the recent years, but the data that is being shared over the internet has also developed a high susceptibility to attacks that tend to steal your information and circulate it on the open forum without any concern. Encryption has played a crucial role in controlling the damage done by these attacks or any kind of data breach. In this paper we will implement one such method to conceal any information on an image such that the original image has almost no change in itself after the encoding has been done. This work would be extremely reliable in providing a second hand security in case of any data breach or malware attack.

The aim of this project is to execute a novel information hiding method based on Edge Embedding Technique and LSB (Least Significant Bit) Technique on digital images. Moreover, traditional AES Encryption will also be performed to provide an extra level of security to the information before embedding it to the digital image making it very tough for attackers to decode the actual piece of information from just a simple looking image.

Literature Review:

<p>Title – Hiding Data Using Efficient Combination of RSA Cryptography, and Compression Steganography Techniques</p> <p>Author – Hesham F. A. Hamed, Aziza I. Hussein, Ashraf A. M. Khalaf, Osama Fouad Abdel Wahab</p> <p>Journal – IEEE Access (Volume: 9)</p> <p>Year - 2021</p>	<p>Work – 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 process, and also getting or extracting the message process.</p> <p>It includes –</p> <ul style="list-style-type: none">• Embedding Algorithm• Extracting Algorithm	<p>Observation – In this paper, a combination of RSA, Huffman coding, and DWT 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 the more effective visual quality and storage capacity, and it has high security and acceptable durability against attacks than the existing techniques.</p>
---	--	---

<p>Title – Enhanced Least Significant Bit Replacement Algorithm in Spatial Domain of Steganography Using Character Sequence Optimization.</p> <p>Author – Jagan Raj Jayapandiyan, C. Kavitha, K. Sakthivel</p> <p>Journal – IEEE Access (Volume 8)</p> <p>Year - 2020</p>	<p>Work - 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 colour 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.</p>	<p>Observation – 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 secret message embedding process. Proposed algorithm also enhances the secret message as they can't be interpreted for the character sequences, which went through the optimization.</p>
<p>Title – Image steganography method with Spread Spectrum Technique</p> <p>Author – Manjot Kumar Bhatia</p> <p>Journal – Proceedings of Fourth International Conference on Soft Computing for Problem Solving</p> <p>Year – 2015</p>	<p>Work – We have proposed a spread spectrum image stenographic method that is the combination of encryption, properties of orthogonal image planes, spread spectrum, modulation and image encryption techniques. Embedding process uses the orthogonal planes of the cover image to modulate and hide the important message. Secret message can be extracted from the stego-image without the cover image.</p>	<p>Observation – The proposed image steganography method is secure and robust against various attacks. The message embedded in the cover image is encrypted with RC4 using first secret Key. The stego image is also encrypted using secret key, known to sender and receiver only. Receiver decrypts the stego image using second secret key. It is not possible to know the secret message because steganalyst is not aware of the orthogonality of both stego image and its red image plane.</p>

<p>Title – Hiding Data in Images Using Steganography techniques with compression algorithms</p> <p>Author – Abdel Wahab, Aziza I. Hussain, Hesham F.A. Hamed, Hamdy M. Kelash, Ashraf A.M. Khalaf, Hanafy M. Ali</p> <p>Journal – TELKOMNIKA, Vol.17</p> <p>Year - 2019</p>	<p>Work – We 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.</p>	<p>Observation – 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 we can hide the intended data in messages while minimizing its size.</p>
<p>Title – A comparative study of recent steganography techniques for multiple image formats</p> <p>Author – Arshiya Sajid Ansari, Mohammed Sajid Mohammadi, Mohammed Tanvir Parvez</p> <p>Journal – International Journal of Computer Network and Information Security</p> <p>Year - 2019</p>	<p>Work – We review a number of reported works on image Steganography. We categorize the methods based on cover image formats like JPEG, RGB and PNG and their domain information. Classification of Image Steganography:</p> <ul style="list-style-type: none"> • RGB/Bitmap (lossless) • JPEG (lossy) • PNG (lossless) • TIFF (lossy/lossless) 	<p>Observation – 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.</p>

Proposed Methodology:

Following are the steps proposed for Image Steganography:

i. Text Compression –

- Huffman coding will be used for the purpose of minimization of tree nodes and leading to compression of text.
- 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.

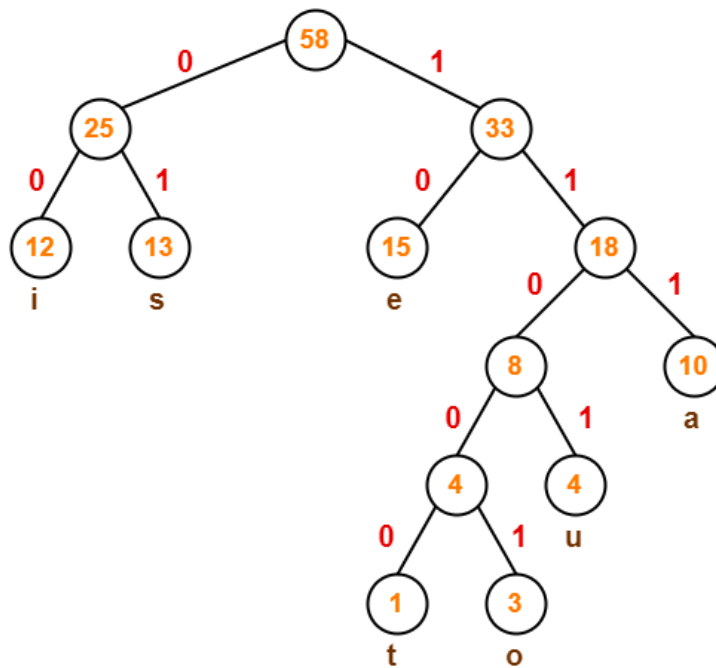


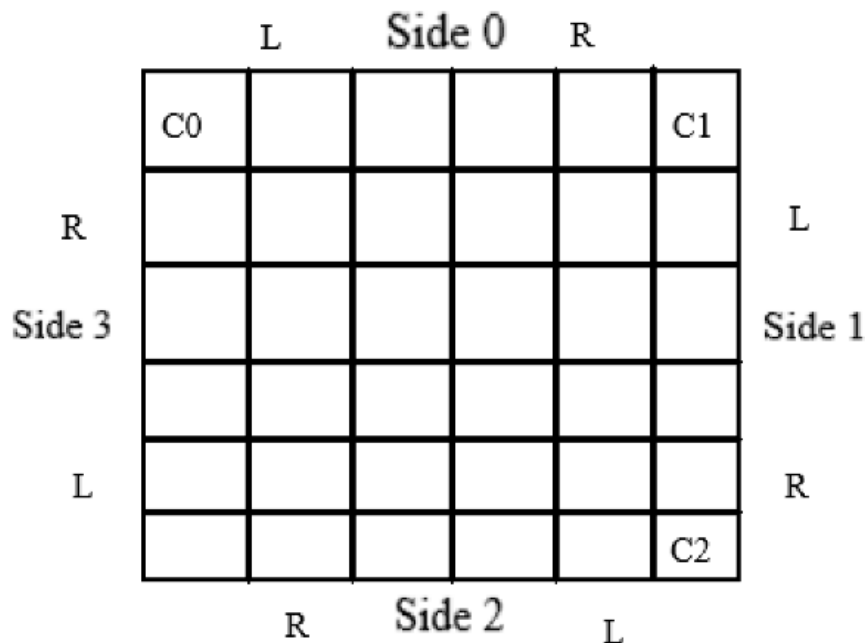
Fig: In the above example 'i' will be '00' and 'o' will be '11001'

ii. AES Encryption –

- The compressed text is padded so that it is in form of $120 \bmod 128$.
- 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 much padding is done. 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.

iii. Text Embedding in Image –

- 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 represented in 2-bit binary as 00, 01, 10 and 11 and also three corners starting from top left corner are marked as 0, 1 and 2.



- Three unique integers are randomized between 0 and 3 to get 3 sides.
- Modes for these three 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.

- 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.

iv. Text Retrieving from Image –

- While embedding, we had stored the side order and modes in 3 corners. First, that information is 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.

v. 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.

vi. 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 and text is decompressed.
- 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.

Code:

AES.py

```
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(flatten(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:
                    currRow)
                    side0(temp1 // 3, temp1 % 3, compTextRef[embeddedBits],
currRow)
                    elif sides[j1] == 1:
                        currRow)
                        side1(temp1 // 3, temp1 % 3, compTextRef[embeddedBits],
currRow)
                        elif sides[j1] == 2:
                            currRow)
                            side2(temp1 // 3, temp1 % 3, compTextRef[embeddedBits],
currRow)
                            else:
                                currRow)
                                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("CSE3501 | Information Security Analysis and Audit | Project")
print("Title: Image Steganography Using an Edge Based Embedding Technique")
print("Team Members:\n19BCE0684\n19BCE2696")
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)

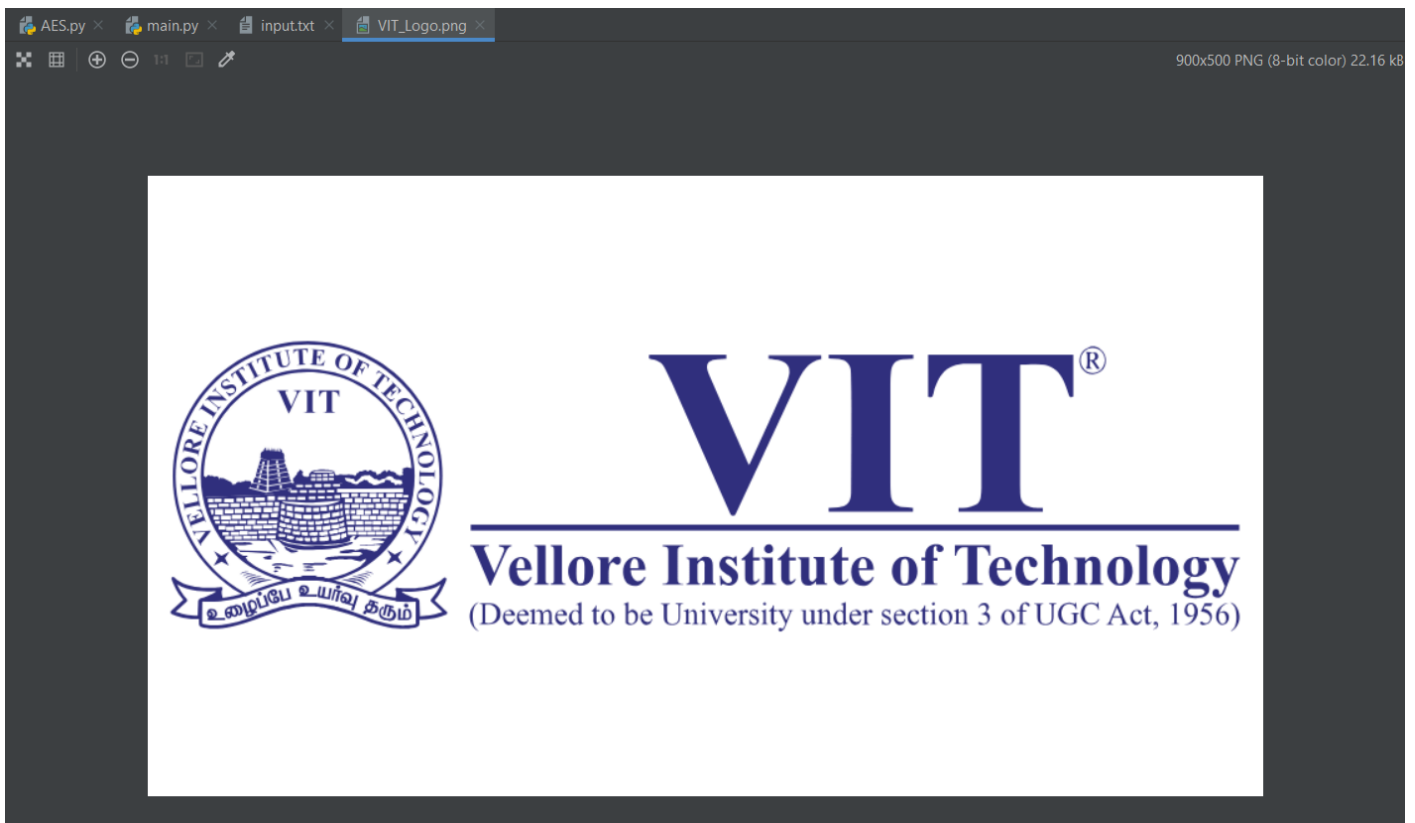
```

1. Results:

Input Text –

```
AES.py × main.py × input.txt ×  
1 This is input text for our CSE3501: Info Security Analysis and Audit Project.  
2 Developed and analysed by Shashwat Sinha - 19BCE0684 and Utkarsh Deep - 19BCE2696.  
3
```

Input Image –



Executing main.py –

```
Run: main ×  
CSE3501 | Information Security Analysis and Audit | Project  
Title: Image Steganography Using an Edge Based Embedding Technique  
Team Members:  
19BCE0684  
19BCE2696  
  
Enter the name of file containing input text: input.txt  
Enter the name of file containing the image: VIT_Logo.png
```

```
In image:
Rows: 500
Columns: 900

Step 1: Text compression
Bits in input text: 1280 bits
Compressed file path: input.bin
Compressed text in binary: 01011110011101010100111011111111011101111111011101010110110000011000
Bits in compressed text: 896 bits
Text compression completed

Step 2: Encrypting using AES
After encryption (in binary): 110101011111100011110101010011010001101101010101000001110100011000
Encryption completed


Step 3: Embedding data into image
Size of data to be embedded after padding 16 bit length: 912 bits
Sides embedding order: [0, 2, 3, 1]
Modes of side 0 to 3: 0 0 1 1
Edge 1 filled
Bits embedded: 912 bits
Data embedded successfully

Step 4: Retrieving data from image
Sides retrieval order: [0, 2, 3, 1]
Modes of side 0 to 3: 0 0 1 1
Edge 1 retrieved
Retrieved bits: 912 bits
Bits in retrieved text after removing 16 bit padding: 896 bits
Retrieved text: 11010101111110001111010101001101000110110101010100000111010001100000011110010101
Successfully retrieved

Step 5: Decrypting retrieved data
Number of bits in decrypted data: 896 bits
Decrypted data: 0101111001110101010011101111111101110111111101110101011011000001100001100000011
Decryption completed

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

Compressed binary file (only to be read by system) –

 input.bin - Notepad

```
File Edit Format View Help
^uN0w00`00
S0000000!3,00 00k0z000000nMİj0Z0X0_j0P~001000^00_00
0hZ '0oL0b0000000|
```

Stego Image –

encrypted image

– □ ×



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Decompressed File –

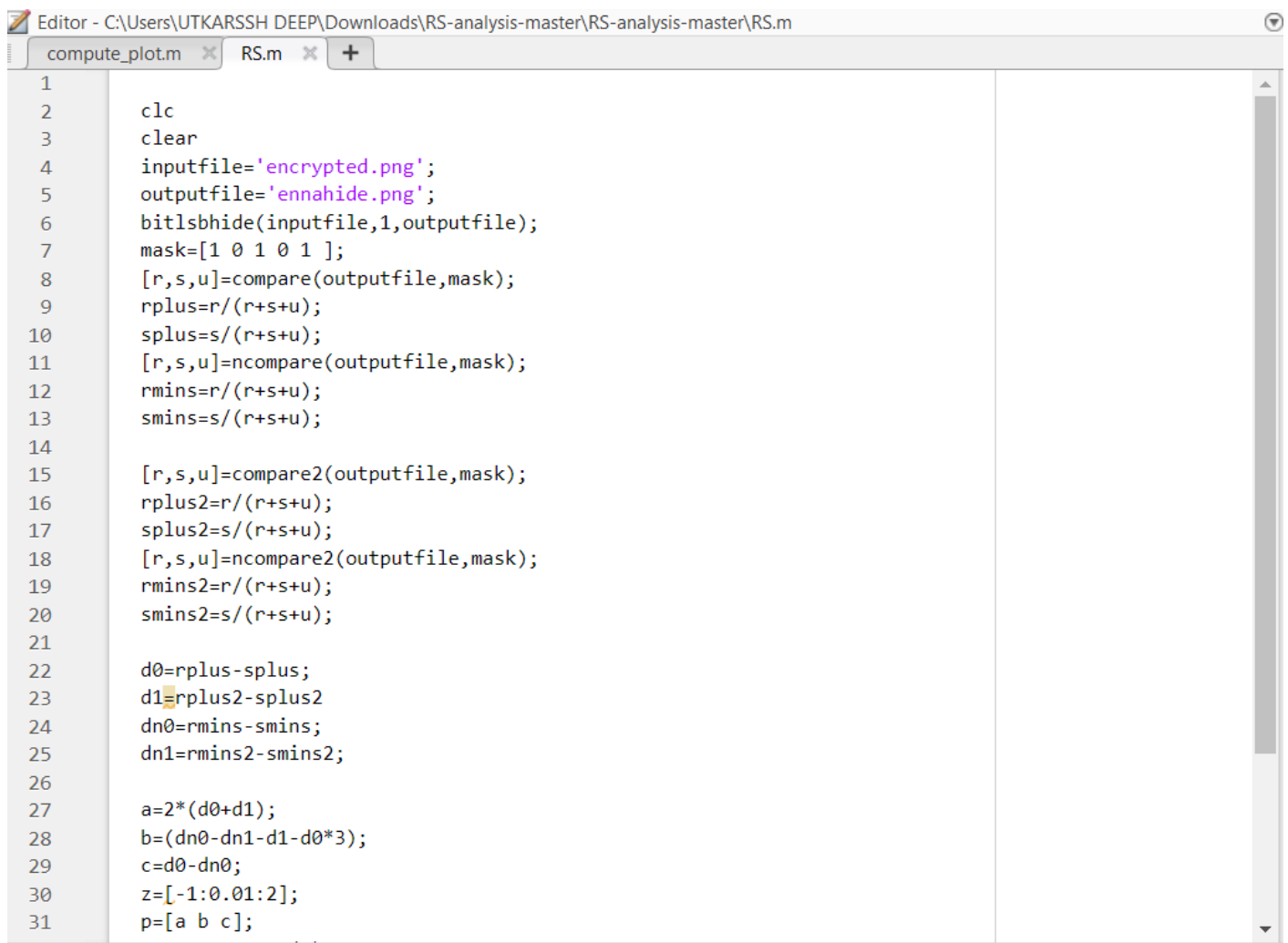
AES.py × main.py × input_decompressed.txt ×

```
1 This is input text for our CSE3501: Info Security Analysis and Audit Project.  
2 Developed and analysed by Shashwat Sinha - 19BCE0684 and Utkarsh Deep - 19BCE2696.
```


COMPARITIVE ANALYSIS OF EDGE BASED TECHNIQUE WITH LSB-DCT USING STEGANALYSIS

Steganalysis is the art and science to detect whether a given digital medium contains hidden data. The art of steganalysis plays a greater role in the selection of features or properties of digital data to test for hidden data while the science helps in designing of the technique to detect or extract tests the hidden data. A steganalysis method is considered as successful if it can detect and extract the hidden data embedded. The objective of steganalysis is to detect messages hidden in cover objects, such as digital images. Steganalysis can be termed as a method of attacking the digital media for estimating whether the media contains secret data embedded in it. Steganalysis can serve as an effective way to judge the security performance of steganographic techniques. A good steganographic technique should be imperceptible not only to human eye, but also to computer analysis. We used the **stego-only attack** for our analysis in which only the steganography medium/object is available for analysis.

We performed RS Analysis of images encrypted by our edge based technique and LSB technique to find vulnerabilities to find the optimum algorithm for performing steganography. It is a method of collecting decryption of message hidden in images and analysing the data. We have Implemented it in MATLAB and plotted graphs to compare the algorithms.



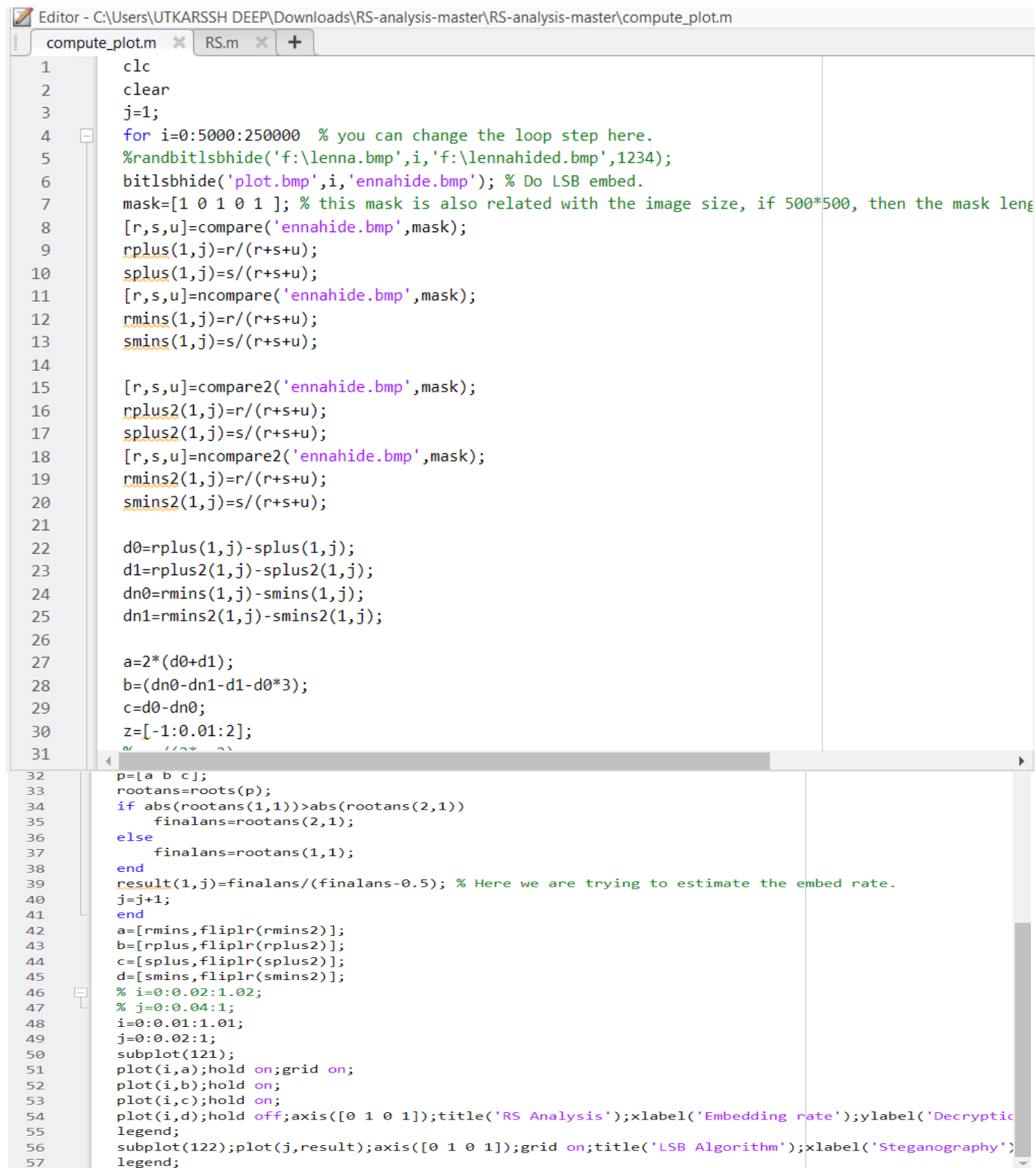
```
Editor - C:\Users\UTKARSSH DEEP\Downloads\RS-analysis-master\RS-analysis-master\RS.m
compute_plot.m  RS.m  +
1
2   clc
3   clear
4   inputfile='encrypted.png';
5   outputfile='ennahide.png';
6   bitlsbhide(inputfile,1,outputfile);
7   mask=[1 0 1 0 1 ];
8   [r,s,u]=compare(outputfile,mask);
9   rplus=r/(r+s+u);
10  splus=s/(r+s+u);
11  [r,s,u]=ncompare(outputfile,mask);
12  rmins=r/(r+s+u);
13  smins=s/(r+s+u);
14
15  [r,s,u]=compare2(outputfile,mask);
16  rplus2=r/(r+s+u);
17  splus2=s/(r+s+u);
18  [r,s,u]=ncompare2(outputfile,mask);
19  rmins2=r/(r+s+u);
20  smins2=s/(r+s+u);
21
22  d0=rplus-splus;
23  d1=rplus2-splus2;
24  dn0=rmins-smins;
25  dn1=rmins2-smins2;
26
27  a=2*(d0+d1);
28  b=(dn0-dn1-d1-d0*3);
29  c=d0-dn0;
30  z=[-1:0.01:2];
31  p=[a b c];
```

```

32 rootans=roots(p);
33 if abs(rootans(1,1))>abs(rootans(2,1))
34     finalans=rootans(2,1);
35 else
36     finalans=rootans(1,1);
37 end
38 p=finalans/(finalans-0.5);
39

```

Analysis and testing:

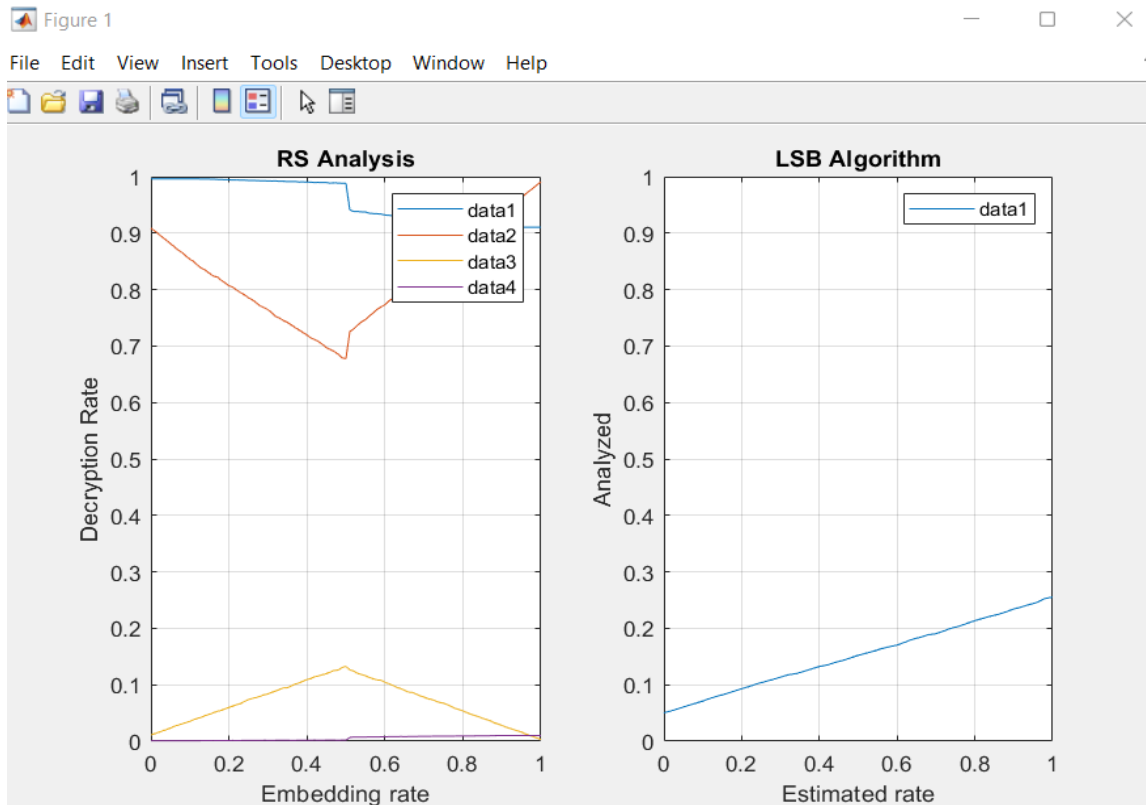


```

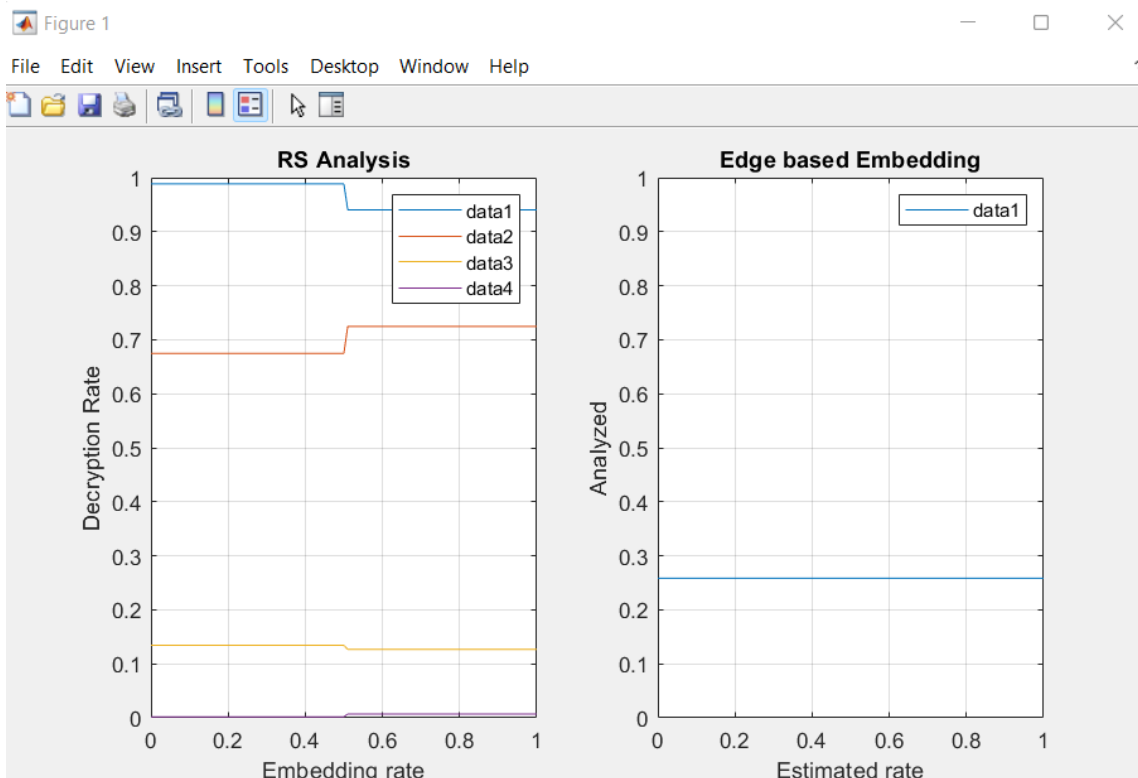
Editor - C:\Users\UTKARSSH DEEP\Downloads\RS-analysis-master\RS-analysis-master\compute_plot.m
compute_plot.m x RS.m x +
1  clc
2  clear
3  j=1;
4  for i=0:5000:250000 % you can change the loop step here.
5  %randbitlsbhide('f:\lenna.bmp',i,'f:\lennahided.bmp',1234);
6  bitlsbhide('plot.bmp',i,'ennahide.bmp'); % Do LSB embed.
7  mask=[1 0 1 0 1]; % this mask is also related with the image size, if 500*500, then the mask leng
8  [r,s,u]=compare('ennahide.bmp',mask);
9  rplus(1,j)=r/(r+s+u);
10 splus(1,j)=s/(r+s+u);
11 [r,s,u]=ncompare('ennahide.bmp',mask);
12 rmins(1,j)=r/(r+s+u);
13 smins(1,j)=s/(r+s+u);
14
15 [r,s,u]=compare2('ennahide.bmp',mask);
16 rplus2(1,j)=r/(r+s+u);
17 splus2(1,j)=s/(r+s+u);
18 [r,s,u]=ncompare2('ennahide.bmp',mask);
19 rmins2(1,j)=r/(r+s+u);
20 smins2(1,j)=s/(r+s+u);
21
22 d0=rplus(1,j)-splus(1,j);
23 d1=rplus2(1,j)-splus2(1,j);
24 dn0=rmins(1,j)-smins(1,j);
25 dn1=rmins2(1,j)-smins2(1,j);
26
27 a=2*(d0+d1);
28 b=(dn0-dn1-d1-d0*3);
29 c=d0-dn0;
30 z=[-1:0.01:2];
31 % (a% 2)
32 p=[a b c];
33 rootans=roots(p);
34 if abs(rootans(1,1))>abs(rootans(2,1))
35     finalans=rootans(2,1);
36 else
37     finalans=rootans(1,1);
38 end
39 result(1,j)=finalans/(finalans-0.5); % Here we are trying to estimate the embed rate.
40 j=j+1;
41 end
42 a=[rmins,fliplr(rmins2)];
43 b=[rplus,fliplr(rplus2)];
44 c=[splus,fliplr(splus2)];
45 d=[smins,fliplr(smins2)];
46 % i=0:0.02:1.02;
47 % j=0:0.04:1;
48 i=0:0.01:1.01;
49 j=0:0.02:1;
50 subplot(121);
51 plot(i,a);hold on;grid on;
52 plot(i,b);hold on;
53 plot(i,c);hold on;
54 plot(i,d);hold off,axis([0 1 0 1]);title('RS Analysis');xlabel('Embedding rate');ylabel('Decryptic
55 legend;
56 subplot(122);plot(j,result);axis([0 1 0 1]);grid on;title('LSB Algorithm');xlabel('Steganography')
57 legend;

```

Plotting the analysis for better picture:



The above graphs depict the RS analysis of LSB algorithm as we can see that there are clear vertices in the first graph which shows that they are the parts of the embedded image file that can be retrieved easily leading to further easy data access. In simple words they are the vulnerable parts which once exposed can lead to the easy access of whole data. This part can be proved by the second graph which shows that once the weak spot is exposed (at the point ~ 0.05), the estimated access rate steadily increases showing the impact of presence of a vulnerable point(s).



In this Edge based embedding technique we can see that there is no sudden peak but a gradual increase and then constant embedding rate hence exploiting the vulnerable part is very hard as it is negligible. Also, in the second graph we can see that unlike LSB, here is no gradual increase in fact, the analysis show that estimated rate is constant depicting that vulnerable part could not be found to exploit.

This information gained from the analysis clearly shows that Edge based Embedding technique has an upper hand on LHB algorithm in terms of security which makes it a more reliable method of steganography.

Now we will test the efficiency of both Algorithms to find the final more optimum algorithm.:

PSNR Test:

The peak signal-to-noise ratio (PSNR) is the most common metric used to evaluate the stego image quality.

```

Editor - C:\Users\UTKARSSH DEEP\Documents\MATLAB\getPSNR.m
getPSNR.m  getMSSIM.m  main.m  +
1  function psnr=getPSNR(frameReference,frameUnderTest)
2  s1=double(frameReference-frameUnderTest).^2;
3
4      s = sum(sum(s1));
5      sse = s(:, :, 1)+s(:, :, 2)+s(:, :, 3);
6      if( sse <= 1e-10)
7          psnr=0;
8      else
9          mse = sse / double(size(frameReference,1)*size(frameReference,2)*size(frameReference,3));
10         psnr = 10.0 * log10((255 * 255) / mse);
11     end
12 end

```

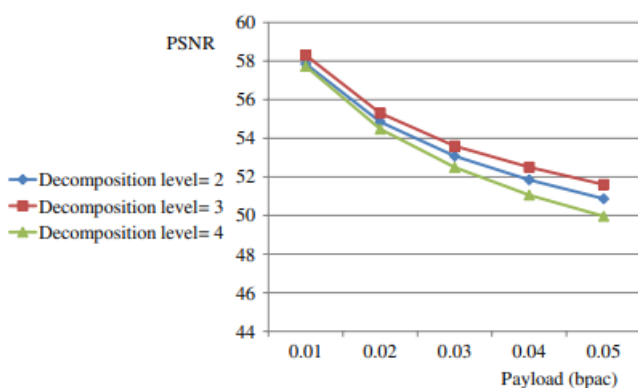


Fig 3

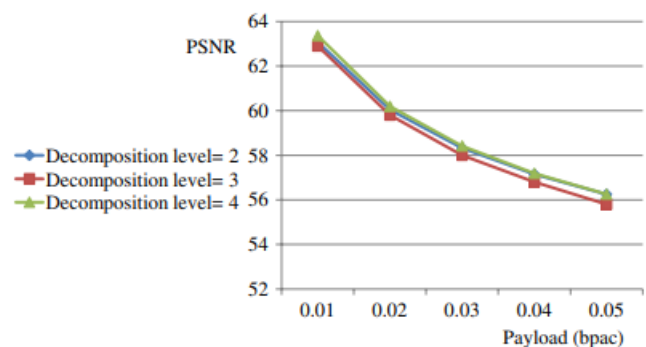


Fig 4

In Figs. 3 and 4, the PSNR values of the LSB Algorithm and Edge based embedding for the decomposition levels 2, 3, and 4 are compared. As shown in Fig. 3, the best decomposition level in LSB for different payloads is 3. It shows that while the payload is increased, the differences between PSNR values are also increased. In Fig. 4, the best decomposition level is 4 while two other decomposition levels have almost equal PSNRs.

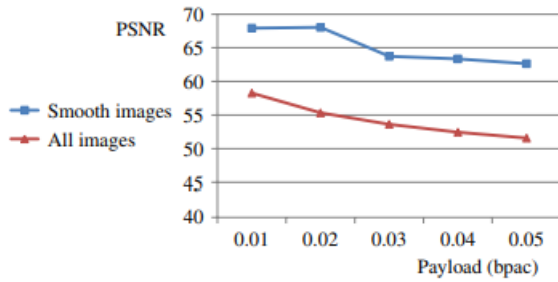


Fig 5

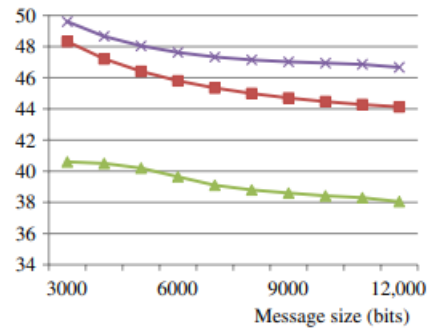


Fig 6

Fig 5 depicts The PSNR test results of Edge Based Embedding for smooth images and all images and Fig 6 depicts PSNR Test result of Edge based embedding (red) with LSB (green) and the benchmark (purple).

Results:

RS Analysis:

- RS analysis proves that Edge based embedding technique has far less vulnerabilities to the stego-only attack as compared to LSB Algorithm.
- LSB embedded images had higher rate of exploitation after finding vulnerable part whereas Edge based embedded had a constant curve.

PSNR Test

- PSNR test shows that the decomposition rate in LSB is higher after level 3 whereas in edge based embedding in gradually after level 4 only.

Conclusion:

- Edge based embedding is found to be more secure and reliable with less vulnerabilities.
- Change in decomposition rate in not so great but edge based computing has slight edge as it starts later but end time is same.
- Hence Finally Edge based computing is found to be more reliable after analysing both the test results over different images and parameters.

SECURITY ANALYSIS REPORT

Vulnerability Assessment and Functionality Test

CONTENTS

SL.NO	CONTENT	PG. NO
1.	EXECUTIVE SUMMARY	30
2.	SCOPE AND OBJECTIVES	30
3.	BACKGROUND	30
4.	FUNCTIONALITY TESTING	31
5.	VULNERABILITY ATTACKS	32
4.	TESTING ANALYSIS	33
5.	RECOMMENDATIONS	34

Executive Summary

This report covers the security issues with steganography techniques focussing on edge based embedding and it's comparison with conventional LSB technique and how they perform on the different parameters of our testing and analysing them on scales like security, vulnerabilities & integrity of the algorithm.

Objectives and Scope:

The objective of performing this vulnerability assessment is to create an overview of the security risks to the proposed system of image steganography using edge based embedding technique. This assessment will serve as a guideline to overcome the threats and provide a more reliable system. The scope of this audit report will also extend to the functionality testing of the system which will ensure that all the modules are stable and working as proposed earlier.

Background:

The system constitutes mainly of three modules on the encryptor end and inverse functionalities on the retriever end. At the encryptor end, first the input data is compressed using Huffman Encoding then it is encrypted using general AES Algorithm and finally it is

embedded into the selected image. Similarly, in the retriever end, first the embedded text is retrieved then inverse AES and Huffman Algorithm is used to convert the encrypted text into normal readable format.

The proposed system is implemented using OpenCV and Python making it platform independent so that it can be configured on any end device. This ensures high acceptability rate among the users but also makes it vulnerable to a bulk of attacks which try to steal the data mid-way during decryption or encryption and also during the transfer of the stego-image via any network. Security weakness in the model would expose the hidden data embedded on the image resulting in complete failure of the system. Therefore, it becomes necessary to carry out a detailed vulnerability and functionality assessment.

Functionality Testing (Check list approach):

S. No	Functionality	Present/Absent
Encryptor End		
1.	User able to enter a custom input text.	PRESENT
2.	User able to select custom image to continue.	PRESENT
3.	Huffman Encoding of the input text.	PRESENT
4.	AES Encryption on encoded bits.	PRESENT
	4.1. User able to enter custom secret key.	ABSENT
	4.2. 10 round 128 – bit encryption.	PRESENT
5.	Encrypted text embedding on edges of image.	PRESENT
	5.1. Leaving the corners out.	PRESENT
6.	Generating stego-image	PRESENT
Retriever End		
7.	Retrieve encrypted text from image.	PRESENT
8.	Decrypt retrieved bits using inverse AES Algorithm.	PRESENT
9.	Decode decrypted text using inverse Huffman Algorithm.	PRESENT
10.	Store the original text in the machine.	PRESENT

From the above functionality check, it is clear that all the proposed functionalities have been implemented. But there is still room for improvement as it can be seen that user doesn't get a choice to enter a custom secret key during the encryption of the encoded text. But after the analysis it is has been proven a boon for the system security and usability. As, the end user would not have to worry about entering the secret key for retrieving the actual text data. We can compare this to modern day chatting applications such as WhatsApp, which provides an end-to-end encryption but in the similar manner the user is not given a choice to enter their own secret key. Thus, it can be concluded that all the features are efficiently running on the system thereby closing gap for any structural vulnerabilities.

Vulnerability Analysis:

1. Brute Force Attacks –

- A brute force attack uses trial-and-error to guess login info, encryption keys, or find a hidden web page. Hackers work through all possible combinations hoping to guess correctly.
- In the proposed system, AES – 128 bit encryption module can be subjected to this type of attack which could lead directly to exposure of the secret key.

2. Biclique Attacks –

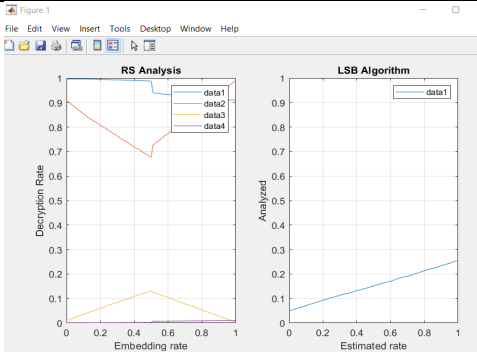
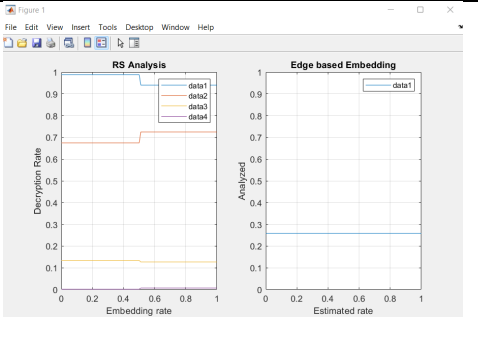
- A biclique attack is a variant of the meet-in-the-middle (MITM) method of cryptanalysis. It utilizes a biclique structure to extend the number of possibly attacked rounds by the MITM attack. Since biclique cryptanalysis is based on MITM attacks, it is applicable to both block ciphers and (iterated) hash-functions. Biclique attacks are known for having broken both full AES and full IDEA.
- For AES – 128, the key can be recovered with a computational complexity of $2^{126.1}$ using the biclique attack. For biclique attacks on AES – 192 and AES – 256, the computational complexities of $2^{189.7}$ and $2^{254.4}$ respectively apply.

TESTING ANALYSIS

We used the **stego-only attack** for our analysis in which only the steganography medium/object is available for analysis.

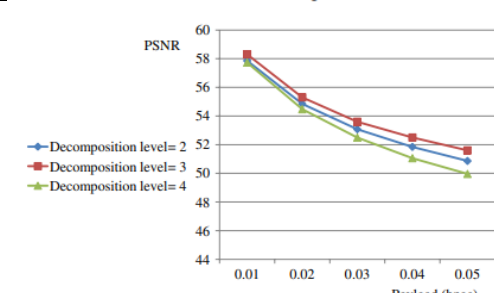
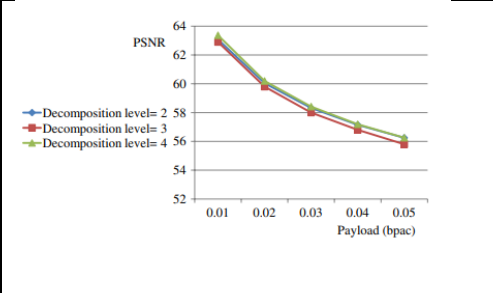
We performed RS Analysis of images encrypted by our edge based technique and LSB technique to find vulnerabilities to find the optimum algorithm for performing steganography. It is a method of collecting decryption of message hidden in images and analysing the data. We have Implemented it in MATLAB and plotted graphs to compare the algorithms

I. RS ANALYSIS

PROPERTY	LSB ALGORITHM	EDGE BASED EMBEDDING
Embedding Time	Relatively High	Low
Vulnerability	Exploited Easily	Not found by this method
Decryption	Accelerates rapidly after exploiting vulnerability	Constant decryption rate throughout
Security	Very low	High because of 2 level encryption
		

II. PSNR TEST

The peak signal-to-noise ratio (PSNR) is the most common metric used to evaluate the stego image quality

TEST RESULT	LSB ALGORITHM	EDGE BASED EMBEDDING																																																
Decomposition	Encrypted image starts decomposing after level 2 decomposing after level 2	Encrypted image starts decomposing after level 4																																																
Final image result	Similar level of decomposition	Similar level of decomposition																																																
	 <table><caption>Approximate PSNR values for LSB algorithm</caption><tr><th>Payload (bpac)</th><th>Level 2</th><th>Level 3</th><th>Level 4</th></tr><tr><td>0.01</td><td>58.5</td><td>58.0</td><td>57.5</td></tr><tr><td>0.02</td><td>56.5</td><td>56.0</td><td>55.5</td></tr><tr><td>0.03</td><td>54.5</td><td>54.0</td><td>53.5</td></tr><tr><td>0.04</td><td>52.5</td><td>52.0</td><td>51.5</td></tr><tr><td>0.05</td><td>50.5</td><td>50.0</td><td>49.5</td></tr></table>	Payload (bpac)	Level 2	Level 3	Level 4	0.01	58.5	58.0	57.5	0.02	56.5	56.0	55.5	0.03	54.5	54.0	53.5	0.04	52.5	52.0	51.5	0.05	50.5	50.0	49.5	 <table><caption>Approximate PSNR values for Edge Based Embedding</caption><tr><th>Payload (bpac)</th><th>Level 2</th><th>Level 3</th><th>Level 4</th></tr><tr><td>0.01</td><td>63.5</td><td>63.0</td><td>62.5</td></tr><tr><td>0.02</td><td>61.5</td><td>61.0</td><td>60.5</td></tr><tr><td>0.03</td><td>59.5</td><td>59.0</td><td>58.5</td></tr><tr><td>0.04</td><td>57.5</td><td>57.0</td><td>56.5</td></tr><tr><td>0.05</td><td>55.5</td><td>55.0</td><td>54.5</td></tr></table>	Payload (bpac)	Level 2	Level 3	Level 4	0.01	63.5	63.0	62.5	0.02	61.5	61.0	60.5	0.03	59.5	59.0	58.5	0.04	57.5	57.0	56.5	0.05	55.5	55.0	54.5
Payload (bpac)	Level 2	Level 3	Level 4																																															
0.01	58.5	58.0	57.5																																															
0.02	56.5	56.0	55.5																																															
0.03	54.5	54.0	53.5																																															
0.04	52.5	52.0	51.5																																															
0.05	50.5	50.0	49.5																																															
Payload (bpac)	Level 2	Level 3	Level 4																																															
0.01	63.5	63.0	62.5																																															
0.02	61.5	61.0	60.5																																															
0.03	59.5	59.0	58.5																																															
0.04	57.5	57.0	56.5																																															
0.05	55.5	55.0	54.5																																															

RECOMMENDATIONS

- It is always better to use multiple level encryption such as the one used in our edge based embedding where first the text is encrypted and then embedded in the edges of the image.
- In order to make the LSB algorithm more efficient, edge based embedding approach can be followed.
- Prior testing of the image to be embedded can also help make the algorithm more efficient and secure.