An Approach to Hiding Data in the Images Using Steganography Techniques

BY

Ejaj Ahmed Bhuiyan
ID: 183-15-2300

This Report Presented in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Computer Science and Engineering

Supervised By

Tania Khatun
Sr. Lecturer
Department of CSE
Daffodil International University

Co-Supervised By

Mushfiqur Rahman
Lecturer
Department of CSE
Daffodil International University

DAFFODIL INTERNATIONAL UNIVERSITY

DHAKA, BANGLADESH

SEPTEMBER 17, 2021

APPROVAL

This Project/internship titled **"An Approach to Hiding Data in the Images Using Steganography Techniques"**, submitted by Ejaj Ahmed Bhuiyan, ID No: 183-15-2300 to the Department of Computer Science and Engineering, Daffodil International University has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of B.Sc. in Computer Science and Engineering and approved as to its style and contents. The presentation has been held on 18.09.2021.

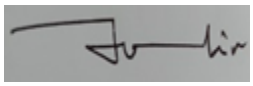
BOARD OF EXAMINERS

Sheak Rashed Haider Noori **Internal Examiner**

Associate Professor

Department of Computer Science and Engineering

Daffodil International University



Ohidujjaman **Internal Examiner**

Senior Lecturer

Department of Computer Science and Engineering

Faculty of Science & Information Technology

Daffodil International University

Dr. Dewan Md. Farid **External Examiner**

Associate Professor

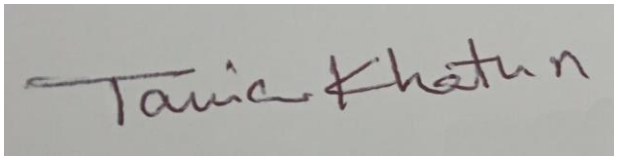
Department of Computer Science & Engineering

United International University, Bangladesh

ANNOUNCEMENT

We hereby declare that, this project has been done by us under the supervision of **Tania Khatun, Sr. Lecturer, Department of CSE** Daffodil International University. We also declare that neither this project nor any part of this project has been submitted elsewhere for award of any degree or diploma.

Supervised by:

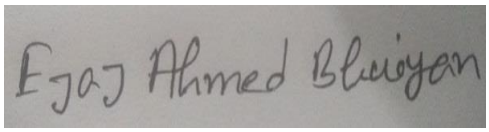


Tania Khatun
Sr. Lecturer
Department of CSE
Daffodil International University

Co-Supervised by:

Mushfiqur Rahman
Lecturer
Department of CSE
Daffodil International University

Submitted by:



Ejaj Ahmed Bhuiyan
ID: 183-15-2300
Department of CSE
Daffodil International University

ACKNOWLEDGEMENT

First we express our heartiest thanks and gratefulness to almighty God for His divine blessing makes us possible to complete the final year project/internship successfully.

We really grateful and wish our profound our indebtedness to **Tania Khatun**,
Sr. Lecturer, Department of CSE Daffodil International University, Dhaka. Deep Knowledge & keen interest of our supervisor in the field of "*Image Processing*" to carry out this project. Her endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, reading many inferior draft and correcting them at all stage have made it possible to complete this project.

We would like to express our heartiest gratitude to **Dr. S.M Aminul Haque**, **Associate Professor & Associate Head,** Department of CSE, for his kind help to finish our project and also to other faculty member and the staff of CSE department of Daffodil International University.

We would like to thank our entire course mate in Daffodil International University, who took part in this discuss while completing the course work.

Finally, we must acknowledge with due respect the constant support and patients of our parents.

ABSTRACT

Steganography is a very useful method of hiding the existence of secret data hidden inside a cover object. Digital Steganography refers to hiding information such as text, images or audio files into another image or audio file.

In digital steganography, Images are mostly used as cover objects because of the wide usage, small file size and the structure of digital images.

Adding Cryptography with Steganography will add another layer of security in digital communication. One distorts the message and the other hides its existence.

Our project aims to develop tools which will allow us to encrypt – decrypt (Cryptography) and hide – retrieve (Steganography) text and image data into and from another cover image. The Steganography technique used here is called **Spatial Domain Technique**. And the implementation was done using C-Sharp (C#).

TABLE OF CONTENTS

CONTENTS **Page No**

LIST OF FIGURES

CHAPTER 1

Introduction

1. What is Steganography?

Steganography is a very old technique for hiding information in plain sight. It has been used throughout human history as a covert way of communication. It is said that the ancient Greeks would shave the head of a messenger and write their message on his head. After some time, his hair would grow back and hide the message. He would then pass through enemy lines without anyone noticing that the valuable message was right in front of them. The messenger would get his head shaved again when he was able to deliver the message to the intended recipient.

Nowadays we send messages mostly by the digital medium. And like ancient times, digital transmissions are also vulnerable. Anyone with the right knowledge can hack a digital transmission medium. This is why most of them use Cryptography as a security measure to distort the real message. But as time goes on and computer hardware is becoming more powerful, it is becoming comparatively easy to break encryptions than before. This is where digital Steganography can help us. To hide the presence of a message or data that we don't want anyone to see or even try to decrypt if it is encrypted. The basic methods of hiding data are 1) Steganography and 2) Cryptography.

Steganography is a simple security method. Its goal is to hide secret data or messages within a cover media such as image or video, in such a way that others cannot notice the existence of the hidden data. In the digital world, it is mostly used in secretive data transmissions, Watermarking, Fingerprinting of digital products, for the privacy of users, etc.

The difference between Cryptography and Steganography is that Cryptography distorts the original data using a key and a specific algorithm. This distorted data can be restored back to its original form using the same key and algorithm. Steganography doesn't distort the data. But hides them into another cover object by modifying data of the cover object. This process is called Encoding. The output that we get from this process is called Stego-Object. The secret data can be retrieved from the Stego-Object using the same method. Which is called Decoding.

Steganography requires three components. They are:

1) A cover object which will hold the secret data.
2) The secret data or message. It can be a string of text or an image.
3) The Stego-object.

Detection of steganography is called Steganalysis.

1.1 How digital Steganography works

Digital Steganography works by replacing unused or less significant bits of the cover medium. This means that the cover medium has to be a file that has enough bits that can be changed without making any significant difference between the old file and the stego-file or corrupting the file entirely. It is done by only replacing the LSB (Least Significant Bit) of the cover file. Changes in LSB does very little change compared to changing MSB (Most Significant Bit.

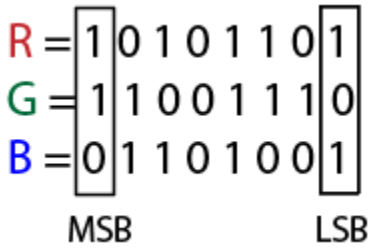


Fig: a1 MSB and LSB

Steganography converts every piece of the input data into a byte array. Then converts each of them into binary, then it takes the last 1, 2, 3 or 4 bits of that input data and replaces with bits from the cover object.

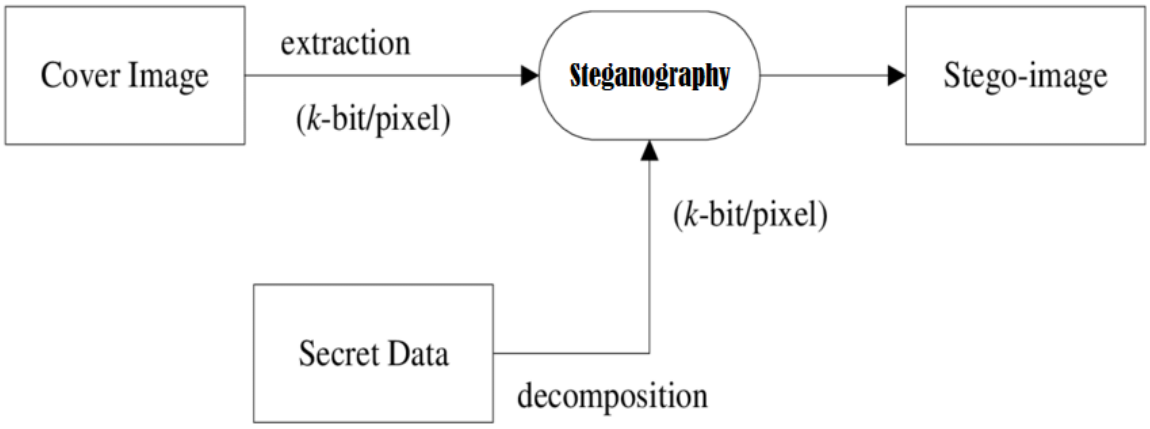


Fig: a2 Replacing LSB with secret data

1.2 Why use Steganography

Encryption protects the contents of a message from unauthorized access. But they are still prone to attacks such as:

1) Brute-Force attack
2) Man in the middle attack
3) Side-channel Attack etc.

If we use steganography to hide the encrypted data's existence, it will be less prone to attacks. The only way to find data hidden in cover files is to perform Steganalysis on suspected files. Steganalysis is a very costly procedure. So it will take attackers a lot of time and processing power to even find the hidden data in the first place.

1.3 Types of Steganography

The most common type of Steganography used today are:

1) Text steganography
2) Image steganography
3) Audio steganography
4) Video steganography

In all of these methods, the main principle of steganography is that a secret data will be embedded into another cover object which may not be important. And hide the secret data in such a way that the output would finally display only the cover data. So it cannot be detected easily by anyone as a Stego Cover object unless proper Steganalysis is used.

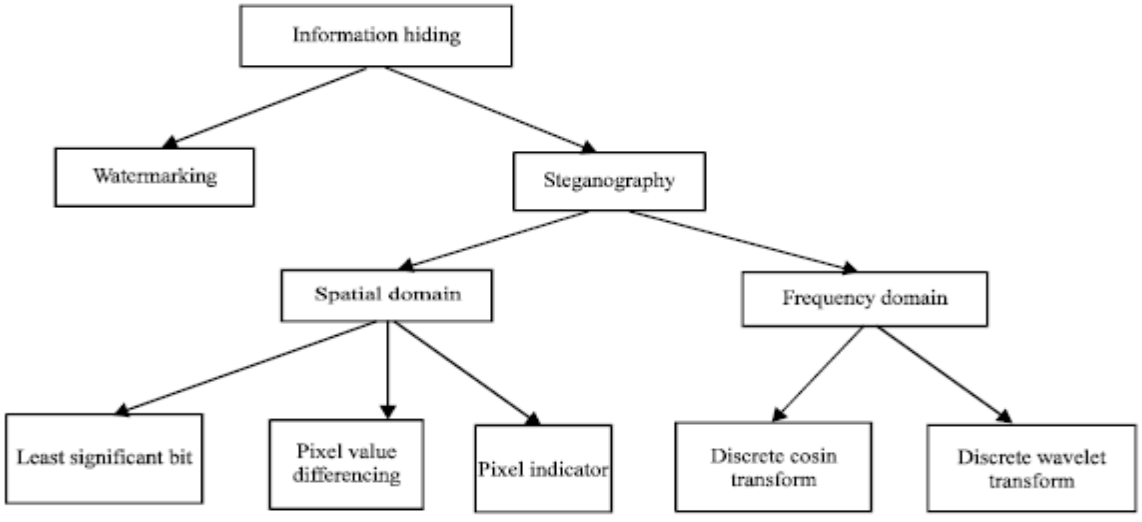


Fig: a3 Types of Steganography

1.4 Motivation for this project

Steganography is a topic that is not well known to everyone. Even though it is a very powerful method of securing information, tools to use Steganography are very rare. This project aims to combine Cryptography with Steganography and develop a system that will allow us to hide secret text or image data inside another cover image. In this project, we will show how to implement the LSB method for Text and Images.

1.5 Project Objective

- To develop a system which will allow us to hide secret text and images inside another cover image.
- To utilize the maximum data hiding capability of any cover image for any given number of LSB replacement.
- To mask the distortion of the cover image caused by the hidden data.
- To provide the ability to encrypt secret data in case of data discovery.
- To provide basic tools to detect LSB based steganography.

CHAPTER 2

Literature Review

G. Prashanti and K. Sandhyarani, [1] had surveyed the achievements of image steganography based on LSB-Replacement in 2015. In that survey, the author discussed the improvements which enhance the steganographic results, like higher robustness, higher embedding capacity, and un-detectability of secret information. He had also proposed two new methods along with that survey. The first method was used to embed secret messages into a cover image and in the second method, a secret grayscale image was embedded into another grayscale image. These methods use a four-state table that produces pseudo-random numbers. This is used for embedding secret information. These two methods had better security because the secret data is hidden inside the pixels of randomly selected locations of the image. The four-state table helped to generate the pseudo-random numbers.

Savita Goel et al [2] proposed a method of hiding secret messages within the cover images using the LSB method by using a different process. The author compare the quality of stego image by comparing it to cover image employing some of image quality parameters, like SSIM (Structure Similarity), PSNR (Peak Signal to Noise Ratio),, histograms and CPU time, and MSE (Mean Square Error) index and having FSIM (Similarity Index Measure). The results from her experiment and study show that her proposed method is efficient, and fast as compared to regular Least Significant Bit methods.

Bingwen Feng, and Wei Sun [3] purposed a state-of-the-art approach of image steganography in 2015. Their method offers to minimize the texture distortion. In this method, the mirroring invariant texture patterns, rotation, and the complement are extracted from the binary image at first. They also offered a measurement, and this approach is basically implemented based on that offered measurement. Results show that the offered steganographic approach has higher statistical security with higher steganographic image quality and higher embedding capacity.

M. Nusrati et al. [4] did a study on a steganographic method based on heuristic genetic algorithm for hiding data for images. It tries to make the least changes in the bits which leads to minimal modifications in the image histogram. To covert the LSBs and secret messages to a set of blocks, segmentation is done in this genetic algorithm. After this algorithm finds the appropriate locations for embedding, the secret blocks are embedded and it generates the key file which is used during the message extraction process. Experimental results show that this genetic-based method is more efficient than the basic LSB algorithm with high stego image quality.

Kazem Qazanfari [5] offered a better version of the LSB++ method. In this technique, the author detects the important pixels and protects them from the embedding process. The result is a lower distortion in the co-occurrence matrix. The author also extends this technique to maintain the DCT coefficients of JPEG images. This technique is also secure

against attacks based on histogram. Because this technique doesn't change the histogram. This is why the histograms of both stego image and cover images will be the same. The resulting stego image also maintains a high quality, because the important pixels were not part of the additional bit embedding process.

P. U. Deshmukh et al. [6] presents a steganography technique based on edge adaptive LSB substitution. The author offers an adaptive scheme and difference between two adjacent pixels of the carrier image, to embed secret data in edges of the cover image. This technique performs better than other Pixel difference and LSB based techniques and the stego image also retains the original quality of the cover image.

R. Modi et al. [7] proposed a novel steganography method to embed hidden data inside the LSBs of the cover image. In this technique, the author utilizes 2 LSBs of the edges of the cover image to store hidden data. Because edges are very good regions to embed the secret data compared to other regions of the cover image. In this technique, edge regions are detected based on the amount of secret data, this means that this technique performs adaptive edge detection. Analyzing the results of experiments shows that, this novel steganography technique has better performance than basic LSB-based image steganographic methods.

D. Samidha [8] offered an LSB-based steganography technique using random bit selection, in the research paper “Random Image Steganography in Spatial Domain” after studying many image steganographic techniques. In this method the LSB is selected randomly to embed the secret data inside the cover image. The author also proposed more methods based on randomly selected pixels of the secret data and cover image which is embedded with random LSBs.

S. Sachdeva and A. Kumar [9] in their method use the vector quantization table to insert the secret data. They proposed a new technique of steganography called JMQT based on a modified version Quantization Table (QT). They also compare their technique

with the JPEG-JSteg steganography method. The Embedding capacity and stego image size were used as parameters for performance test. And the results from their experiments were also compared with the JPEG-JSteg technique. Results from their experiments showed that the data hiding capacity and the size of the output stego image has been increased. We can conclude that, JMQT technique has higher capacity that JSteg. Whereas the JSteg has better stego-size that JMQT.

X. Qing et al. [10] proposed a new technique in which secret data is hidden in all planes of RGB color channels of a cover image. This technique works based on the basis of the Human visual system (HVS). In this technique uses multiple plan-bits, and utilizes the adaptive nature of the data hiding algorithm. This technique which the author proposed, has higher embedding capacity compared to the regular LSB method. And its computational complexity is also lower. The proposed system also keeps the quality of the cover image.

Chapter 3

Methodology

In this project, we will use Bitmap images as cover medium.

A bitmap image is a 2d array or grid of pixels. Each pixel contains information on three colors. Red, Green, and Blue. These are 8 bit integers. So they range from 0 to 255. Since each pixel has three values, they contain 24 bits of data each.

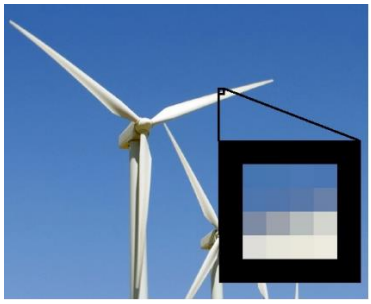


Fig: c1 A bitmap image

3.1 Bitmap Example

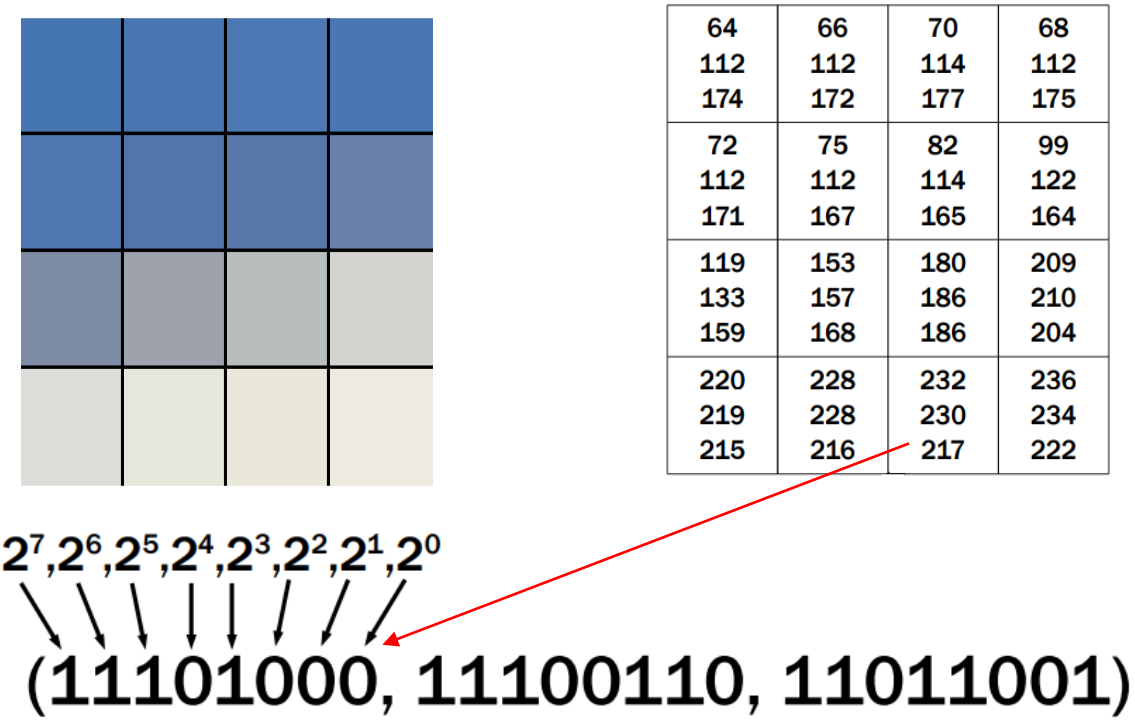


Fig: c2 Binary representation of a pixel

3.2 Image Steganography Methods

There are several different methods of steganography. They are shown below.

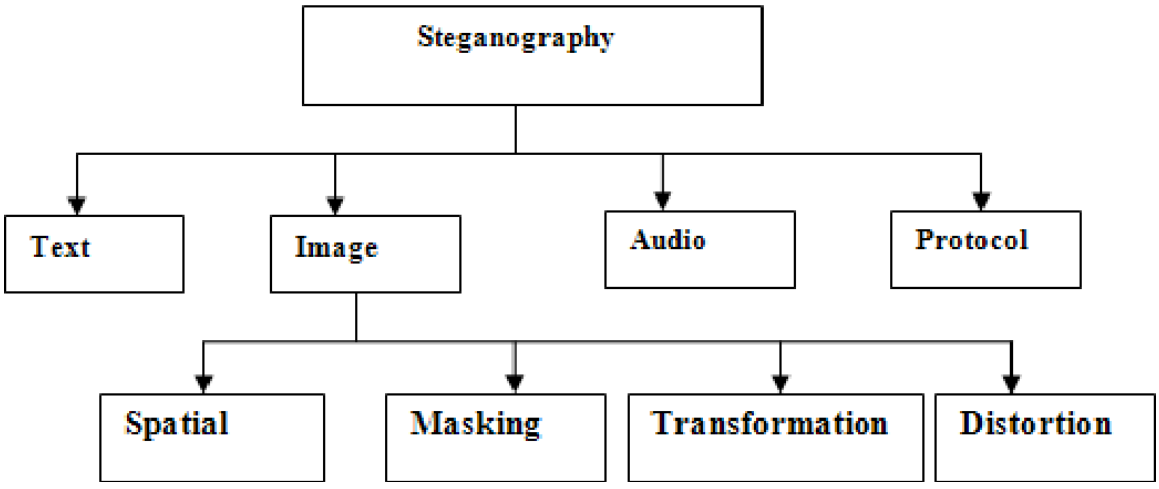


Fig: c3 Types of Steganography

From theses, two are mostly used. They are:

1. Spatial.
2. Transformation.

3.3 Spatial Domain

Spatial domain techniques process the image pixels directly. The pixel values are changed to achieve desired results. Spatial domain techniques like the logarithmic transforms, histogram equalization, are based on the direct manipulation of the pixels in the image. The value of the pixels change with respect to the scene.

The most used spatial domain technique is LSB (Least Significant Bit).

There are two type of LSB techniques we can use.

1. LSB Replacement
2. LSB Matching

3.4 Frequency Domain

In frequency domain, first, we get the frequency distribution of the image. Any processing that need to be done is done on the transformation. The output of the process is not an image. But the altered transformation. To get the image back, we need to perform inverse transformation on the output.

There are three main Frequency Domain technique available to use.

1. DCT (Discrete Cosine Transform)
2. FFT (Fast Fourier Transform)
3. DWT (Discrete Wavelet Transform)
4. DFT (Discrete Fourier Transform)

In this project, we will implement LSB Replacement technique.

3.5 LSB Replacement method of RGB image

Every image has three components (RGB). This pixel information is stored in encoded format in one byte. The first bits containing this information for every pixel can be modified to store the hidden text. For this, the preliminary condition is that the text to be stored has to be smaller or of equal size to the image used to hide the text.

LSB based method is a spatial domain method. But this is vulnerable to cropping, compression and noise. In this method, the MSB (most significant bits) of the message image to be hidden are stored in the LSB (least significant bits) of the image used as the cover image.

The main idea of LSB replacement is to replace one or two LSB bits from the cover image with bits of secret data (text / images).

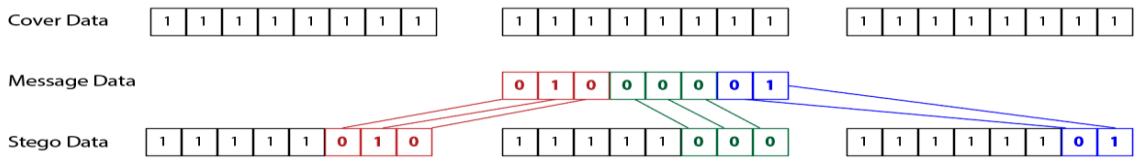


Fig: c4 Replacing 3 LSB of cover data.

The Human visual system (HVS) cannot detect changes in the color or intensity of a pixel when the LSB bit is modified. This is a known psycho-visual redundancy of human eyes. Which is why this can be used as an advantage to store information in these bits. And not have any noticeable difference in the output bitmap image.

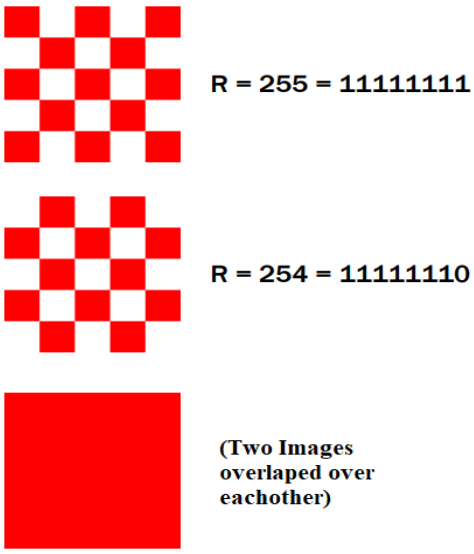


Fig: c5 Two pixels with different shades of red overlapped looks the same to us

3.5 Steganalysis

Steganalysis is the process of detecting steganographic images. LSB steganography depends on altering n number of LSB of the color values of a cover image.

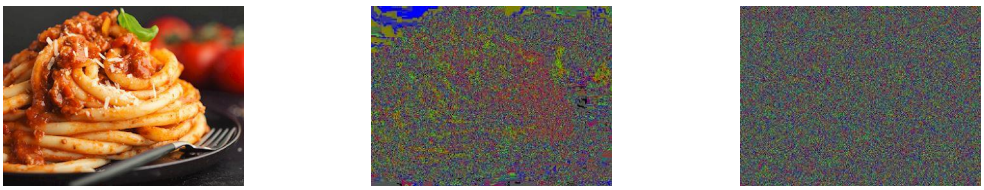
We can detect stego-images by isolating n amount of LSBs and display them alone. This is called Linear Isolation.

The formula for RGB image normalization for every pixels is as follows:

$$Normalized\ channel\ value\ =\ channel\ value\ /\ (Red\ +\ Green\ +\ Blue)\ *\ 255$$

If we have two copies of an image, and one of them is suspected as a stego image, then we can make a subtraction image from them and compare their pixel values.

Example of Steganalysis:



Original 2-bit LSB of original 2-bit LSB of Stego-Image

Fig: c6 Normalization of an original and stego-image

CHAPTER 4

Implementation

To implement LSB Replacement, we first need to define the steps we need to take.

Steps used in LSB Replacement:

a. Steps for embedding data in image:

1. Read cover image.
2. Read secret data (text / image).
3. Convert cover image and secret data to byte stream.
4. Embed secret data into cover image.
5. Convert output of embedding process to image.

From our discussion above, we know that the LSB (least significant bit) plane contains the least information associated with any image, and the MSB (most significant bit) plane contains most of the shape, color information of an image.

It is generally ideal to replace up to 3 LSB of the cover image with data bits without revealing any changes. The best choice would be to replace only 1 LSB. Replacing more than 3 will allow us to embed more and bigger data into a single image. But it will also distort the cover image to such extent that the distortion will be easy to detect just by looking. The amount of visible distortion which will be caused by replacing LSB depends on the number of LSBs replaced and also on the image itself.

b. Steps for retrieving data:

1. Read the Stego-Image.
2. Convert into byte stream.
3. Read LSBs of each byte until the end indicator if found.
4. Combine the LSBs.
5. Convert to Image.

4.1 LSB Replacement example

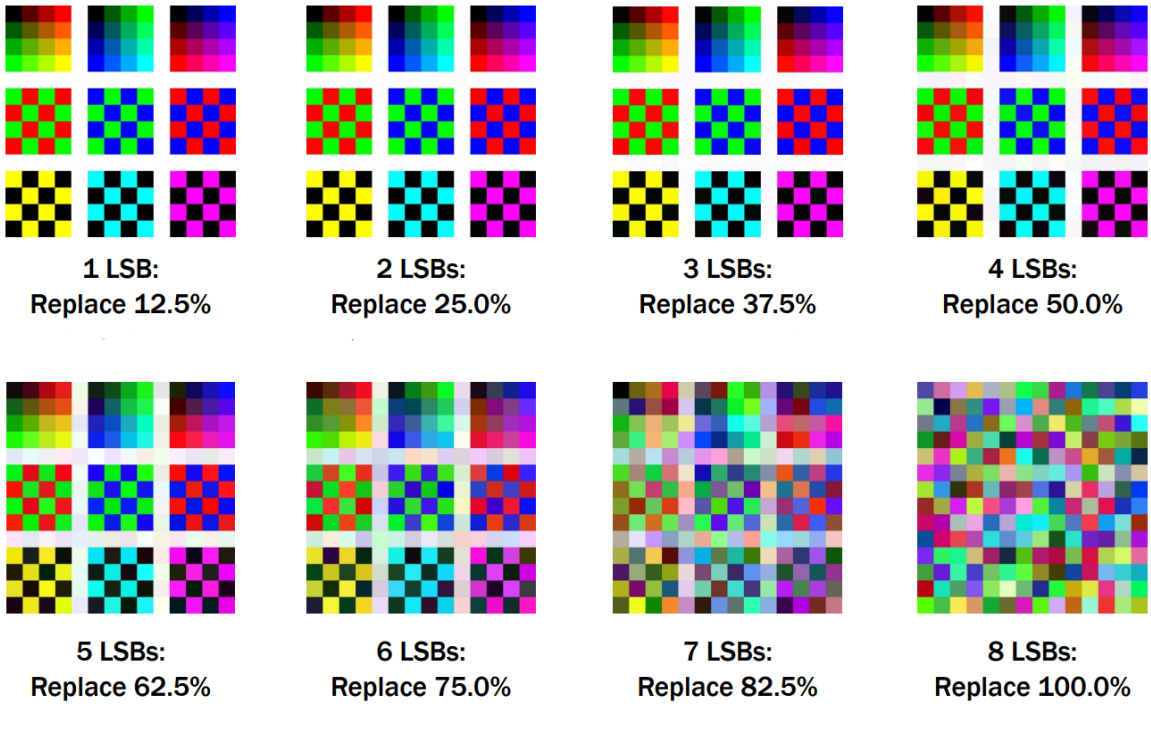


Fig: d1 Result after replacing 1 to 8 LSBs from a colorful image

From the figure above, we can see that an image with various colors show more visible distortion in color values compared to images with less color variation.

Images with less colors does not become distorted easily.

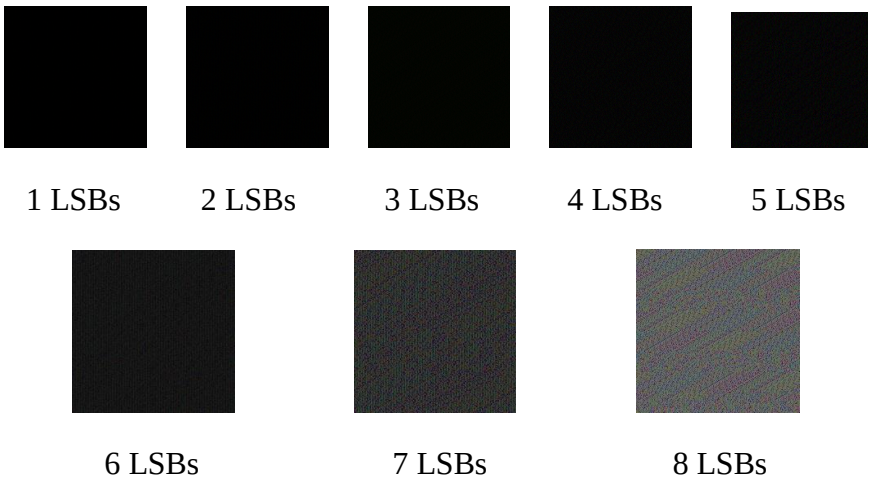


Fig: d2 Result after replacing 1 to 8 LSBs from a single color image

The process of encoding both text and images are quiet the same, only with some minor changes.

If the message data is short, embedding them will cause noise in a portion of the bit-plane where they were added. And the rest will have no noise. This is why we add random bits after the data so that the distribution of noise become even across the cover image.

4.2 Embedding Texts using LSB

Computers cannot recognize characters as they are. Which is why computers use Encoding table to show characters. Theses encoding tables identify characters by certain sequence of binary digits. There are many encoding tables such as ASCII, UTF-8, UTF-16, etc. Among them, UTF-8 is the most popular and ASCII and UTF-8 both share the same encoding table and use 8 bits to represent a single character.

In this project, we will be using ASCII / UTF-8 encoding to handle input characters.

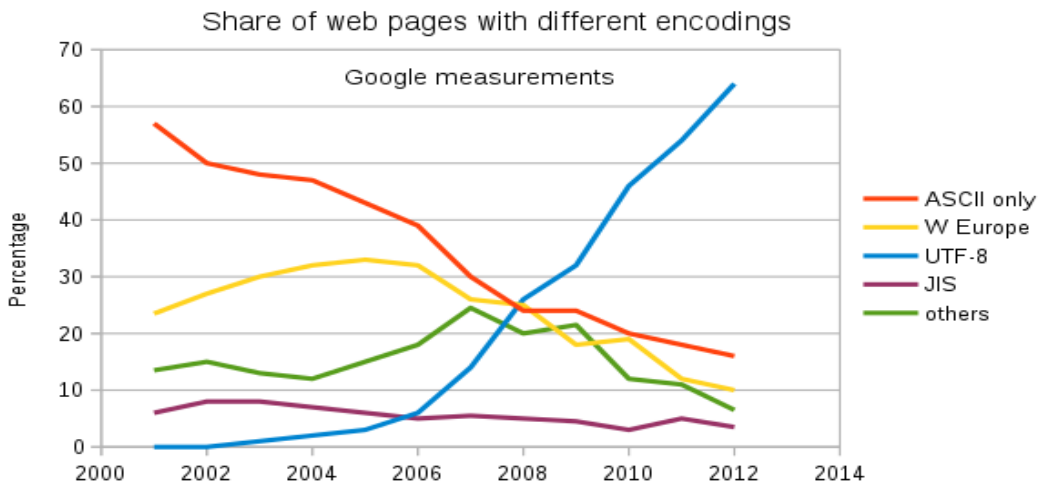


Fig: d3 usage of various encoding tables (Source: https://en.scratch-wiki.info/w/images/Encodings_Popularity.png)

If the number of characters in secret data is lower than the embedding capability of the image, then it will create very tiny, un-even noise. This noise is very small for human eyes to detect, but proper Steganalysis might be able to detect the existence of the secret data.

To avoid this detection, we will add randomly generated bytes after the message bytes. These random bytes will fill the cover image. Since theses bytes are random, we do not need to retrieve them. This noise will be distributed randomly throughout the image which will reduce the risk of detection. To distinguish between the message and random bytes, we use a special characters.

Procedure for embedding text:

1. Read Cover image.
2. Read secret text.
3. If secret text contains the “End of message” indicator, then replace them with something similar.
4. Add “End of message” indicator at the end of secret text.
5. Convert character string into byte array.
6. Add random bytes after the secret text bytes.
7. Convert secret text byte array to an array of their binary representation.
8. Convert cover image to an array of pixels.
9. Loop through every pixel of the pixels array.
10. Embed secret data into the image by replacing n LSBs of Red, Green, And Blue channel of required pixels with n LSBs of modified secret data.
11. Convert the modified pixels array to image.

Procedure for retrieving text:

1. Read Stego-Image.
2. Convert image into pixels array.
3. From each pixel, Extract n number of LSB from Red, Green, and Blue channel.
4. If **8 bit** extracted, convert them to byte.
5. Add extracted byte to new array.
6. Convert the extracted bytes to character string.

Given a secret message, we can use the below equation to calculate the minimum number of pixels the Cover image must have in order to hide them.

$$((l + 1) * (8 / n)) / 3$$

Where $l =$ number of characters of the secret message, $n =$ number of LSBs to replace.

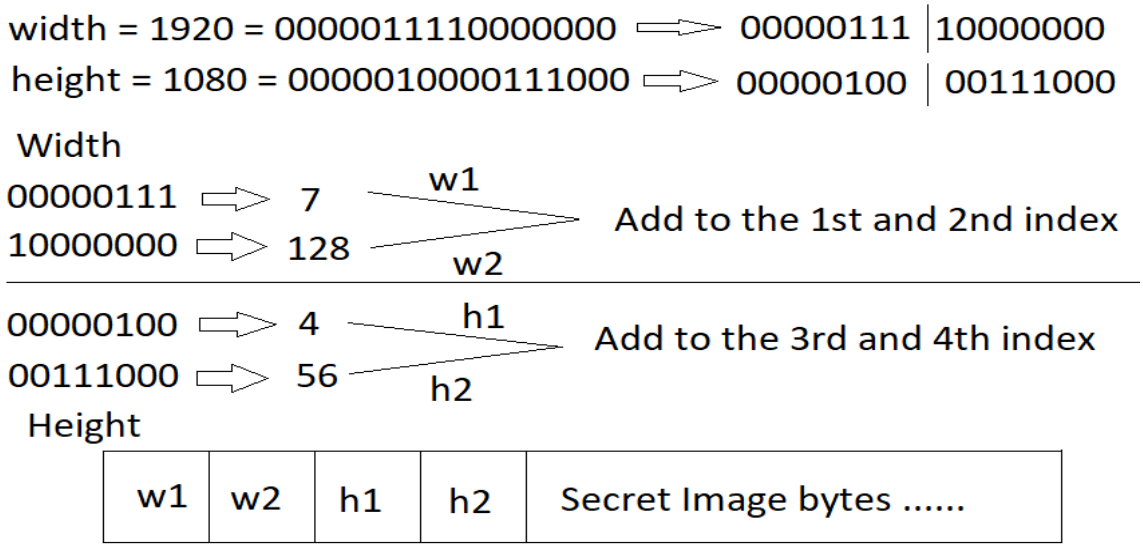
On the other hand, if we only have a cover image, we can calculate how many characters we can hide in that image using the following equation.

$$(((n * 3) * w * h) / 8) - 1$$

Where $n =$ number of LSBs to replace, $w =$ width, $h =$ height of the cover image.

4.3 Embedding Images using LSB

Embedding image is also similar to embedding texts. But we need to provide some additional information such as image width and image height, along with the pixel values. Since image resolution usually don't exceed 7680 X 4320 pixels (8K), we can store their binary representation in two 16 bit variables. We also need to split the 16 bits in half. Which will give us four 8 bit or 1 byte values. 1st and 2nd byte will contain 1st half and 2nd half of the width. 3rd and 4th byte will contain 1st and 2nd half of the height.
Example:



Procedure for embedding an image inside another image:

1. Read secret image.
2. Read cover image.
3. Convert the secret image into byte array.
4. Convert Cover image into pixels array.
5. Get **16 bit** binary representation of the width and height of the secret image.
6. Split **16 bits** of both width and height in half.
7. Take a new byte array.
8. Add the four 8 bit values at the beginning of the new byte array.
9. Add secret image bytes to the new byte array
10. Add random bytes to the new byte array.
11. Convert the new byte array to binary.
12. Embed secret data into the image by replacing n LSBs of Red, Green, And Blue channel of required pixels with n LSBs of modified secret data.
13. Convert the modified pixels array to image.

Before embedding, we need to check if the cover image has enough pixels to hide the secret image or not. We can do the following calculation to check this.

$$Ceiling((((w * h * 3) + 4) * (8 / n)) / 3)$$

Where $w =$ width of secret image, $h =$ height of secret image, n = number of LSBs to replace.

For a cover image, the maximum number of pixels it can hide can be determined by

$$Floor(((n * 3) * w * h) / 24) - 1$$

Where $w =$ width of secret image, $h =$ height of secret image, n = number of LSBs.

Procedure of retrieving image:

1. Read Stego-Image.
2. Convert to pixels array.
3. Extract first 16 (8 + 8) bits.
4. Extract Second 16 (8 + 8) bits.
5. Convert extracted bits to integers.
6. Set width to first extracted integer value.
7. Set height to second extracted integer value.
8. From each pixel, Extract n number of LSB from Red, Green, and Blue channel.
9. If 8 bit extracted, convert them to byte
10. Add to a byte array.
11. Convert byte array to image.

We can see that in both text and image embedding, the cover image needs to be converted into a 1d array of bytes. The following figure show an example of how to do that.

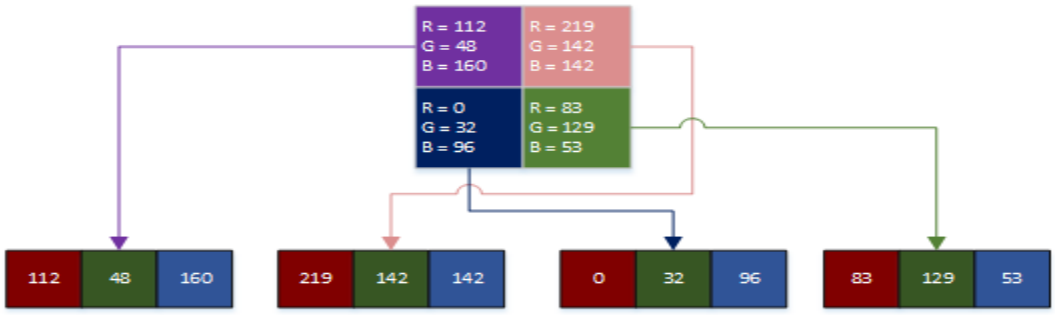


Fig: d4 Turing a 2x2 image into a 1d array

4.4 Encryption

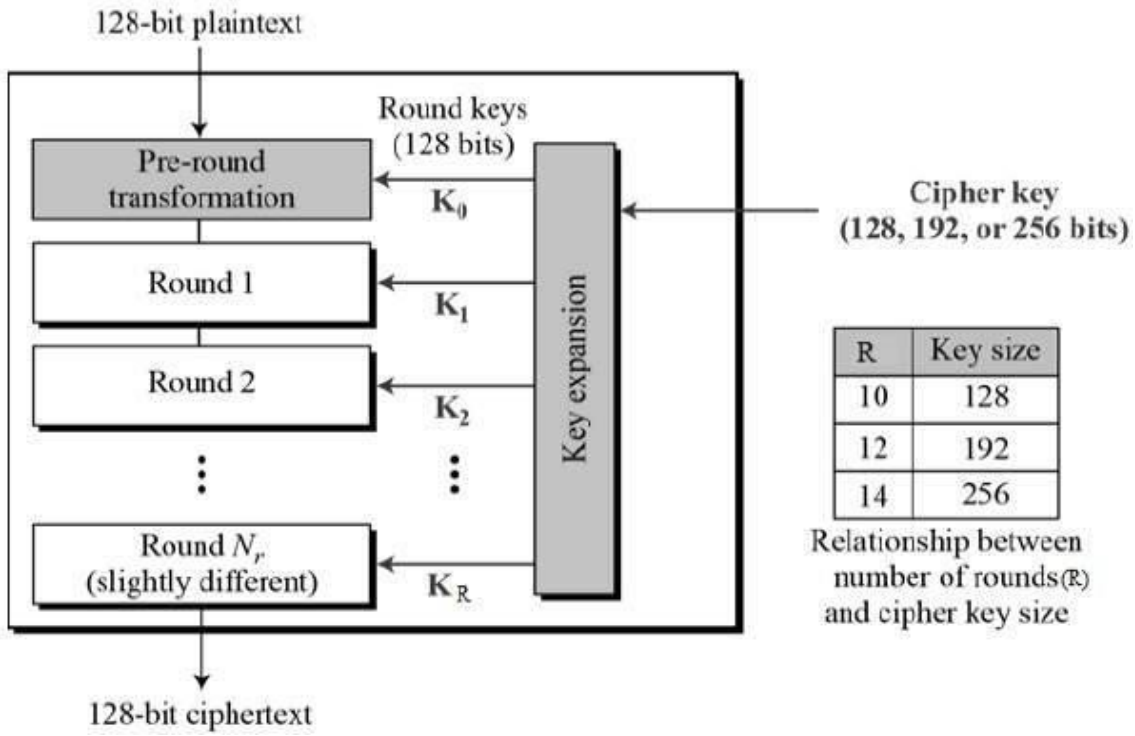
In this project, AES and RSA is used to secure hidden text, and AES and a hybrid of AES + RSA is used to protect images.

AES: AES works by performing bitwise XOR operation on the key and the plaintext or message multiple time. Since we cannot do it to the characters directly, they have to be converted to byte array. AES does bitwise XOR to the binary of theses bytes.

The amount of time this operation happens depends of the key size. 10 times for 128 bit key, 12 times for 192 bit key, 14 times for 256 bit key. Bigger key length means that the cypher will be harder to break.

AES divides the message into many blocks. The blocks have the same bit length as the key.

RSA: RSA is an Asymmetric encryption algorithm. A pair of two different keys are used in this system. One public key which can only be used to encrypt a message, and one private key which can only be used to decrypt an encrypted message. The public key can be shared publicly with anyone. While the private key must be kept secret. The typical size of an RSA key pair is 2048, or 4096 bits.



R	Key size
10	128
12	192
14	256

Relationship between number of rounds(R) and cipher key size

Fig: d5 AES encryption process. (Source: https://www.tutorialspoint.com/cryptography/images/aes_structure.jpg)

But most of the time the message that we write does not have enough bits to fill a block. If this happens then the AES algorithm will not work. To fix this problem, we use paddings. Paddings are some specific values that are added to the secret message in some specific way to fill that gap. There are many padding modes available such as:

1. PKCS7
2. Zeros
3. ISO 10126
4. ANSIX923
5. None

All these padding mode add some extra bytes at the end of the message bytes to fill the last block.

4.5 Problem with encrypting images using AES Block Cypher

As we have seen above from the brief description of AES, if the message length is not a multiple of 32, meaning does not have enough bits to fill a 128 bit block, then we will have to use a padding mode.

But when encrypting images, the extra bytes added to the image bytes will prevent us from reconstructing the encrypted image. And if we ignore them, and reconstruct the image, then we won't be able to decrypt them because the padding mode will require those extra bytes that were added.

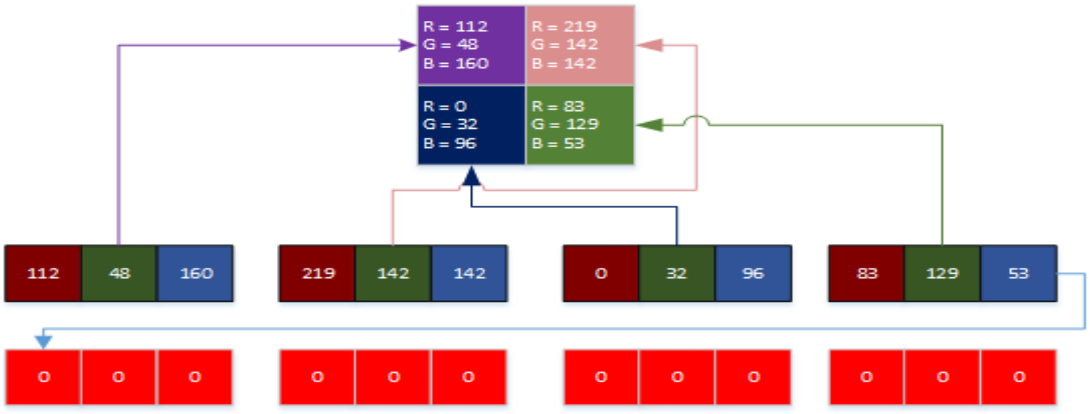


Fig: d6 Extra bytes were added to the array (in red)

4.6 Solution of image encryption problem using AES Block Cypher

The easy way to solve this problem is to not to use any padding mode at all. And to avoid the block size mismatching problem, we simply have to add our own padding to the byte array. This will allow the AES algorithm to encrypt our image. And after encrypting it, we can just ignore those extra bytes. Since we are not using any padding mode, the decryption algorithm will not look for any. But we do need to perform the same process of adding our own padding to the encrypted image bytes, before decrypting it. Which means that, we must add the same amount of padding bytes to the byte array as we did before encrypting.

To calculate how many bytes needs to be added as padding, we use this formula:

$$missing\_bytes = (Ceiling((l * 8.0f) / (bs * 1.0f)) * bs - l * 8) / 8$$

Where l = size in byte array, bs = AES block size.

4.7 Flowcharts

Flowchart for replacing bits:

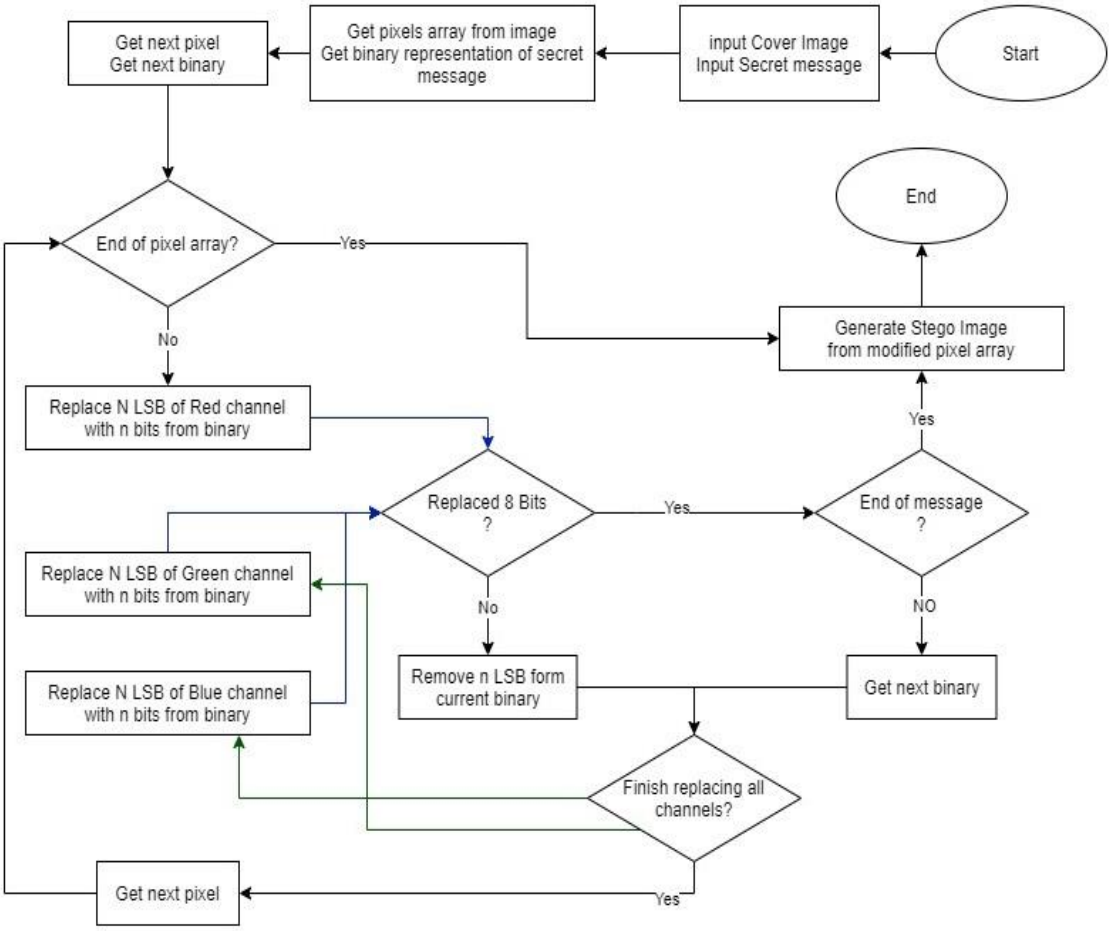


Fig: d7. Flowchart for replacing bits

Flowchart for extracting bits:

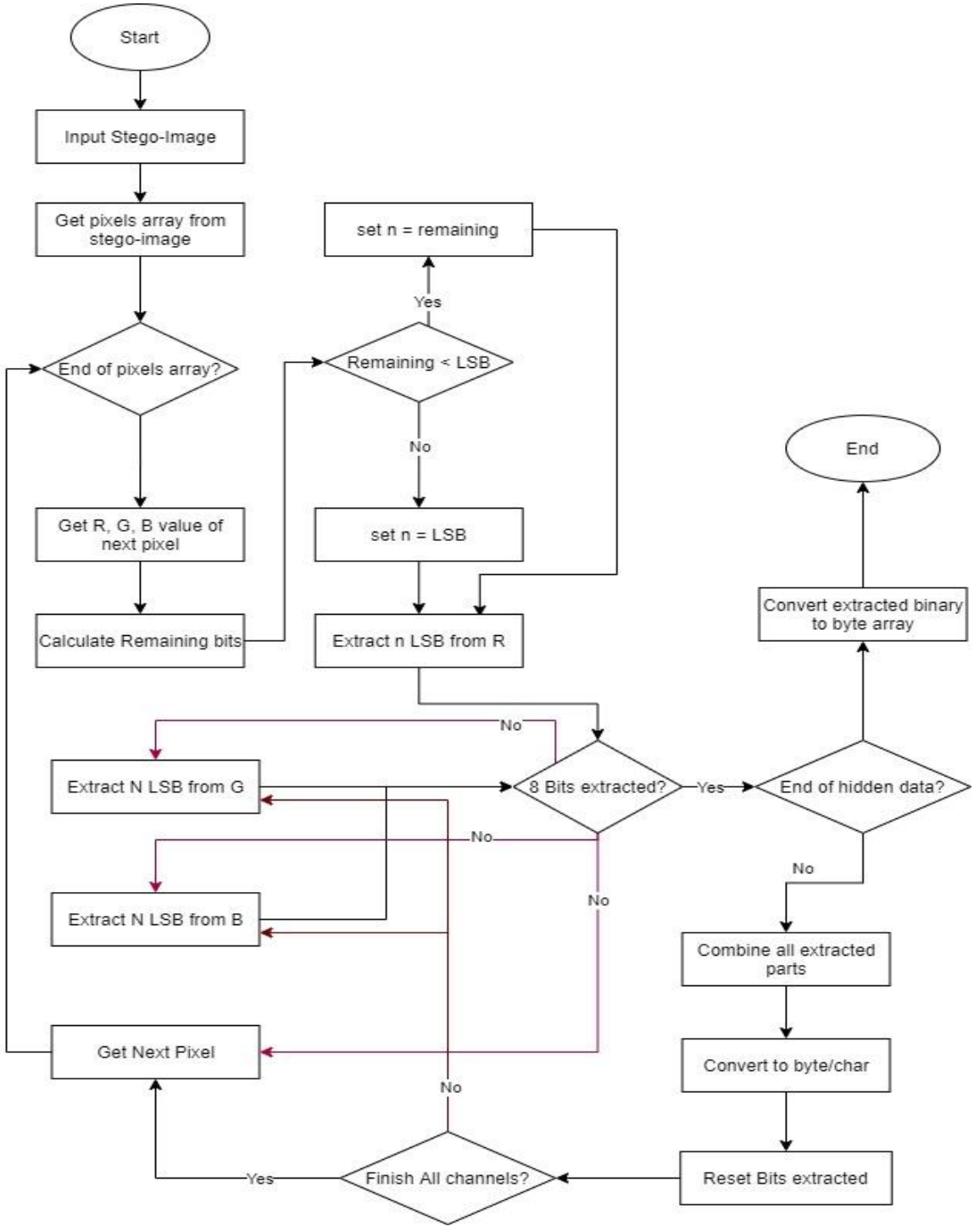


Fig: d8. Flowchart for retrieving bits

Chapter 5

Results

For image set 1: In this set we will be hiding a string of randomly generated text. It contains **4564 words, and 30828** characters.

Cover Image

Fig: e1 Original image. 1920 x 1200 pixels, size = 1 mb

Stego Images

Fig: e2 3-bit LSB replacement, hiding capacity = 2591999 characters, size = 4.83mb

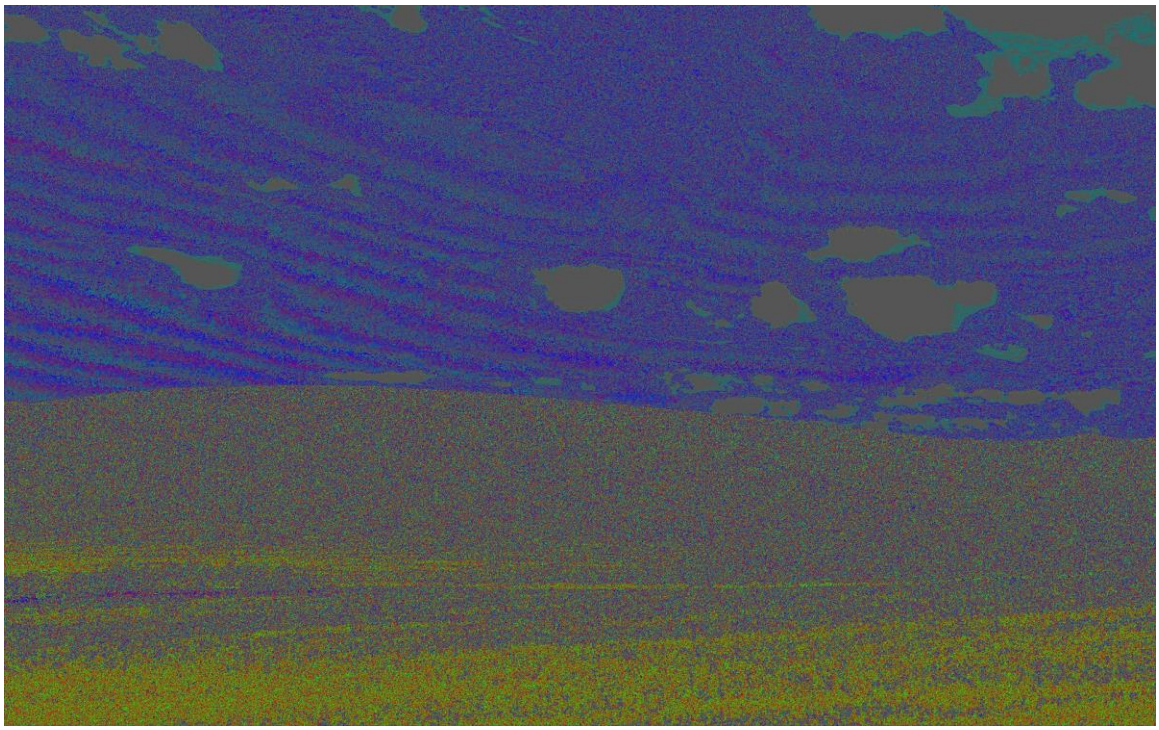


Fig: e3 3-bit original image, Normalized

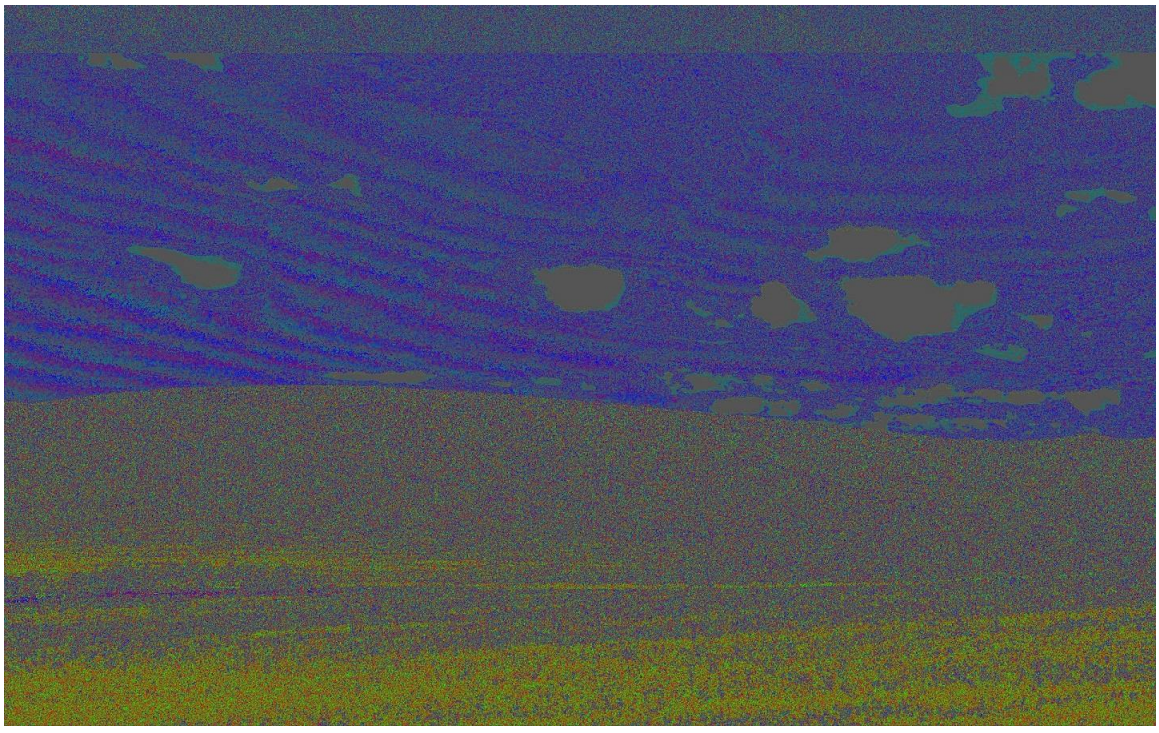


Fig: e4 3-bit stego image, normalized

Extracted data: The retrieved starts with: “This is a secret text which is needed to be hidden”, and contains **4564 words, and 30828** characters.

For image set 2: In this set, we will be hiding images as secret data. The secret image has a resolution of 375 x 280 pixels.

Cover Image

Fig e5 Original image. Resolution 1457 x 1170 pixels, size = 710kb

Secret-Image

Fig: e6 secret RGB image. Resolution 375 x 280 pixels

Stego-Images

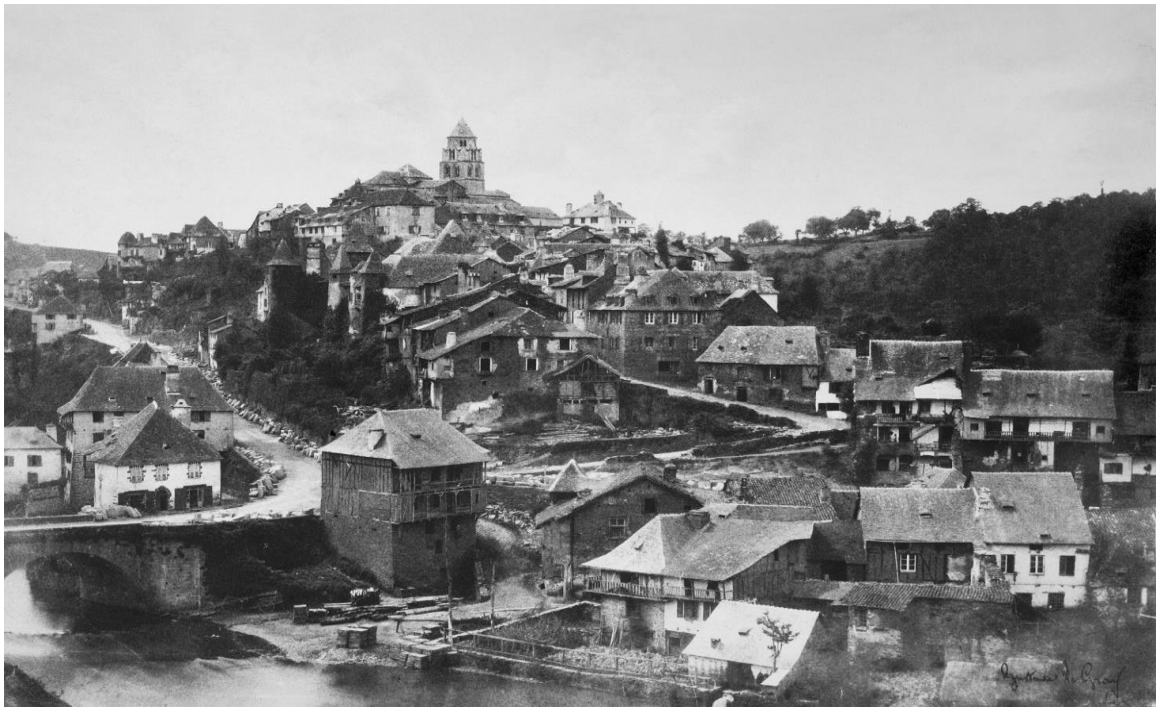


Fig: e7 1-bit LSB replacement, hiding capacity = 213085 pixels, size = 2.68mb

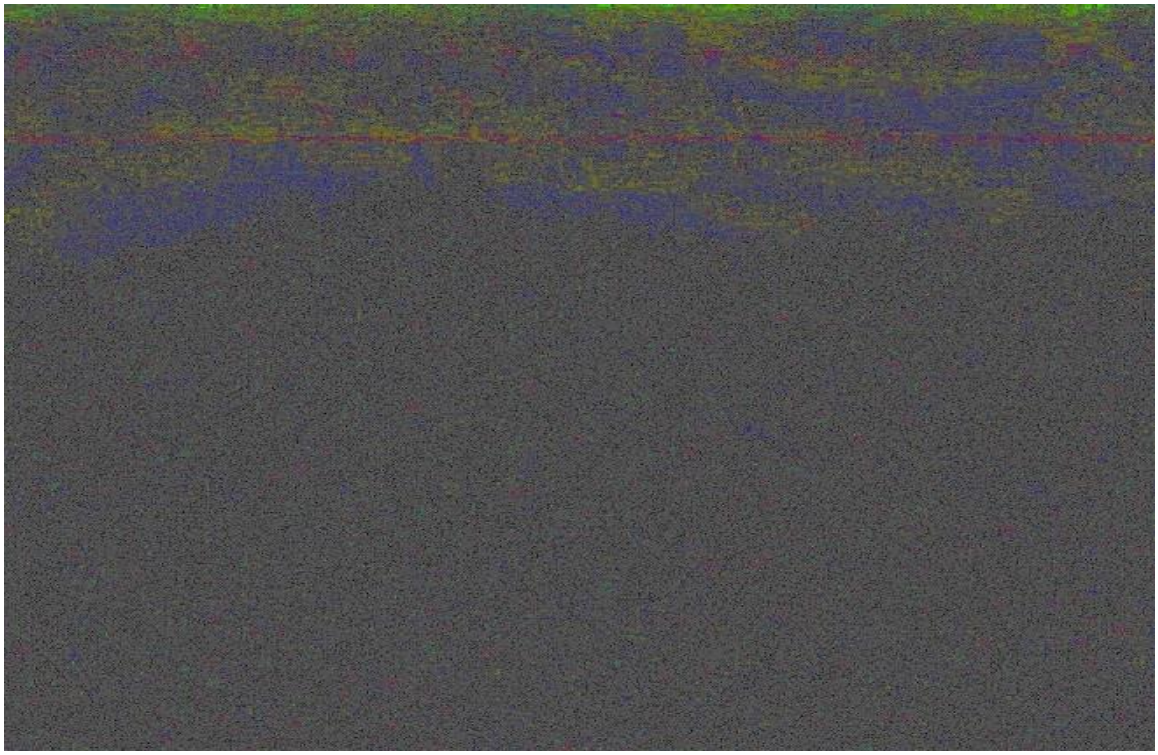


Fig: e8 1bit image of original, Normalized.

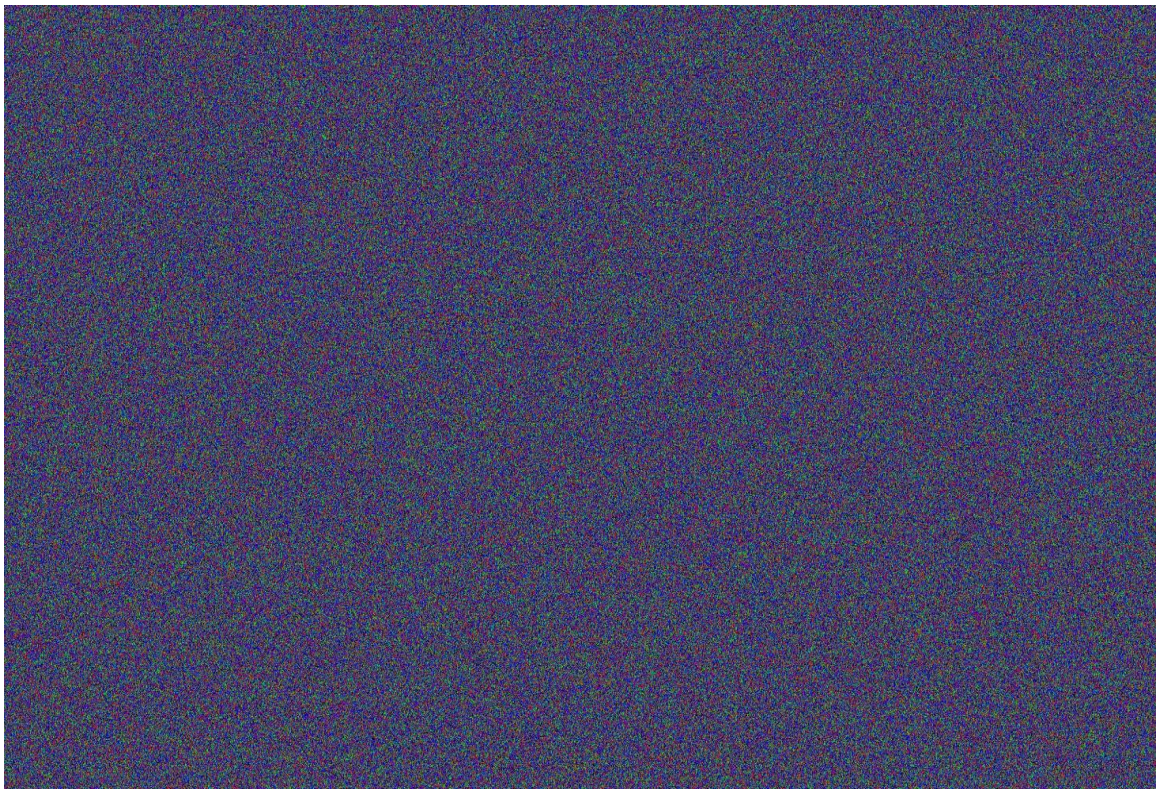


Fig: e9 1-bit stego image, Normalized

Extracted message image

Fig: e10 Extracted secret image from all the stego-images

Chapter 6

Conclusion

By using LSB Substitution method for Steganography, the results we have got in data hiding are very impressive. Because it utilizes the fact that, any image can be broken up to individual bits. Each of them containing different levels of information.

We should note that, this method only works for bitmap images. Because bitmaps do not include any compression methods. Also, in this project, we have used RGB images for demonstration purposes. But this process can also be extended for gray-scale images. Where every pixel has only one value.

For images, we observed that, the extracted images (Fig e.10) retains almost the same quality as the original secret image. From Figs. e.4 and e.9, we can see that when more than 3 LSBs are replaced, the distortion caused by the embedded data becomes visible to the human eyes.

To obtain low distortion, it is advised to use only 1 LSB for embedding, if possible. This results in the best result. Replacing more than 3 LSBs will introduce visible distortion in the image.

We should also mention that, even thou steganographic techniques such as LSB Replacement, or LSB matching was once un-detectable, nowadays it has become easier to detect steganographic images. For instance, even without using any software or complex tools, we can detect possible stego-images by simply observing 2 factors:

1. Size of stego-images: LSB-based Stego-Images has huge file size compared to regular images of the same resolution. Where a regular JPEG image has a file size of a few kb, a Stego-image may have size in mb. From our results, we can see that, all the stego-images has at least ten times the size of their original image. Although this can be partially attributed to the Bitmap image formatting.

2. Noise: Embedding data into an image introduces noise in it. Every stego-image has noise when compared to its original. If higher bit-planes are used, then the noise will become visible to naked eyes.

In this project, we focused on LSB steganography which is a spatial domain technique. From the many articles and theory available, we have learned that frequency domain methods performs better than spatial domain techniques.

From our results, we can conclude that our result has higher data hiding capacity compared to frequency domain techniques, but higher stego-file size.

Appendix

PSEUDOCODE for text embedding:

```
encode_text(cover_pixels_array[], message, lsbCount, capacity, add_noise = true)
   message = Replace(message, '`', '\'')
   message += end_char
   if add_noise then
      remaining = Abs(capacity - message.Length)
      chars = "abcdefghijklmnopqrstuvwxyz!@#$%^&*()_+[]ABCDEFGHIJKLMNOPQRSTUVWXYZ \\|:'\",<.>/?0123456789"
                        while message.Length < capacity do
                                 message += chars[Random(0, chars.Length)]
                        end while
   end if
   message_binary[] = stringtobinary(message)
   current_pixel_index = 0
   current_msg_index = 0
   cover_pixel = cover_pixels_array[current_pixel_index]
   current_binary = message_binary[current_msg_index]
   bits_written = 0
   while current_msg_index < message_binary.Length do
      mod_red = 0
      mod_green = 0
      mod_blue = 0
      temp_binary = byte_to_binary(cover_pixel.R)
      appendBits(&temp_binary, &current_binary, lsbCount, &bits_written)
      mod_red = todecimal(temp_binary)
      if bits_written == bit_max_size then
         current_msg_index += 1
         if current_msg_index >= message_binary.Length then
            cover_pixels_array[current_pixel_index] = Color.FromArgb(mod_red, cover_pixel.G, cover_pixel.B)
            break
         else
            current_binary = message_binary[current_msg_index]
            bits_written = 0
         end if

      temp_binary = byte_to_binary(cover_pixel.G)
      appendBits(&temp_binary, &current_binary, lsbCount, &bits_written)
      mod_green = todecimal(temp_binary)
```

```
    if bits_written == bit_max_size then
      current_msg_index += 1
      if current_msg_index >= message_binary.Length then
        cover_pixels_array[current_pixel_index] = Color.FromArgb(mod_red, mod_green, cover_pixel.B)
        break
      else
        current_binary = message_binary[current_msg_index]
        bits_written = 0
      end if

    temp_binary = byte_to_binary(cover_pixel.B)
    appendBits(&temp_binary, &current_binary, lsbCount, &bits_written)
    mod_blue = todecimal(temp_binary)

    if bits_written == bit_max_size then
      current_msg_index += 1
      if current_msg_index >= message_binary.Length then
        cover_pixels_array[current_pixel_index] = Color.FromArgb(mod_red, mod_green, mod_blue)
        break
      else
        current_binary = message_binary[current_msg_index]
        bits_written = 0
      end if
    cover_pixels_array[current_pixel_index] = Color.FromArgb(mod_red, mod_green, mod_blue)
    current_pixel_index += 1
    if current_pixel_index >= cover_pixels_array.Length then break end if
    cover_pixel = cover_pixels_array[current_pixel_index]
  return cover_pixels_array
```

PSEUDOCODE for text retrieving:

```
decode_text(pixel_array[], lsbCount){
  result = ""
  current_pixel_index = 0
  Color current_pixel = pixel_array[current_pixel_index]
  bits_extracted = 0
  binary = 0
  remaining_bits = 0
  extracted_char
  num_zeros = Power(10, lsbCount)

  while current_pixel_index < pixel_array.Length do
    current_pixel = pixel_array[current_pixel_index]
    remaining_bits = bit_max_size - bits_extracted
                    n = 0
                    if remaining_bits >= lsbCount then
                            n = lsbCount
                    else
                            n = remaining_bits
                    end if
    num_zeros = Power(10, bits_extracted)
    binary = extractBinary(current_pixel.R, n, &bits_extracted) * num_zeros + binary
    if bits_extracted >= bit_max_size then
      chr = todecimal(binary)
      extracted_char = GetString(chr)
      if extracted_char == end_char then break end if
      else
        result += extracted_char
```

```
            bits_extracted = 0
            binary = 0
        end if
    end if
    remaining_bits = bit_max_size - bits_extracted
                    n = 0
                    if remaining_bits >= lsbCount then
                            n = lsbCount
                    else
                            n = remaining_bits
                    end if
    num_zeros = Power(10, bits_extracted)
    binary = extractBinary(current_pixel.G, n, &bits_extracted) * num_zeros + binary
    if bits_extracted >= bit_max_size then
        chr = todecimal(binary)
        extracted_char = GetString(chr)
        if extracted_char == end_char then break end if
        else
            result += extracted_char
            bits_extracted = 0
            binary = 0
        end if
    end if
    remaining_bits = bit_max_size - bits_extracted
                    n = 0
                    if remaining_bits >= lsbCount then
                            n = lsbCount
                    else
                            n = remaining_bits
                    end if
    num_zeros = Power(10, bits_extracted)
    binary = extractBinary(current_pixel.B, lsbCount, &bits_extracted) * num_zeros + binary
    if bits_extracted >= bit_max_size then
        chr = todecimal(binary)
        extracted_char = GetString(chr)
        if extracted_char == end_char then break end if
        else
            result += extracted_char
            bits_extracted = 0
            binary = 0
        end if
    end if
    current_pixel_index += 1
  end while
  return result
}
```

PSEUDOCODE for image embedding:

```
encode_image(host_colors[], data_colors[], width, height, lsbCount, capacity, add_noise = true)
            width_bi = to_binary(width)
  width_bi_half_2 = width_bi % 100000000
  width_bi /= 100000000
  width_bi_half_1 = width_bi % 100000000

  height_bi = to_binary(height)
  height_bi_half_2 = height_bi % 100000000
  height_bi /= 100000000
```

```
height_bi_half_1 = height_bi % 100000000

w_h_1 = todecimal(width_bi_half_1)
w_h_2 = todecimal(width_bi_half_2)
h_h_1 = todecimal(height_bi_half_1)
h_h_2 = todecimal(height_bi_half_2)

data_byte_array[data_colors.Length * num_channels + 4]
data_byte_array[0] = w_h_1
data_byte_array[1] = w_h_2
data_byte_array[2] = h_h_1
data_byte_array[3] = h_h_2
reserved_bytes = 4

for i = reserved_bytes upto data_colors.Length + reserved_bytes do
   data_byte_array[(i * num_channels) - reserved_bytes * 2] = data_colors[(i - reserved_bytes)].R
   data_byte_array[(i * num_channels + 1) - reserved_bytes * 2] = data_colors[(i - reserved_bytes)].G
   data_byte_array[(i * num_channels + 2) - reserved_bytes * 2] = data_colors[(i - reserved_bytes)].B
end for

data_color_channels_with_noise[]
data_color_channels_with_noise += data_byte_array

if add_noise then
                        for i = data_byte_array.Length upto capacity * 3 do
                                data_color_channels_with_noise[i] = Random(0, 255)
                        end for
end if

data_bytes_binary[] = bytes_to_binary(data_color_channels_with_noise)
current_pixel_index = 0
current_pixel = host_colors[current_pixel_index]
current_data_binary_index = 0
current_data_byte_binary = data_bytes_binary[current_data_binary_index]
bits_written = 0

while current_data_binary_index < data_bytes_binary.Length do
   mod_red = 0
   mod_green = 0
   mod_blue = 0

   temp_binary = byte_to_binary(current_pixel.R)
   appendBits(&temp_binary, &current_data_byte_binary, lsbCount, &bits_written)
   mod_red = todecimal(temp_binary)

   if bits_written == 8 then
      current_data_binary_index += 1
      if current_data_binary_index >= data_bytes_binary.Length then
         host_colors[current_pixel_index] = Color.FromArgb(mod_red, current_pixel.G, current_pixel.B)
         break
      else
         current_data_byte_binary = data_bytes_binary[current_data_binary_index]
         bits_written = 0
      end if
   end if

   temp_binary = byte_to_binary(current_pixel.G)
   appendBits(&temp_binary, &current_data_byte_binary, lsbCount, &bits_written)
   mod_green = todecimal(temp_binary)
```

```
    if bits_written == 8 then
      current_data_binary_index += 1
      if current_data_binary_index >= data_bytes_binary.Length then
        host_colors[current_pixel_index] = Color.FromArgb(mod_red, mod_green, current_pixel.B)
        break
      else
        current_data_byte_binary = data_bytes_binary[current_data_binary_index]
        bits_written = 0
      end if
    end if

    temp_binary = byte_to_binary(current_pixel.B)
    appendBits(&temp_binary, &current_data_byte_binary, lsbCount, &bits_written)
    mod_blue = todecimal(temp_binary)

    if bits_written == 8 then
      current_data_binary_index += 1
      if current_data_binary_index >= data_bytes_binary.Length then
        host_colors[current_pixel_index] = Color.FromArgb(mod_red, mod_green, mod_blue)
        break
      else
        current_data_byte_binary = data_bytes_binary[current_data_binary_index]
        bits_written = 0
      end if
    end if

    host_colors[current_pixel_index] = Color.FromArgb(mod_red, mod_green, mod_blue)
    current_pixel_index += 1
    if current_pixel_index >= host_colors.Length then break end if
    current_pixel = host_colors[current_pixel_index]
  end while
  return host_colors
}
```

PSEUDOCODE for image retrieving:

```
decode_image(stego_pixels[], lsbCount)
  width = Infinity
  height = Infinity
  reserved_bytes = 4
  current_pixel_index = 0
  current_pixel = stego_pixels[current_pixel_index]
  current_byte_index = 0
  secret_data_bytes[4]
  num_bits_extracted = 0
  total_stego_bytes = 4
  binary = 0
  remaining_bits = 0
  num_zeros = Power(10, lsbCount)
  while current_byte_index < ((width * height) * num_channels + reserved_bytes) do
    current_pixel = stego_pixels[current_pixel_index]
    remaining_bits = (bit_max_size - num_bits_extracted)
                    n = 0
                    if remaining_bits >= lsbCount then
                            n = lsbCount
                    else
                            n = remaining_bits
```

```
                    end if
num_zeros = Power(10, num_bits_extracted)
binary = extractBinary(current_pixel.R, n, &num_bits_extracted) * num_zeros + binary
if num_bits_extracted >= bit_max_size then
   secret_data_bytes[current_byte_index] = todecimal(binary)
   if current_byte_index == 3 then
      dims[] = { secret_data_bytes[0], secret_data_bytes[1], secret_data_bytes[2], secret_data_bytes[3] }
      change_dimensions(dims, &width, &height)
      total_stego_bytes = width * height * num_channels + reserved_bytes
      secret_data_bytes[width * height * num_channels]
   end if
   current_byte_index += 1
   if current_byte_index >= total_stego_bytes then break end if

   num_bits_extracted = 0
   binary = 0
end if
remaining_bits = bit_max_size - num_bits_extracted
                    n = 0
                    if remaining_bits >= lsbCount then
                              n = lsbCount
                    else
                              n = remaining_bits
                    end if
num_zeros = Power(10, num_bits_extracted)
binary = extractBinary(current_pixel.G, n, &num_bits_extracted) * num_zeros + binary

if num_bits_extracted >= bit_max_size then
   if current_byte_index == 3 then
      dims[] = { secret_data_bytes[0], secret_data_bytes[1], secret_data_bytes[2], secret_data_bytes[3] }
      change_dimensions(dims, &width, &height)
      total_stego_bytes = width * height * num_channels + reserved_bytes
      secret_data_bytes[width * height * num_channels]
   end if
   current_byte_index += 1
   if current_byte_index >= total_stego_bytes then break end if

   num_bits_extracted = 0
   binary = 0
end if
remaining_bits = (bit_max_size - num_bits_extracted)
                    n = 0
                    if remaining_bits >= lsbCount then
                              n = lsbCount
                    else
                              n = remaining_bits
                    end if
num_zeros = Power(10, num_bits_extracted)
binary = extractBinary(current_pixel.B, n, &num_bits_extracted) * num_zeros + binary
if num_bits_extracted >= bit_max_size then
   if current_byte_index == 3 then
      dims[] = { secret_data_bytes[0], secret_data_bytes[1], secret_data_bytes[2], secret_data_bytes[3] }
      change_dimensions(dims, &width, &height)
      total_stego_bytes = width * height * num_channels + reserved_bytes
      secret_data_bytes[width * height * num_channels]
   end if
   current_byte_index += 1
   if current_byte_index >= total_stego_bytes then break end if
   num_bits_extracted = 0
```

```
        binary = 0
    end if
    current_pixel_index += 1
    if current_pixel_index >= stego_pixels.Length then break end if
  end while
  hidden_pixels[total_stego_bytes / num_channels)
  for i = 4 upto hidden_pixels.Length + reserved_bytes do
    hidden_pixels[i - reserved_bytes].R = secret_data_bytes[(i * num_channels) - reserved_bytes * 2]
    hidden_pixels[i - reserved_bytes].G = secret_data_bytes[(i * num_channels + 1) - reserved_bytes * 2]
    hidden_pixels[i - reserved_bytes].B = secret_data_bytes[(i * num_channels + 2) - reserved_bytes * 2]
  end for
          return hidden_pixels
```

PSEUDOCODE for appending bits:

```
appendBits(*colorBit, *dataBit, lsbCount, *b_written)
  if bit_max_size - *b_written < lsbCount then
                lsbCount = bit_max_size - *b_written
          end if
  mod = Power(10, lsbCount)
  *colorBit /= mod
  *colorBit *= mod
  *colorBit += *dataBit % mod
  *dataBit /= mod
  *b_written += lsbCount
```

PSEUDOCODE for extracting bits:

```
extractBinary(color, lsbCount, *bits_extracted)
          mod = Power(10, lsbCount)
  *bits_extracted += lsbCount
  return byte_to_binary(color) % mod
```

References

[1] G. Prashanti, K. Sandhyarani, "A New Approach for Data Hiding with LSB Steganography", Emerging ICT for Bridging the Future - Proceedings of the 49th Annual Convention of the Computer Society of India CSI, Springer 2015, pp. 423- 430.

[2] S. Goel, S. Gupta, N. Kaushik, "Image Steganography – Least Significant Bit with Multiple Progressions", Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA), Springer 2014-2015, pp. 105 112.

[3] B. Feng, W. Lu, and W. Sun, "Secure Binary Image Steganography Based on Minimizing the Distortion on the Texture", IEEE transactions on Information Forensics and Security, Feb. 2015.

[4] M. Nusrati, A. Hanani and R. Karimi, "Steganography in Image Segments Using Genetic Algorithm", 5th IEEE International Conference on Advanced Computing & Communication Technologies (ACCT), Feb 2015, pp. 102-107.

[5] K. Qazanfari and R. Safabakhsh, “A new Steganography Method which Preserves Histogram: Generalization of LSB++”, Elsevier International Journal of Information Sciences, Sept. 2014, pp. 90-101.

[6] P.U. Deshmukh and T.M. Pattewar, “A Novel Approach for Edge Adaptive Steganography on LSB Insertion Technique” IEEE International Conference on Information Communication and Embedded Systems (ICICES), Feb. 2014, pp. 1-5.

[7] M. R. Modi, S. Islam and P. Gupta, “Edge Based Steganography on Colored Images”, 9th International Conference on Intelligent Computing (ICIC), July 2013, pp. 593-600.

[8] D. Samidha and D. Agrawal, “Random Image Steganography in Spatial Domain” IEEE International Conference on Emerging Trends in VLSI, Embedded System, Nano Electronics and Telecommunication System (ICEVENT), Jan. 2013, pp. 1-3.

[9] S. Sachdeva and A. Kumar, “Colour Image Steganography Based on Modified Quantization Table”, Proceedings of IEEE 2nd International Conference on Advanced Computing & Communication Technologies, Jan. 2012, pp. 309-313.

[10] X. Qing, X. Jianquan and X. Yunhua., “A High Capacity Information Hiding Algorithm in Color Image”, Proceedings of 2nd IEEE International Conference on E Business and Information System Security, May 2010, pp. 1-4.

Plagiarism Report

22%