JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

A Different Approach to Image Steganography using 2BRGB Mapping

Ahmed M
Department of Computer Science and Engineering,
Dr. M.G.R. Educational and Research Institute,
Maduravoyal, Chennai

Dr. Cyril Raj V
Department of Computer Science and Engineering,
Dr. M.G.R. Educational and Research Institute,
Maduravoyal, Chennai

Dr. Geetha S
Department of Computer Science and Engineering,
Dr. M.G.R. Educational and Research Institute,
Maduravoyal, Chennai

***Abstract*— The transmission of confidential information is crucial in today's world. However, not all forms of transmission are secure, even if the information is encrypted. One covert methodology used for secure transmission is called steganography. This term, having its origins in Greek, means "hiding writing" and describes the process of hiding a message inside another. A computer file, message, picture, or video may be hidden inside another file, message, image, or video in a computing or electronic setting. However, some steganographic techniques such as LSB and DCT have disadvantages like Visual Attack and Histogram Attack that can reveal the presence of a hidden message. To address this issue, this project proposes a new approach for retrieving the secret message using an image, which we call 2-Bit RGBinary Mapping. This approach uses a specific reference image instead of the source of the secret text, ensuring the confidentiality of the message. By using this method, the hidden message can be retrieved securely without any fear of detection.**

Keywords*—LSB (Least Significant Bit), DCT (Discrete Cosine Transformation), 2BRGBM (2 Bit Red Green Blue Binary Mapping), JPEG (Joint Photographic Experts Group), OSS (Open Stego Server), IM (Instant Messaging), UUID (Universally Unique Identifier).

I. Introduction

Steganography's goal is to hide a data or to conceal something. It's a method of secret communication that uses every means possible to cover up interaction. Since no secret information is being used, this cannot be considered cryptography. However, this is only one method of concealing information; there are many more. In contrast to cryptography, which facilitates privacy via scientific means, steganography allows for concealment and deception by practical means. Standard radio and communications technology vocabulary is sometimes used while discussing steganography. Some terminologies however are unique to computer programs and are often mistaken with one another. In terms of digital steganographic systems, these are the most important ones: The data that is conveyed in secret is called the "payload." In contrast to the channel, which often refers to the sort of input, such a JPEG picture, the carrier is the signal, stream, or data file that conceals the payload. The resultant package, stego file, or covert message may be a signal, stream, or data file containing the encoded payload. The encoding density is a numeric value between 0 and 1 that describes the fraction of signal bytes, samples, or pieces that have been altered to encode the payload.[6]

Suspect files in a collection are those that have a higher probability of being the actual payload. A candidate is a potential suspect that has been found by statistical analysis.

II. Literature Survey

(E.V. Sidi et al., 2022) Although steganography and encryption are different, combining the two helps to ensure that sensitive data remains secure and that concealed conversations remain hidden. In this context, they have suggested a mashup of the LSB method, the RSA public-key cryptosystem, and the Caesar cipher symmetric encryption algorithm to create a new shortcut approach. The testing findings validate their assertion that the stego picture is visually indistinguishable from the original image and demonstrate the efficacy of their technique from a security perspective.[1]

L. Negi et al. (2021) The cell phone is quickly becoming one of the most widely used forms of communication today. Users attach a high degree of importance to the data they post on this platform. Therefore, protecting the message from eavesdroppers is crucial. This article proposes Steg!, a cryptographic steganographic security mechanism for Android. In this case, Advanced Encryption Standard (AES) is used as the cryptographic method while Least Significant Bit is employed as the steganographic technique. This dual method of encrypting the message and concealing it inside the picture raises the bar for keeping sensitive data safe from prying eyes. The program allows the user to conceal and reveal text superimposed on a picture. It has been shown that the aforementioned method is more effective and secure than systems that just use cryptography and steganography.[2]

(N. M. Abdali et al., 2020) This study provides a method for identifying spatial domain picture tampering such as image steganography, which is challenging to achieve due to the lack of a database of original images necessary for the classification process. The system works by first applying a high-pass filter with a threshold, and then deriving the auto-correlation function of the picture histogram. Without using the original picture, this method may determine if an image is a cover image or a stego image. The outcomes have shown that this method works. They believe that their findings, which have so far only been applied to least-significant-bit (LSB) steganography, may be generalized to other forms of image compression as well.[3]

(F. A. Rafrastara et al., 2019) Research into steganography, one type of data concealment, is constantly directed at enhancing picture quality, message payload, and security. One approach that may be utilized to improve the steganography's quality is the Inverted Least Significant Bit (LSB) method. This research suggests using an inverted LSB method with certain tweaks. The prior research used a two-bit LSB pattern on the second and third LSB when using the inverted LSB technique. Whether or whether the inverted LSB operation is carried out is determined by calculating the bit changes in each calculation pattern. The LSB three-bit pattern, where the pattern is derived from the second, third, and fourth bits of LSB, is advocated for usage in this study. The goal was to reduce the number of LSB swaps in the cover's individual pixels. It has been shown via testing that the inverted LSB approach using three LSB patterns outperforms the two-bit LSB pattern. The quality of the stego picture improved for eight of the 10 evaluated cover photos. The suggested LSB inverted approach is used in conjunction with other message encryption techniques based on chaotic maps.[5]

III. EXISTING SYSTEM

The LSB (Least Significant Bit) algorithm was often used to conceal messages inside a picture by swapping out the image's least significant bits for the secret information. Our secret message may be inserted into an image by altering its first rightmost bit, which also makes the image undetectable. However, if our message is too huge, it will begin altering the second rightmost bit, and so on, and a third party will be able to detect the changes.

Steganography makes use of another technique, DCT (Discrete Cosine Transformation), which divides the picture into 88 blocks of pixels. Blocks are DCT-ed in a left-to-right, bottom-to-top fashion. While the secret message is contained in the DCT coefficients and each block is compressed using a quantization table to scale the coefficients, the amount of information that can be hidden using this method decreases as the size of the secret message increases.

Also, since both these methods have a generalized clear view of how they work, their potential has reduced to be used out in the open for secret sharing and the reliance upon cryptography has become the facto standard for years.

IV. PROPOSED SYSTEM

Instead of hiding the message in the image itself we will 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 map file upload and the secret message can only be retrieved from the same image which acts as the key. Following are the advantages of the proposed system:

• 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. Also, there is a time interval for the mapfile to expire and the information is wiped off from the server to ensure that when physical tampering of the server if done it will ensure that the data will not be present.

• There will be no footprint or residue of secret message in the image which opens the possibility to make use of public domain images as this was a common problem between all the previous techniques of steganography where a suspected image when used against a scanning tool that when looked for matches with public domain

V. ARCHITECTURE DIAGRAM

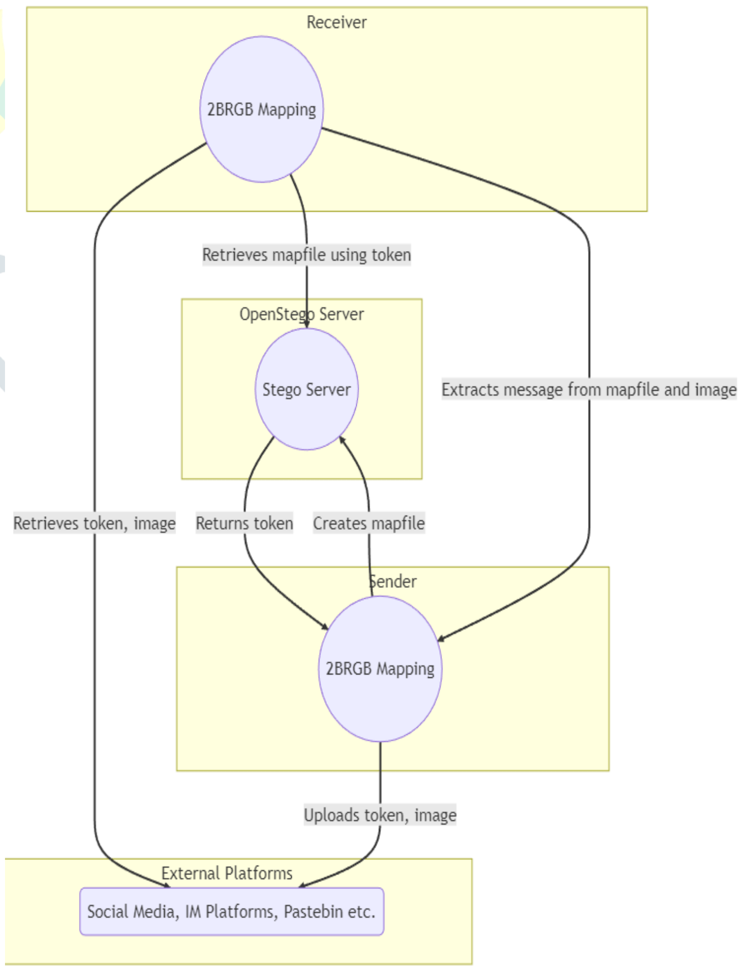


Figure 5.1 **Architecture Diagram**

VI. System 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.

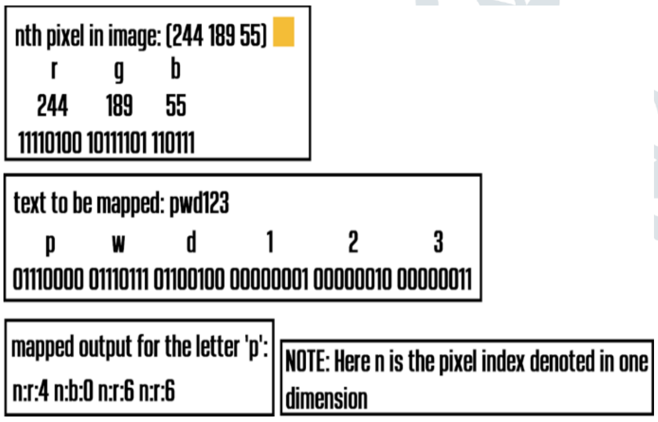
There are 2 modules:

A) **2-Bit RBGBinary Mapping**

B) **Open-Stego Server**

A. 2-Bit RBGBinary 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 (**mxn**), 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 us 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

Figure 6.A.1 **2Bit RGB Binary Illustration**

The secret text to be mapped is given as '**pwd123**'. We see a bunch of zero filled binary values before each secret character but limited to 8 bits. The reason is that since we will be mapping these messages to the image with maximum of 2 bits each and if they are lesser than 8 bits, there will 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 '**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 that consists of the exhaustive 2Bit indices. Also, note that when selecting an index from dictionary D, it will be chosen randomly just to avoid making the mapping look something sequential, which might result in brute forcing all the combinations.

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.

B. Open Stego Server

A server dedicated to respond users with the 2-bit RGBinary Map (2BRGB) file 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 parts to ensure uniqueness. UUIDs are built from a string of 128-bit numbers. The ID is written using the letters A through F and the numerals 0 through 9. The hexadecimal digits are organized as 32 hexadecimal characters with four hyphens: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX. Eight, four, four, twelve characters are used for every hyphen. Indicating the format and encoding in one to three bits, the N position is the fourth and final segment of the binary code.[7] Following image illustrates the structure of a UUID:

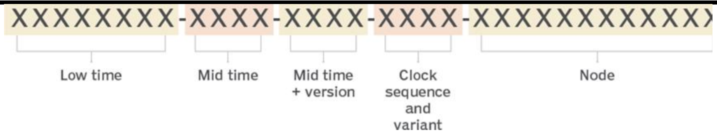

XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
Low time
Mid time
Mid time + version
Clock sequence and variant
Node

Figure 6.B.1 **UUID Structure**

Time-based UIDs, for instance, are composed of three distinct parts denoted by hyphens: a low time, a medium time, and a medium time and version. The node's MAC address is represented by the numbers in the last section.

We'll be utilizing a node.js package named uuid to create the open-stego server. It's a package for Node.js that creates secure cryptographically-signed IDs without a lot of extra code. The uuid npm package is recommended whenever a unique identifier is required since it has no dependencies and is relied on by more than 30,000 other packages. Thanks to its compatibility with both commonJS and ECMAScript Modules, it's an excellent option for use on several platforms. The uuid npm package does more than just generate a unique id; its API also includes utility methods for verifying the integrity of unique identifiers.

There are five distinct implementations of uuid in its present iteration, known as "variant 1." The structures of these variants are different from one another. Various versions 1, 2, 3, 4, and 5 of uuids exist.

VII. IMPLEMENTATION

The client-side program (script used by the sender and the receiver) uses the Pillow library for image processing and the requests library to interact with the open stego server. Now, following is the algorithm of **2 Bit Red Green Blue Binary Mapping**, and explanation of what each one of those functions do and result of their Computational Time Complexity Analysis.

```
# Contains the cache of all indexes of 2 bit Paired binaries from the given image
_2Bit_Dict = {11 :[], 00 : [], 01 :[], 10 :[]}

# Output mapfile in the form of a dictionary (later saved as a map file)
_2BRGBINMAP = []

# Array of arrays of RGB values of the given image in 1 Dimension
pix_vals = []

# 2BRGBM specific dict & array for cached value retrieval
cs = ['r','g','b']
cs_map = {'r':0, 'g':1, 'b':2}

# To convert given string to binary
to_binary(string):
    for i in string:
      binary_string+=str(bin(ascii(i))[2:].0fill_as_prefix(8))
    return binary_string

# To convert binary string to characters
to_string(binstring):
    for i in range(binstring[i:i+8], 2):
        string += character(binstring[i:i+8])
    return string

# Find the 2Bit match with the secret text
find_2bit_matches(pix_vals):
    for pixel_index, pixel_i in enum(pix_vals):
        for i in range(len(pixel_i)):
            for j in range(0, len(pixel_i), 2):
                _2Bit_Dict[pixel_i[j:j+2]].append(str(pixel_index:{cs}:str(j))

# Maps each 2-bit sequence to a randomly selected 2-bit value from the _2Bit_Dict
map_string_to2bitdict(bin_secret_msg):
    for i in range(0, len(bin_secret_msg), 2):
      chosen = random_pick(_2Bit_Dict[bin_secret_msg[i:i+2]])
      _2BRGBINMAP.append(chosen)

# Reversal algorithm of 2BRGB Mapping to extract the secret with the mapfile and the image
secret_bit_extraction(_2BRGBINMAP, pix_vals):
    secret_bits = ""
    for i in range(len(_2BRGBINMAP)):
      pixel_index,cs,csbi=_2BRGBINMAP[i].split(":")
      secret_bits+=pix_vals[int(pixel_index)][cs_map[cs]][int(csbi):int(csbi)+2]
    return secret_bits
```

Analysis wise:

to_binary function is $O(m)$, where m is the length of the input string.

to_string function is $O(m)$, where m is the length of the input binary string.

find_2bit_matches function is $O(n \cdot k)$, where n is the number of pixels in the image, and k is the number of color channels ($k = 3$ for RGB images), so further it can be approximated as $O(n)$.

map_string_to2bitdict function is $O\left(\frac{m}{2}\right)$, where m is the length of the binary message.

secret_bit_extraction function is $O(m)$, where m is the length of the secret text that is in the form of 2-Bit RGBinary Mapfile.

Overall, the complexity of the algorithm is:

$$3O(m) + 2O(n) + O\left(\frac{m}{2}\right)$$

where n is the number of pixels in the image, m is the length of the binary message.

Following are the detailed explanations of what each function does in 2Bit RGB Mapping Algorithm:

- **to_binary(string):** This function takes in a string as an argument and returns its binary representation. It achieves this by first iterating over each character in the input string and finding its corresponding ASCII value. It then converts each ASCII value to its binary representation using the bin() function and concatenates them all together to form the final binary string. For example, the string "hello" would be converted to the binary string "0110100001"…

- **to_string(binstring):** This function takes in a binary string as an argument and returns its string representation. It achieves this by iterating over the binary string in chunks of 8 bits, converting each chunk to its corresponding ASCII value using the int() function, and concatenating them all together to form the final string. For example, the binary string

"0110100001"... would be converted to the string "hello".

- **find_2bit_matches(pix_vals):** This function takes in a list of pixel values as an argument and populates a global dictionary _2Bit_Dict with pixel indices that contain each possible 2-bit combination. It achieves this by iterating over each pixel in the input list and for each pixel, iterating over its red, green, and blue color values (in that order). It then divides each color value into non-overlapping 2-bit chunks and calls the mapp() function on each chunk to update the _2Bit_Dict dictionary. The resulting _2Bit_Dict dictionary contains four keys ("00", "01", "10", "11"), with each key containing a list of pixel indices in the format "<**pixel_index**>:<**color**>:<**color_bit_index**>", where "<**pixel_index**>" is the index of the pixel in pix_vals, "<**color**>" is either "r", "g", or "b" depending on the color being examined, and "<**color_bit_index**>" is the starting bit index of the 2-bit chunk being examined.

- **mapp(pixel_index, cs, pj):** This function takes in a pixel index, a color string, and a 2-bit chunk string as arguments and updates the global _2Bit_Dict dictionary accordingly. It achieves this by using the pixel index and color string to construct a unique key in the dictionary and then appending the key to the value list corresponding to the 2-bit chunk string in the dictionary.

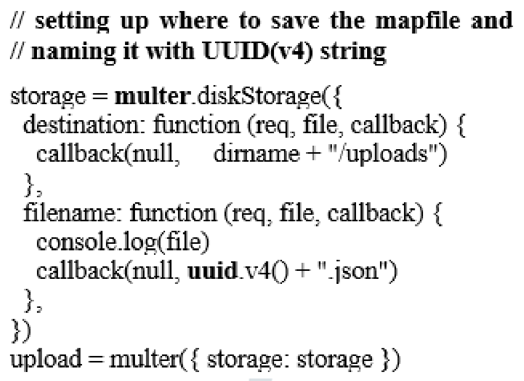
- **map_string_to2btdct(bin_secret_msg):** This function takes the bin_secret_msg which is the secret message that has been converted to binary format and maps each 2-bit sequence to a randomly selected 2-bit value from the _2Bit_Dict dictionary. It does this by iterating over the bin_secret_msg in steps of 2 (since each 2-bit sequence needs to be mapped), and for each 2-bit sequence, it gets the corresponding 2-bit value from _2Bit_Dict and adds it to the _2BRGBINMAP list.

- **secret_bit_extraction(_2BRGBINMAP, pix_vals):** This function takes in the _2BRGBINMAP list which has the 2-bit values mapped to the secret message and pix_vals which is the list of RGB pixel values of the image. It then extracts the secret message from the image by iterating over each element of _2BRGBINMAP and extracting the corresponding bits from the appropriate pixel of the image. To do this, it first splits the _2BRGBINMAP element into three parts - pixel_index, cs and cs_bit_index - which correspond to the index of the pixel in the pix_vals list, the color channel of the pixel ('r', 'g' or 'b') and the starting bit index of the 2-bit sequence in the color channel, respectively. It then uses these values to extract the 2-bit sequence from the pixel's color channel and appends it to the secret_bits string. Once all the elements of _2BRGBINMAP have been processed, the function returns the extracted secret message.

The **Open Stego Server** uses the Express.js library to handle HTTP requests and Multer middleware for file handling. Following are the dependencies for creating and running the server:

- **fs**: This is a Node.js built-in module that provides an API for interacting with the file system. In this code, it is used to read the contents of a file.
- **express**: This is a popular Node.js web application framework used for building web applications and APIs. It provides a set of features for handling HTTP requests, routing, middleware, and more.
- **uuid**: This is a module used for generating unique identifiers (UUIDs). It provides a method for generating a random UUID version 4.
- **multer**: This is a piece of middleware meant to manage file uploads through multipart/form-data requests. It provides a way to specify the destination directory and filename for uploaded files.

In code, these modules are imported using **require()** function, which is used to load and import external modules in Node.js. Following is the implementation used for the Open Stego Server and explanation of what each API endpoint does.



```
// setting up where to save the mapfile and
// naming it with UUID(v4) string

storage = multer.diskStorage({
  destination: function (req, file, callback) {
    callback(null,    dirname + "/uploads")
  },
  filename: function (req, file, callback) {
    console.log(file)
    callback(null, uuid.v4() + ".json")
  },
})
upload = multer({ storage: storage })
```

```
// end point to store requested incoming    //    map   file from sender
route.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)
  }
})
```

```
// endpoint to get back the mapfile with a  // token by the receiver
route.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)
}})
```

Following are the detailed explanation of what each API endpoint does in the Open Stego Server:

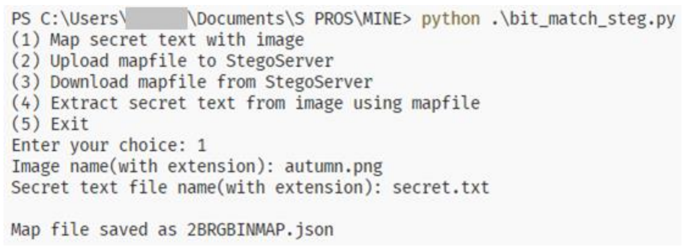
- **/sendmap** route expects a multipart/form-data request with a file named map attached. When a file is received, the server generates a unique identifier using the uuid library, saves the file in the server's disk storage under a folder named uploads(to be added manually on the server side), and returns the token(identifier) in the response. The receiver can later use this token to retrieve the mapfile.

- **/receivemap** route expects a POST request with a token field in the request body. The server reads the file with the name of the token from the uploads folder, parses it as a JSON object, and returns it in the response to the legitimate receiver.

VIII. RESULT

The program provides a menu-driven interface on the client-side to map secret text to an image, upload the map to a server, download the map from the server and extract the secret text from an image using a downloaded map.

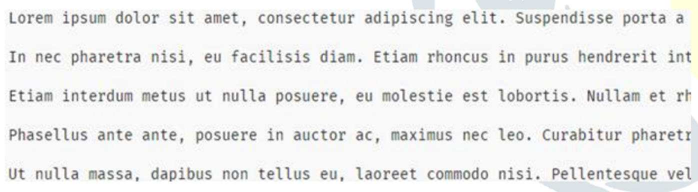
Following PoC shows the steps involved in the proposed method:



```
PS C:\Users\[redacted]\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
```

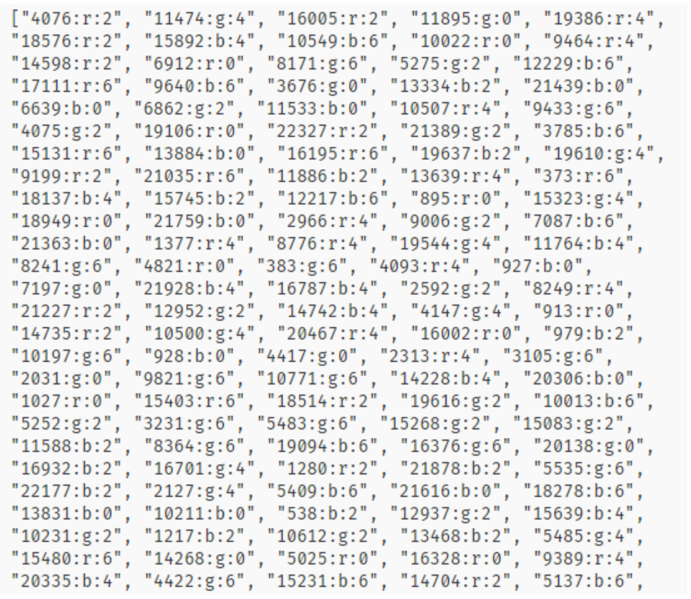
Figure 8.1 **Mapping Secret Message to Image**

Figure 8.2 **Image used for Mapping - autumn.png**



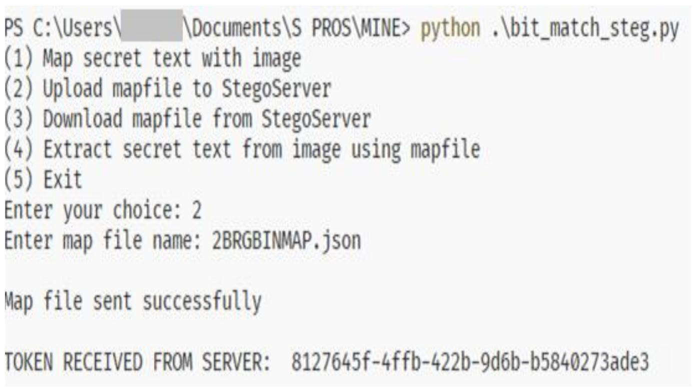
```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse porta a

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
```

Figure 8.3 **Secret Message to map – secret.txt**



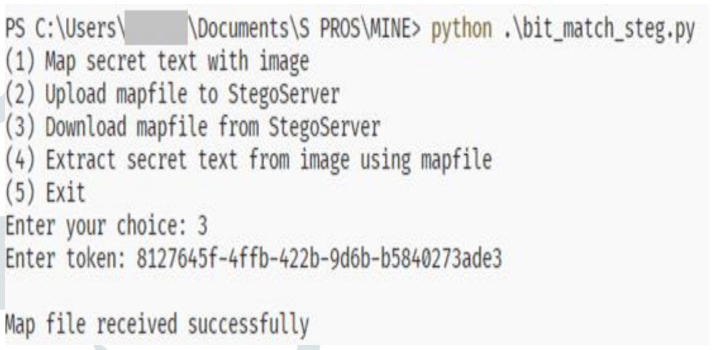
```
["4076:r:2", "11474:g:4", "16005:r:2", "11895:g:0", "19386:r:4",
"18576:r:2", "15892:b:4", "10549:b:6", "10022:r:0", "9464:r:4",
"14598:r:2", "6912:r:0", "8171:g:6", "5275:g:2", "12229:b:6",
"17111:r:6", "9640:b:6", "3676:g:0", "13334:b:2", "21439:b:0",
"6639:b:0", "6862:g:2", "11533:b:0", "10507:r:4", "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",
```

Figure 8.4 **Mapfile Contents (2BRGBINMAP.json)**



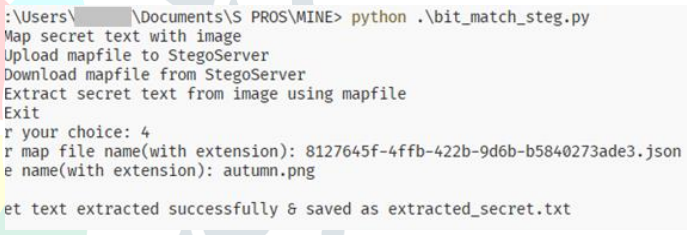
```
PS C:\Users\[redacted]\Documents\S PROS\MINE> python .\bit_match_steg.py
(1) Map secret text with image
(2) Upload mapfile to StegoServer
(3) Download mapfile from StegoServer
(4) Extract secret text from image using mapfile
(5) Exit
Enter your choice: 2
Enter map file name: 2BRGBINMAP.json

Map file sent successfully

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

Figure 8.5 **Upload Mapfile to OSS & receive Token**



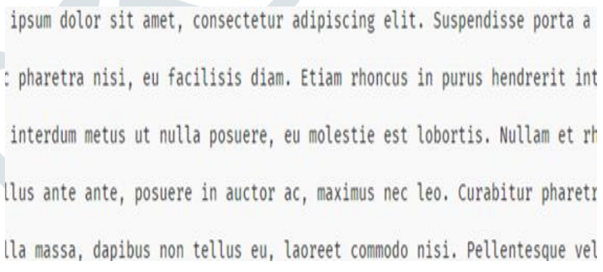
```
PS C:\Users\[redacted]\Documents\S PROS\MINE> python .\bit_match_steg.py
(1) Map secret text with image
(2) Upload mapfile to StegoServer
(3) Download mapfile from StegoServer
(4) Extract secret text from image using mapfile
(5) Exit
Enter your choice: 3
Enter token: 8127645f-4ffb-422b-9d6b-b5840273ade3

Map file received successfully
```

Figure 8.6 **Download Mapfile from OSS by receiver**



```
:\Users\[redacted]\Documents\S PROS\MINE> python .\bit_match_steg.py
Map secret text with image
Upload mapfile to StegoServer
Download mapfile from StegoServer
Extract secret text from image using mapfile
Exit
r your choice: 4
r map file name(with extension): 8127645f-4ffb-422b-9d6b-b5840273ade3.json
e name(with extension): autumn.png

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

Figure 8.7 **Extract secret message from the Mapfile**



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

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

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

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

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

Figure 8.8 **Final Extracted Secret Text**

IX. SCOPE FOR FUTURE DEVELOPMENT

The benefits and drawbacks of each innovation are never completely lopsided. The needs of the project have been met nearly entirely. For a particular plaintext, the encoding produced by the 2BRGBM technique has a high file size, which may be avoided by using any standard compression algorithm, therefore there is room for additional requirements and improvements in the mapfile generating process. Also, a chain of stego images can be used to hide the message (in our case, use images as multiple references to retrieve the secret text)

X. CONCLUSION

This proposed methodology for secretly sharing messages in a covert manner can be achieved in an effective way if we try to use self-created images and not an image from a public domain. It still can be used but, if the token is not taken care of and it is being seen by a third person, it becomes obvious that it should be an identifier to retrieve the mapfile from the OSS and further mapping can be done by them and could retrieve the secret making this method useless. Also, when using self-created images like a selfie or a landscape photo of any kind, people around the internet will not mind it as no suspicion takes place since someone does not upload a photo from a public domain and place it in their social media account. But excluding the intricacies mentioned so far, this method extensively increases current standards of confidentiality of secret message when shared in an open platform since we use it only as a reference of the secret data and not the source of secrecy.

XI. REFERENCES

[1] E. V. Sidi, I. Diop and K. Tall, "A New hybrid approach of Data Hiding Using LSB Steganography and Caesar cipher and RSA algorithm (S-ccr)," 2022 International Conference on Computer Communication and Informatics (ICCCI), Coimbatore, India, 2022, pp. 1-4, doi: 10.1109/ICCCI54379.2022.9740979.

[2] L. Negi and L. Negi, "Image Steganography Using Steg with AES and LSB," 2021 IEEE 7th International Conference on Computing, Engineering and Design (ICCED), Sukabumi, Indonesia, 2021, pp. 1-6, doi: 10.1109/ICCED53389.2021.9664834.

[3] N. M. Abdali and Z. M. Hussain, "Reference-free Detection of LSB Steganography Using Histogram Analysis," 2020 30th International Telecommunication Networks and Applications Conference (ITNAC), Melbourne, VIC, Australia, 2020, pp. 1-7, doi: 10.1109/ITNAC50341.2020.9315037.

[4] M. Baziyad and M. S. Obaidat, "On the Importance of the DCT Phase for Image Steganography Schemes," 2020 IEEE 5th International Conference on Computing Communication and Automation (ICCCA), Greater Noida, India, 2020, pp. 791-795, doi: 10.1109/ICCCA49541.2020.9250849.

[5] F. A. Rafrastara, R. Prahasiwi, D. R. Ignatius Moses Setiadi, E. H. Rachmawanto and C. A. Sari, "Image Steganography using Inverted LSB based on 2nd, 3rd and 4th LSB pattern," 2019 International Conference on Information and Communications Technology (ICOIACT), Yogyakarta, Indonesia, 2019, pp. 179-184, doi: 10.1109/ICOIACT46704.2019.8938503.

[6] Stanger, James. “The Ancient Practice of Steganography: What Is It, How Is It Used and Why Do Cybersecurity Pros Need to Understand It?” Default. CompTIA, December 19, 2022. https://www.comptia.org/blog/what-is-steganography.

[7] Gillis, Alexander S. “What Is UUID?” App Architecture. TechTarget, August 31, 2021. https://www.techtarget.com/searchapparchitecture/definition/UUID-Universal-Unique-Identifier.