ABSTRACT

Confidential information is sent and received everywhere, but not all forms of transmission is secure even though the information is encrypted. There are covert methodologies used to send and receive data and one of the type is called as "Steganography". It is a Greek origin word which means "hiding writing". It is the practice of concealing a message within another message or a physical object. In computing/electronic contexts, a computer file, message, image, or video is concealed within another file, message, image, or video. There are different methods to perform Steganography such as LSB, DCT but along with that they come with disadvantages such as Visual Attack, Histogram Attack that lets the third person know that a secret information is hidden inside the image. In order to eliminate this, my project introduces a new approach for retrieving the secret message using the image by a concept called as 2-Bit RGBinary Mapping which uses the specific image as a reference and not the source of secret text itself, ensuring the confidentiality of the secret message.

CHAPTER 1

INTRODUCTION

The purpose of steganography is to conceal and deceive. It is a form of covert communication and can involve the use of any medium to hide messages. It's not a form of cryptography, because it doesn't involve scrambling data or using a key. Instead, it is a form of data hiding and can be executed in clever ways. Where cryptography is a science that largely enables privacy, steganography is a practice that enables secrecy – and deceit.

Discussions of steganography generally use terminology analogous to and consistent with conventional radio and communications technology. However, some terms appear specifically in software and are easily confused. These are the most relevant ones to digital steganographic systems:

The payload is the data covertly communicated. The carrier is the signal, stream, or data file that hides the payload, which differs from the channel, which typically means the type of input, such as a JPEG image. The resulting signal, stream, or data file with the encoded payload is sometimes called the package, stego file, or covert message. The proportion of bytes, samples, or other signal elements modified to encode the payload is called the encoding density and is typically expressed as a number between 0 and 1.

In a set of files, the files that are considered likely to contain a payload are suspects. A suspect identified through some type of statistical analysis can be referred to as a candidate.

1.2 PROPOSED METHOD:

Instead of hiding the message in the image itself we'll outsource the secret information in a secure server known as open-stego server that can be used by anyone. This server consists of mappings of the secret text to the image which is created using a technique called 2-Bit RGBinary mapping. This mapping can only be downloaded by the legitimate receiver from the open-stego server by using a unique token given by the server at the time of the mapfile upload and the secret message can only be retrieved from the same image which acts as the key.

ADVANTAGES:

- The information is not present in the image which makes it only as a reference for retrieval and not the source of secrecy.

- The Mapfile is stored in a secure server and can only be downloaded by the legitimate receiver who has the token.

- There will be no footprint of secret message in the image which opens the possibility to make use of public domain images.

1.3 EXISTING :

Algorithm such as LSB(Least Significant Bit) was mostly used to hide messages inside an image by replacing Least significant bit of image with the bits of message to be hidden. By modifying only the first most right bit of an image we can insert our secret message and it also make the picture unnoticeable, but if our message is too large it will start modifying the second right most bit and so on and a third party can notice the changes in picture. Another algorithm called DCT(Discrete Cosine Transformation) is used in steganography where image is broken into 8×8 blocks of pixels. Working from left to right, top to bottom, the DCT is applied to each block. Each block is compressed through quantization table to scale the DCT coefficients and message is embedded in DCT coefficients but when the secret message is bigger, the stuffing of those bits in the image won't be enough to conceal the whole information.

CHAPTER 2

SYSTEM REQUIREMENTS

2.1 HARDWARE REQUIREMENTS:

□ Processor   :   Intel Core i5 (or higher)

□ Speed   :   2.50 GHz

□ RAM   :   8 GB

□ Hard Disk   :   60 GB
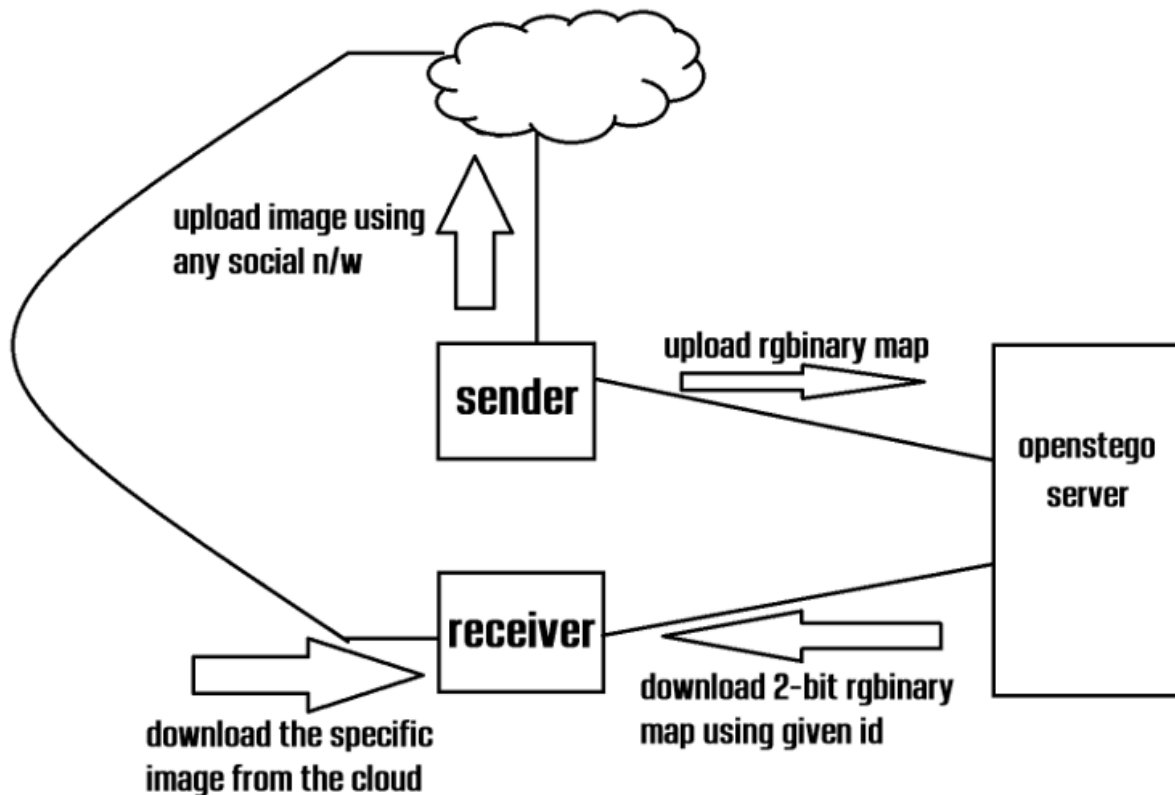
2.2 SOFTWARE REQUIREMENTS:

□ Operating System   :   Windows / Linux

□ Client-Side Language :   Python 3.8 (or higher)

□ Server-Side Language :   Nodejs 16.13.0 (or higher)

CHAPTER 3

SYSTEM DESIGN

3.1 Architecture diagram:

3.1.1. ARCHITECTURE DIAGRAM

WORKING

- The sender starts by creating the mapfile filled with secret message with the image using the 2Bit RGBinary algorithm and upload the rgbinary map file to the open-stego server resulting in receival of a token for receiving it later.

- Now using any social network platforms like Facebook, Instagram, Pinterest the image can be uploaded along with the token identifier can be shared using any other IM platforms.

- Next on the receiving side the image can be downloaded along with the token that is being shared by the sender. This token can be used to download the specific mapfile from the open-stego server later using them with a 2Bit extraction logic to retrieve the secret message.

3.2 FLOW DIAGRAM
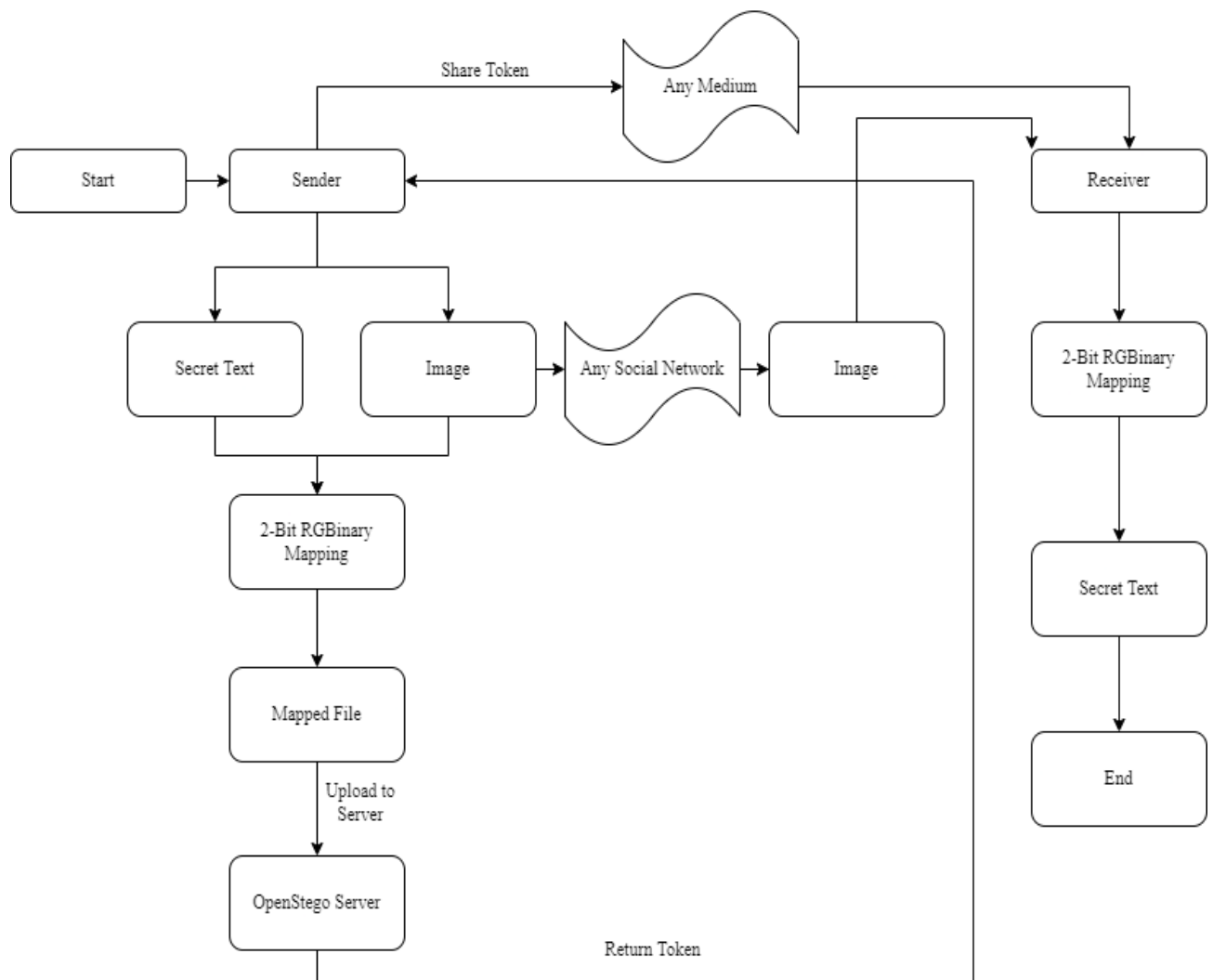
3.2.2. FLOW DIAGRAM

CHAPTER 4

MODULES

There are 2 main modules:

1. 2Bit RGBinary Mapping

2. Open-Stego Server

4.1. 2Bit RGBinary Mapping:

The full form of 2Bit RGBinary is 2Bit Red, Green, Blue Binary Mapping. The name proposed exactly defines what the algorithm does. Given an image as an input to this algorithm with a dimension of (m x n), it first creates a dictionary filling with an exhaustive index of all 2 Bit combinations (00, 01, 10, 11) present within the pixels (r, g, b) of the image. The runtime of this specific algorithm depends on the number of pixels in the image which makes the sender to decide whether an image with high/low resolution is to be selected for the stego process. Now let's name this exhaustive index dictionary as **D**. Next using **D**, the actual mapping of the secret message to the image pixels (in binary form) starts happening as shown in the following image:

```
nth pixel in image: (244 189 55) ▮
      r         g        b
    244       189       55
 11110100 10111101 110111
```

```
text to be mapped: pwd123
       p        w        d        1        2        3
 01110000 01110111 01100100 00000001 00000010 00000011
```

```
mapped output for the letter 'p':
n:r:4 n:b:0 n:r:6 n:r:6
```

```
NOTE: Here n is the pixel index denoted in one
dimension
```

4.1.1. 2Bit RGBinary Mapping Example

The secret text to be mapped is given as '**pwd123**'. We actually see a bunch of zero filled binary values before each secret characters but limited to 8 bits. The reason is that since we'll be mapping these messages to the image with maximum of 2bits each and if they're lesser than 8 bits, there'll be intricacies within the algorithm to retrieve the secret message. Now the mapped output as shown in the diagram is generalized as follows:

<pixel_index> : <colorspace> : <colorspace_bit_index>

- **pixel_index:** This value represents the nth pixel in the image i.e., if the image dimension is **mxn** (let **mxn** = **k**) the arrangement of these pixels will be structured in a 1-dimensional array making the index starting from **0** to **k**.

- **colorspace:** This represents whether the hidden bit is present in the **red** (r) or **green** (g) or **blue** (**b**) colorspace of the image.

- **colorspace_bit_index:** This represents within the given **colorspace**, at what index the secret data starts at. As mentioned already the 8 Bit limit, the value of this specific field will only be 0 or 2 or 4 or 6.

Now the output specific for the first letter in the secret message '**p**' and its binary equivalent is **01110000.** We assume that the image has only one pixel for this illustration purpose, and the **pixel_index** is **n**. Then we observe the first 2 bits of the secret text i.e. **01** in 'p'. Next we search for the occurrence of **01** in the image's binary. In this case it is present in the **red** colorspace starting at index **4** (assuming index starting at 0). So finally, we can now create the map for the first 2 bits as follows:

pixel_index = **n**, colorspace = **r**, colorspace_bit_index = **4**

The mapfile will have the data as '**n:r:4**', and this process will be repeated for all the secret message characters. Computationally this mapping will be created proactively creating the dictionary **D** as mentioned earlier that consist of the exhaustive 2Bit indices. Also to note that when selecting an index from dictionary **D** it'll be chosen random just to not make the mapping look something sequential.

Now on the receiving side, using the mapfile along with the image given as an input to the 2Bit extraction logic, the reverse process will be carried out i.e., using the output as shown previously '**n:r:4**' the algorithm searches for the **nth pixel_index** in the **red colorspace** at the **colorspace_bit_index** number **4** resulting in the value **'01'** to be retrieved and so on for the remaining hidden message.

4.2. Open-Stego Server:

A server dedicated to respond users with the 2-bit RGBinary Mapping (2BRGB) to the authorized user. There are 2 different roles for a user:

- **Sender**: Send a post request to the open-stego server 2BRGB mapping in plain text or even encrypted to the open-stego server. The server responds with a unique token that acts an identifier so that this could be sent to the receiver using any IM platforms for further retrieval of the same mapfile.

- **Receiver**: Using the unique token sent by the sender, the 2BRGB mapping can be downloaded by the receiver and further retrieve the secret message from the mapfile using the image.

The unique token is created in a way that it is universally distinct and there'll be no repetition of the same token for 2 different mapfile.

We name it as a uuid or **Universally Unique Identifier**. It relies on a combination of components to ensure uniqueness. uuids are constructed in a sequence of digits equal to 128 bits. The ID is in hexadecimal digits, meaning it uses the numbers 0 through 9 and letters A through F. The hexadecimal digits are grouped as 32 hexadecimal characters with four hyphens: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX. The number of characters per hyphen is 8-4-4-4-12. The last section of four, or the N position, indicates the format and encoding in either one to three bits.
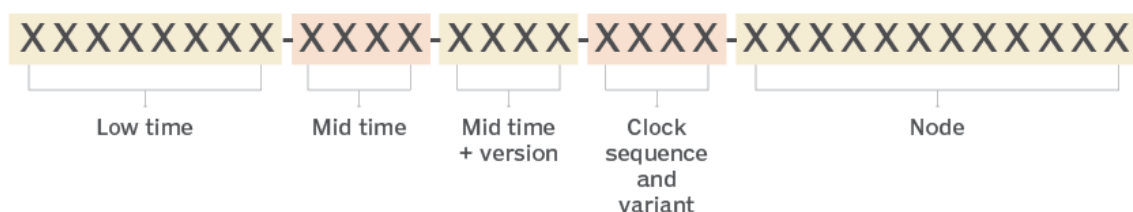
4.2.1. UUID Structure

As an example, uuids based around time have segments that are divided by hyphens that signify low, mid and mid time and version as different timestamps used to identify the UID. The digits under the last section, the node, denote the MAC address.

Since we'll be using nodejs for implementing the open-stego server we'll be using a package called **uuid**. It is a package that generates cryptographically strong unique identifiers with Node.js that doesn't require a large amount of code. The uuid npm package has zero

dependencies, and over thirty thousand packages depend on it, making it a safe choice when an id is needed that is guaranteed to be unique. It supports commonJS modules and also ECMAScript Modules, making it a good cross-platform choice. Besides generating a unique id, the uuid npm package has other utility methods available in its API that can be useful when working with unique identifiers, to ensure that they are valid.

The current variant of uuid, variant 1, consists of five different versions. These versions differ in how they are constructed. Types of uuids include:

Version 1. This version is generated from a specified time and node It is a time stamp-based unique host identifier.

Version 2. This version is generated similarly to version 1, however, less significant bits are replaced. Namely, eight bits of the clock sequence are replaced by a local domain number and 32 bits of the timestamp are replaced with the number for the specified local domain. These are reserved for DCE Security UUIDs.

Version 3. This version is generated by hashing both a namespace identifier and a name. Versions 3 and 5 are constructed similarly; however, version 3 uses message-digest algorithm 5 (MD5) as the hashing algorithm.

Version 4. This version of UUID is generated randomly. Although the random UUID uses random bytes, four bits are used to indicate version 4, while two to three bits are used to indicate the variant. These can be created using a random or pseudo-random number generator. More bits are used in this version, so there are fewer UUID combinations. However, there are still enough UUID combinations to avoid the possibility of a collision.

Version 5. Version 5 is generated the same way as version 3. However, it is generated using Secure Hash Algorithm 1, or SHA-1, as opposed to MD5, which version 3 uses for hashing. Versions 3 and 5 are well suited for use as unique identifiers for information and data within a namespace of a system.

For our Open-Stego Server we'll be using the Version 4 uuid due to as mentioned above, we still have enough uuid combinations to be used with no collisions.

CHAPTER- 5

IMPLEMENTATION

5.0. Things to configure before starting:

- **Client-Side:**
    - After installing Python, we need to install some libraries with pip using the following commands in the command line:
        - :$> **pip install Pillow**
        - :$> **pip install requests**

- **Server-Side:**
    - After installing Nodejs, we need to initialize the codebase and install dependencies as follows in the command line:
        - :$> **npm init**
          [Set the package name as **open-stego**]
          [Set the entry point as **open_stego**]
          [Leave other fields to default]
        - :$> **npm install express**
        - :$> **npm install multer**
        - :$> **npm install uuid**

    - Create a folder named '**uploads**' relative to the file location of **open-stego.js**

5.1. CLIENT-SIDE CODE (bit_match_steg.py):

```python
from PIL import Image, PngImagePlugin
import requests
import json
import random

PngImagePlugin.MAX_TEXT_CHUNK = 100 * (1024**2)
_2Bit_Dict = {"11" :[], "00" : [], "01" :[], "10" :[]}
_2BRGBINMAP = []
pix_vals = []
cs = ['r','g','b']
cs_map = {'r':0, 'g':1, 'b':2}

def to_binary(string):
    binary_string = ""
    for i in string:
        ascii_value = ord(i)
        binary_string += str(bin(ascii_value)[2:].zfill(8))
```

```python
    return binary_string

def to_string(binstring):
    string = ""
    for i in range(0, len(binstring), 8):
        ascii_value = int(binstring[i:i+8], 2)
        string += chr(ascii_value)
    return string

def send_map():
    mapfile_name = input("Enter map file name: ")
    file = {'map': open(mapfile_name, 'rb')}
    url = 'http://127.0.0.1:8080/sendmap'

    try:
        r = requests.post(url, files = file)
        status = r.status_code
        r = json.loads(r.text)
        if status == 200:
            print("\nMap file sent successfully\n")
            print("TOKEN RECEIVED FROM SERVER: ", r['token']+'\n')
    except:
        print("\nError occurred while uploading map\n")

def receive_map():
    token = input("Enter token: ")
    url = 'http://127.0.0.1:8080/receivemap'
    data = {'token': token}
    try:
        r = requests.post(url, data = data)
        status = r.status_code
        r = json.loads(r.text)
        if status == 200:
            with open(token+'.json', "w") as outfile:
                j_data = json.dumps(r)
                outfile.write(j_data)
            print("\nMap file received successfully\n")
        else:
            print("\nMap file not found\n")
    except:
        print("\nError occurred while receiving map file\n")

def find_2bit_matches(pix_vals):
    for pixel_index, pixel in enumerate(pix_vals):
        for j in range(len(pixel)):
            mapp(pixel_index, cs[j], pixel[j])
```

```python
def mapp(pixel_index, cs, pj):
    for i in range(0, len(pj), 2):
        _2Bit_Dict[pj[i:i+2]].append(str(pixel_index)+":"+cs+":"+str(i))

def map_string_to2bitdict(bin_secret_msg):
    for i in range(0, len(bin_secret_msg), 2):
        _2BRGBINMAP.append(random.choice(_2Bit_Dict[bin_se-
cret_msg[i:i+2]]))

def secret_bit_extraction(_2BRGBINMAP, pix_vals):
    secret_bits = ""
    for i in range(len(_2BRGBINMAP)):
        pixel_index, cs, cs_bit_index = _2BRGBINMAP[i].split(":")
        secret_bits += pix_vals[int(pixel_in-
dex)][cs_map[cs]][int(cs_bit_index):int(cs_bit_index)+2]
    return secret_bits

if __name__ == '__main__':
    while True:
        print('(1) Map secret text with image')
        print('(2) Upload mapfile to StegoServer')
        print('(3) Download mapfile from StegoServer')
        print('(4) Extract secret text from image using mapfile')
        print('(5) Exit')
        choice = input('Enter your choice: ')
        if choice == '1':
            image_name = input('Image name(with extension): ')
            secret_file_name = input('Secret text file name(with exten-
sion): ')

            im = Image.open(image_name, 'r')
            pix_vals = list(im.getdata())

            with open(secret_file_name, 'r') as f:
                message = f.read()
            message_in_bin = to_binary(message)

            for i in range(len(pix_vals)):
                r = bin(pix_vals[i][0])[2:].zfill(8)
                g = bin(pix_vals[i][1])[2:].zfill(8)
                b = bin(pix_vals[i][2])[2:].zfill(8)
                pix_vals[i] = [r, g, b]

            find_2bit_matches(pix_vals)
            map_string_to2bitdict(message_in_bin)

            with open("2BRGBINMAP.json", "w") as outfile:
                j_data = json.dumps(_2BRGBINMAP)
                outfile.write(j_data)
```

```python
        print('\nMap file saved as 2BRGBINMAP.json\n')

    if choice == '2':
        send_map()
    if choice == '3':
        receive_map()
    if choice == '4':
        filename = input('Enter map file name(with extension): ')
        image_name = input('Image name(with extension): ')

        im = Image.open(image_name, 'r')
        pix_vals = list(im.getdata())

        with open(filename, "r") as read_file:
            _2BRGBINMAP= json.load(read_file)

        for i in range(len(pix_vals)):
            r = bin(pix_vals[i][0])[2:].zfill(8)
            g = bin(pix_vals[i][1])[2:].zfill(8)
            b = bin(pix_vals[i][2])[2:].zfill(8)
            pix_vals[i] = [r, g, b]

        ext_secret_text = to_string(secret_bit_extraction(_2BRGBINMAP,
pix_vals))

        with open("extracted_secret.txt", "w") as outfile:
            j_data = json.dumps(ext_secret_text)
            outfile.write(j_data)
            print('\nSecret text extracted successfully & saved as ex-
tracted_secret.txt\n')
    elif choice == '5':
        exit()
```

5.2. SERVER-SIDE CODE (open_stego.js)

```javascript
const fs = require("fs")
const express = require("express")
const app = express()
app.use(express.urlencoded({ extended: true }))

const uuid = require("uuid")
var multer = require("multer")
var storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, __dirname + "/uploads")
  },
```

```javascript
    filename: function (req, file, cb) {
      console.log(file)
      cb(null, uuid.v4() + ".json")
    },
})
var upload = multer({ storage: storage })
const PORT = 8080

app.post("/sendmap", upload.single("map"), (req, res) => {
  console.log(req.file)
  if (req.file) {
    res.status(200).json({
      token: req.file.filename.split(".")[0]
    })
  } else {
    res.status(400)
  }
})

app.post("/receivemap", (req, res) => {
  token = req.body.token
  try {
    mapfile = fs.readFileSync(`./uploads/${token}.json`)
    mapfile = JSON.parse(mapfile)
    res.json(mapfile)
  } catch (err) {
    console.log(err)
  }
})

app.listen(PORT, () => console.log(`Server listening on PORT ${PORT}`))
```

5.3. OUTPUT

```
PS C:\Users\Sufyan\Documents\S PROS\MINE> python .\bit_match_steg.py
(1) Map secret text with image
(2) Upload mapfile to StegoServer
(3) Download mapfile from StegoServer
(4) Extract secret text from image using mapfile
(5) Exit
Enter your choice: 1
Image name(with extension): autumn.png
Secret text file name(with extension): secret.txt

Map file saved as 2BRGBINMAP.json
```

5.3.1. Mapping Secret Message to Image

5.3.2. Image used for Mapping (autumn.png)

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse porta a

In nec pharetra nisi, eu facilisis diam. Etiam rhoncus in purus hendrerit int

Etiam interdum metus ut nulla posuere, eu molestie est lobortis. Nullam et rh

Phasellus ante ante, posuere in auctor ac, maximus nec leo. Curabitur pharetr

Ut nulla massa, dapibus non tellus eu, laoreet commodo nisi. Pellentesque vel
```

5.3.3. Secret Message to map (secret.txt)

```
["4076:r:2", "11474:g:4", "16005:r:2", "11895:g:0", "19386:r:4",
 "18576:r:2", "15892:b:4", "10549:b:6", "10022:r:0", "9464:r:4",
 "14598:r:2", "6912:r:0", "8171:g:6", "5275:g:2", "12229:b:6",
 "17111:r:6", "9640:b:6", "3676:g:0", "13334:b:2", "21439:b:0",
 "6639:b:0", "6862:g:2", "11533:b:0", "10507:r:4", "9433:g:6",
 "4075:g:2", "19106:r:0", "22327:r:2", "21389:g:2", "3785:b:6",
 "15131:r:6", "13884:b:0", "16195:r:6", "19637:b:2", "19610:g:4",
 "9199:r:2", "21035:r:6", "11886:b:2", "13639:r:4", "373:r:6",
 "18137:b:4", "15745:b:2", "12217:b:6", "895:r:0", "15323:g:4",
 "18949:r:0", "21759:b:0", "2966:r:4", "9006:g:2", "7087:b:6",
 "21363:b:0", "1377:r:4", "8776:r:4", "19544:g:4", "11764:b:4",
 "8241:g:6", "4821:r:0", "383:g:6", "4093:r:4", "927:b:0",
 "7197:g:0", "21928:b:4", "16787:b:4", "2592:g:2", "8249:r:4",
 "21227:r:2", "12952:g:2", "14742:b:4", "4147:g:4", "913:r:0",
 "14735:r:2", "10500:g:4", "20467:r:4", "16002:r:0", "979:b:2",
 "10197:g:6", "928:b:0", "4417:g:0", "2313:r:4", "3105:g:6",
 "2031:g:0", "9821:g:6", "10771:g:6", "14228:b:4", "20306:b:0",
 "1027:r:0", "15403:r:6", "18514:r:2", "19616:g:2", "10013:b:6",
 "5252:g:2", "3231:g:6", "5483:g:6", "15268:g:2", "15083:g:2",
 "11588:b:2", "8364:g:6", "19094:b:6", "16376:g:6", "20138:g:0",
 "16932:b:2", "16701:g:4", "1280:r:2", "21878:b:2", "5535:g:6",
 "22177:b:2", "2127:g:4", "5409:b:6", "21616:b:0", "18278:b:6",
 "13831:b:0", "10211:b:0", "538:b:2", "12937:g:2", "15639:b:4",
 "10231:g:2", "1217:b:2", "10612:g:2", "13468:b:2", "5485:g:4",
 "15480:r:6", "14268:g:0", "5025:r:0", "16328:r:0", "9389:r:4",
 "20335:b:4", "4422:g:6", "15231:b:6", "14704:r:2", "5137:b:6",
```

5.3.4. Mapfile Contents (2BRGBINMAP.json)

```
PS C:\Users\Sufyan\Documents\S PROS\MINE> python .\bit_match_steg.py
(1) Map secret text with image
(2) Upload mapfile to StegoServer
(3) Download mapfile from StegoServer
(4) Extract secret text from image using mapfile
(5) Exit
Enter your choice: 2
Enter map file name: 2BRGBINMAP.json

Map file sent successfully

TOKEN RECEIVED FROM SERVER:  8127645f-4ffb-422b-9d6b-b5840273ade3
```

5.3.5. Uploading Mapfile to OSS and receive Token

[After this, the process of sharing the image and the token will be done using any chat or a social network platform as a medium]

```
PS C:\Users\Sufyan\Documents\S PROS\MINE> python .\bit_match_steg.py
(1) Map secret text with image
(2) Upload mapfile to StegoServer
(3) Download mapfile from StegoServer
(4) Extract secret text from image using mapfile
(5) Exit
Enter your choice: 3
Enter token: 8127645f-4ffb-422b-9d6b-b5840273ade3

Map file received successfully
```

5.3.6 Downloading Mapfile from OSS on the receiving side

```
PS C:\Users\Sufyan\Documents\S PROS\MINE> python .\bit_match_steg.py
(1) Map secret text with image
(2) Upload mapfile to StegoServer
(3) Download mapfile from StegoServer
(4) Extract secret text from image using mapfile
(5) Exit
Enter your choice: 4
Enter map file name(with extension): 8127645f-4ffb-422b-9d6b-b5840273ade3.json
Image name(with extension): autumn.png

Secret text extracted successfully & saved as extracted_secret.txt
```

5.3.7 Extracting secret message from the Mapfile

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse porta a

In nec pharetra nisi, eu facilisis diam. Etiam rhoncus in purus hendrerit int

Etiam interdum metus ut nulla posuere, eu molestie est lobortis. Nullam et rh

Phasellus ante ante, posuere in auctor ac, maximus nec leo. Curabitur pharetr

Ut nulla massa, dapibus non tellus eu, laoreet commodo nisi. Pellentesque vel
```

5.3.8. Extracted Secret Text

5.4 RESULTS:

- The sender successfully sends the image and token using any chat or social network platform after uploading the Mapfile to the OSS.

- The Mapfile is successfully saved in the OSS which can be retrieved using the token.

- The receiver successfully receives the image and the token on the other side and downloaded the Mapfile using the unique token.

- Retrieval of the secret message from the Mapfile using the image by the receiver is done successfully.