BCSE410L – Cyber Security

STEGANOGRAPHY IN CYBERSECURITY

21BCE2405 TORAN V ATHANI

Under the Supervision of

SAIRABANU J

Associate Professor Sr.

School of Computer Science and Engineering (SCOPE)

B.Tech.

in

Computer Science and Engineering

School of Computer Science and Engineering

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

November 2024

ABSTRACT

Steganography is a technique that hides words in ordinary documents, making them invisible to observers. This project explores the principles and practical applications of steganography, focusing on two popular methods: text steganography and image steganography.

This project uses Python, OpenCV, and least significant bit (LSB) technology to display text and images in other images in order to protect sensitive information. Unlike encryption, which makes information unreadable, steganography adds an extra layer of security by masking the presence of words.

This approach has important applications in areas where determinism and security are important, such as secure communications and digital forensics. This report provides step-by-step instructions for implementing these techniques, visualizing the results, and discussing recent developments in steganography techniques.

The results demonstrate the effectiveness of steganography in protecting confidential information from eavesdropping and highlight its advantages over traditional encryption methods.

The report provides detailed instructions on how to use text and image steganography, along with screenshots and visual comparisons of encoding and decoding methods. We also discuss recent developments in steganography, including AI-based techniques that increase data encryption capabilities and security. These innovations have made steganography an important tool in the digital age, and its applications are expanding to include areas such as information security, security law, and digital governance.

Keywords - Steganography, data concealment, information security, text steganography, image steganography, Least Significant Bit (LSB), OpenCV, Python Imaging Library (PIL), covert communication, digital watermarking, cyber forensics, data encoding, secure communication, anti-detection mechanisms.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Overview of Steganography

Steganography is a method of hiding confidential information in hidden media (such as images, audio files, or text) to prevent detection. Unlike encryption, which scrambles information beyond recognition, steganography hides information in a masked medium, making it appear harmless and invisible to the human eye or automatic detection systems. The word "steganography" is derived from the Greek words steganos, meaning "to cover" or "to hide," and graphia, meaning "to write." Sensitive information is protected from unauthorized access. For example, text in digital images can be encoded to the smallest possible pixel value using a method known as least significant bit (LSB) steganography. By manipulating the image in a way that does not visually alter its appearance, confidential information is protected and can only be recovered using appropriate techniques.

It avoids paying attention to the message because the news cover remains unchanged for the observer. In areas where encryption is illegal or strictly regulated, steganography provides a covert means of sending sensitive information. Hidden in other images. Using tools such as OpenCV and Python Imaging Library (PIL), the project demonstrates the use of these technologies and their ability to store and transmit confidential information.

1.2 Relevance of Steganography in Data Protection

Steganography plays an important role in today’s data protection strategies, providing an effective way to hide sensitive data as well as its secure transmission and storage. In an increasingly digital world where data privacy and security are important, steganography can be used as a complementary technology to traditional encryption to add an extra layer of known confidentiality to communications. Here are a few important reasons why this is particularly important:

1. Data hiding: Steganography allows sensitive data to be hidden within seemingly useless data (such as images, audio files or text), making it more difficult for unauthorized persons or automated systems to detect confidential information. This is especially true in situations where data encryption alone would be suspicious, arouse suspicion or trigger scrutiny.

2. Covert communication: In situations where encryption is illegal or heavily monitored (e.g. in administrative or business management), steganography offers an alternative. It enables confidential communication and unambiguous exchange of information by storing information in static files.

3. Complementary cryptography: While cryptography protects information by converting it into an unreadable format, steganography goes a step further by hiding the fact that there is a secret message. This combination provides security by minimizing the possibility of detection or interception during transmission.

4. Secure digital watermarking: Steganography is often used in digital watermarking, where digital media owners place digital watermarks on image, audio, and video files to protect their intellectual property. These watermarks provide proof of ownership and help track and prevent illegal use or distribution of digital assets.

5. Preventing data loss or corruption: Steganography can be used as a method to store valuable information in a medium that is prone to data corruption or loss. Confidential data can be stored in other file formats and restored if necessary to provide access to or secure the original data.

6. Steganalysis Resistance: Advanced steganography techniques are designed to make it difficult for steganolysis tools (tools designed to analyze steganographic content) to identify hidden messages. This attack is important in protecting sensitive data from detection algorithms used by attackers. It strengthens messages, enhances encryption, and provides covert communication and intelligence protection.

As digital security threats continue to evolve, the role of steganography in data protection has become increasingly important.

2. DETAILED DESCRIPTION OF STEGANOGRAPHY

2.1 Definition and Types

Steganography is the art and science of hiding secret information in non-secret media (such as images, audio files, or text) in a way that prevents detection. Unlike cryptography, which prevents information from changing its appearance, steganography hides secrets within ordinary information, making it invisible to the human eye or detection. Secret data can only be extracted using special methods or keys. It is often used in conjunction with encryption technology, where data is first encrypted and then hidden to add an extra layer of security.

Types of Steganography :

There are several types of steganography, each designed for different types of media:

- **Text Steganography**
  Involves hiding secret information within text. Techniques include the manipulation of spaces between words, changing font styles, or embedding invisible characters. Although text-based steganography is simple to implement, it often offers limited data capacity.

- **Image Steganography**
  One of the most popular forms of steganography, this technique hides data within digital images. The most common approach is Least Significant Bit (LSB) steganography, where the least significant bit of the image's pixel values is altered to encode the secret message. Image steganography offers a large storage capacity and minimal distortion to the image.

- **Audio Steganography**
  This technique hides data within audio files, such as MP3 or WAV files. Techniques like phase coding or echo hiding can be used to embed information without perceptible changes to the sound. Audio steganography can provide more capacity compared to image steganography, but is more challenging due to the potential for distortion.

- **Video Steganography**
  Video files can be used to hide data by modifying individual frames or audio tracks within the video. This method offers a very large capacity for embedding hidden messages, as it combines the space of both images and audio. However, it may require more sophisticated techniques to prevent noticeable changes.

- **Network Steganography**
  Hides information within network protocols, such as the headers of IP packets. This form of steganography exploits the characteristics of network traffic and is often used for covert communication over the internet.

2.2 Steganography vs. Cryptography

Steganography and cryptography are two methods used to protect information, but they work and operate differently.

Cryptography focuses on converting readable data into unreadable formats to maintain confidentiality. It uses algorithms and encryption keys to scramble data, ensuring that only authorized users with the decryption key can access the original data. However, despite cryptography's strong encryption methods, there are still encrypted devices that can be seen. This means that anyone who intercepts the data will know that some kind of encrypted message has been sent, which can be shocking.

Instead of making information unreadable, steganography embeds hidden information in media (such as images, audio, or text) to reveal hidden information that cannot be detected by the human eye or automatic detection equipment. The main advantage of steganography is that it can be used to send information without marking the message someone is sending. While cryptography protects information by making it difficult to understand, steganography prevents information from being seen in the first place. Writing hides information so that no one even knows it is there. These two technologies can be used together to provide an additional layer of security, where the encryption tool encrypts the object before using steganography to hide it, so that the content and existence of the object are well protected.

However, both methods have their limitations. Cryptography can provide better security because the encryption process makes the data unreadable to anyone without the key, but it does not hide the fact that the data is being transmitted. While steganography is very good at hiding the existence of messages, it often has limitations in terms of capabilities. The amount of information that can be hidden in the overlay medium (such as images) is usually limited, and excessive changes to the overlay medium can make it look bad or damage the hidden text. However, the ability to be undetected with steganography is often more important than the information that can be hidden news.

Advanced algorithms have been developed to detect even the smallest changes in data coverage, and advances in machine learning and statistical analysis have made it increasingly difficult to process hidden text.

However, steganography technology continues to evolve, and new methods are being developed to make hidden information more difficult to discover, depending on the specific needs of users. Cryptography is necessary in situations where information must be securely encrypted to prevent unauthorized access, even if the information is known. However, steganography is ideal for confidential communication, where the main concern is to hide the existence of the information. Together, these two methods provide strong data protection that ensures confidentiality and privacy.

2.3 Applications

Steganography has many applications, especially in the areas of information security transmission and protection. The prevention of sensitive information is very important. From secure communication to digital watermarks, technology that hides information in plain sight has proven useful in both legal and illegal environments. Here are some important places where steganography is used:

- One of the most popular forms of steganography is **Secret Communication**. In an environment where privacy is important and encryption can lead to surprises (such as administrative systems or sensitive communications), steganography allows people to send secret messages without exposing their lives to third parties. This is especially true in situations where encryption is illegal or prohibited, and allows individuals to avoid detection when sending sensitive information. Because steganography hides information within seemingly innocuous information (images, sounds, etc.), it cannot be discovered by someone who does not know how to use the message. This invisibility makes steganography an important tool for journalists, activists, and others who want to protect their communications from surveillance.

- Steganography plays an important role in **Digital Watermarking**, which is used to mark invisible information on digital media such as images, video, and audio files. Content creators often use watermarks to claim ownership of their work or to track illegal distribution. This form of steganography can embed identifiers, personal information, or tracking information that are hidden from prying eyes but visible to authorized users. Digital watermarks are widely used in the media industry to protect intellectual property, prevent copyright infringement, and ensure that digital content is attributed to its creator.

- In the field of **Intellectual Property Protection**, steganography is a powerful tool for protecting digital content from unauthorized legitimate use. For example, an artist or video producer may place a hidden watermark or identifier on a digital file to claim ownership of their work or track its distribution. This is especially important in industries like music, video, and advertising, where the protection of digital assets is critical. By placing invisible labels on documents, creators can prove their ownership in the event of unauthorized copying or distribution, making it easier to track down the site of copyright infringement. Data Integrity and Authentication**

- Steganography can also be used to improve **Data Integrity and Authentication** in digital data. In this case, a password or code placed on the form can be used to verify the accuracy of the data or to ensure that it has not been tampered with. For example, organizations can place hidden signatures or timestamps on digital contracts to verify their integrity and ensure that they have not been altered since the initial signature. This technology helps prevent fraud and ensures that the original documents remain intact, which is important for legal and financial reasons.

- **Digital Forensics** is another area where steganography is used, particularly in cybercrime investigations. Law enforcement and forensic investigators use steganalysis techniques to identify information hidden in digital files that may be relevant to criminal investigations. For example, during a forensic investigation of a computer or digital device, hidden messages or embedded malware that uses steganography may be discovered. Analyzing steganographic content can provide valuable evidence in cases such as identity theft, unauthorized communication, or malware infections.

- While steganography is typically used for protection and legal purposes, it has also entered the world of **Cybersecurity Threats**. Steganography is often used by criminals to hide malware, spyware, or other types of crimes in seemingly innocuous information, such as images or documents. By placing malicious payloads in seemingly innocuous files, attackers can be thwarted by antivirus programs and other security measures. Steganography allows malware to bypass filters and be distributed undetected, often through email links or online file sharing. The practice highlights the need for powerful detection that can detect hidden threats in digital data.

- Steganography is also used **to hide information** in public spaces, such as social media platforms or websites where information is disseminated. Sensitive text warnings may be issued. For example, secret messages can be hidden in photos or videos shared on social media platforms. Since these images are distributed in many ways, they provide an ideal cover for embedding secret information that the intended recipient can extract using appropriate techniques. This method is used in surveillance or covert operations that require secure and invisible communication.

- Steganography is increasingly used in **Cloud Storage and Backup Systems** as a secure way to store sensitive data. By embedding encrypted data in seemingly innocent data stored in the cloud, users can hide their data even if the storage service is compromised. In this case, steganography not only helps prevent unauthorized access to information, but also allows information to be stored even if an attacker breaks into the storage system. This method provides an additional layer of security for sensitive information online.

From communication security and asset protection to digital forensics and cybersecurity, the ability to hide information in plain sight makes the protection of everyday information an important part of the strategy. However, as the use of steganography continues to spread, the need for advanced detection, especially to combat malicious uses such as malware distribution, also increases. As steganography technologies develop, their applications in many areas will continue to expand and offer new solutions for privacy and security.

3. TOOLS AND TECHNOLOGIES USED

3.1 OpenCV

OpenCV (Open Source Computer Vision Library) is a highly efficient and versatile library for image processing and computer vision tasks. It is widely used in various fields such as robotics, machine learning, and computer vision. In the context of steganography, OpenCV is used to manipulate images, allowing for pixel-level modifications that are necessary for embedding hidden data. It provides functions for reading, writing, and processing image files in various formats.

In this project, OpenCV is primarily used for:

- Loading and displaying images: It allows for seamless loading of input images where the data is to be hidden and visualizing the image after embedding the secret data.
- Image manipulation: OpenCV provides functionalities for altering pixel values, which is crucial for the Least Significant Bit (LSB) method of embedding data into images without significantly affecting the image quality.
- Image conversion: OpenCV can convert between various color spaces (e.g., RGB to grayscale), which is useful in encoding and decoding processes to ensure that the data is hidden efficiently.

By leveraging OpenCV, this project can handle a range of image formats and make necessary pixel modifications for successful data embedding and retrieval.

3.2 Least Significant Bit (LSB) Steganography

Least Significant Bit (LSB) Steganography is a straightforward and widely used technique for hiding data within an image. It involves embedding the secret information in the least significant bits of the image's pixel values, ensuring that the changes made are minimal and visually imperceptible. The LSB method works by modifying the least significant bit of each pixel in an image to represent binary data (either 0 or 1).

- **How LSB Works:** Each pixel in an image consists of three color channels (red, green, and blue) in an RGB color space. The LSB technique modifies the least significant bit (the rightmost bit) of each color channel to encode bits of the secret data. Since this modification is very subtle, the human eye does not notice any significant changes in the image, making it an ideal method for steganography.
- **Advantages of LSB:**
  - It is simple to implement and does not require complex algorithms.
  - It offers a relatively high capacity for hiding data because each pixel can carry a bit of information in each of its color channels.
  - It causes minimal distortion to the image, making it a practical choice for hiding messages in a way that preserves the quality of the image.

In this project, the LSB method was utilized to embed both text and image data within cover images. The data is encoded by manipulating the pixel values and decoded by reversing the process, extracting the hidden information bit by bit.

3.3 Python Imaging Library (PIL)

Python Imaging Library (PIL), now known as Pillow, is a powerful library in Python for opening, manipulating, and saving many different image file formats. It provides several tools and methods that simplify the process of working with images, including reading and writing image files, resizing, cropping, rotating, and editing pixels.

In the context of this steganography project, PIL is used for:

- Image File Operations: PIL allows the program to load images in various formats such as PNG, JPEG, etc., and supports saving the manipulated images back into these formats.
- Image Editing: PIL offers pixel manipulation features that are useful for processes such as embedding data into specific pixels of the image and extracting the hidden data afterward.
- Image Conversion: PIL facilitates easy conversion of images from one format to another, ensuring compatibility with various file types used in the project.

Pillow is particularly useful for handling the extraction and embedding of data in the image, enabling operations like modifying pixel values and displaying the image after encoding or decoding the hidden data.

The tools and technologies used in this project form the backbone of the steganography process. OpenCV provides the necessary functions for manipulating images, while LSB steganography offers a simple and effective technique for hiding data in pixel values. Pillow (PIL) ensures easy image handling and manipulation within the Python environment. By combining these tools, the project enables the encoding and decoding of both text and image data in a way that is undetectable to the naked eye, offering an efficient and practical solution for data concealment.

4. IMPLEMENTATION OF STEGANOGRAPHY TECHNIQUES

4.1 Text Steganography

This section explains how to implement **Text Steganography** using the Least Significant Bit (LSB) method in Python. The steps include converting the message to binary, embedding it into the image pixels, and then decoding it from the modified image. Below is a detailed walkthrough of how this can be done using OpenCV and other libraries in Python.

Step 1: Setup the Python Environment

- Import required libraries:

- `*cv2*` for image processing (using OpenCV)
- `*numpy*` for array manipulation
- `*pandas*` for data handling
- `*os*` for file and directory operations

```
import numpy as np
import pandas as pd
import os
import cv2
from google.colab import drive
```

- Mount Google Drive in Google Colab to access images stored in the drive:

```
drive.mount('/content/drive')
```

Step 2: Convert Text to Binary

The first step in embedding text into an image is converting the text message into a binary string. Each character is converted to its ASCII binary equivalent.

```
name = "MynameisToran"
print("The Original String is:- ", name)

# Convert string to binary
res = ''.join(format(ord(i), 'b') for i in name)
print("The Binary value is:", res)
```

Step 3: Convert Binary Back to Text

To verify if the encoding and decoding process works correctly, we can implement a function to convert binary data back into readable text.

```
def BinaryTointeger(binary):
    decimal, i, n = 0, 0, 0
    while binary != 0:
        dec = binary % 10
        decimal = decimal + dec * pow(2, i)
        binary = binary // 10
        i += 1
    return decimal

bin_data = res
str_data = ''

for i in range(0, len(bin_data), 7):
    temp_data = int(bin_data[i:i + 7])
    decimal_data = BinaryTointeger(temp_data)
    str_data = str_data + chr(decimal_data)
```

print("The Binary value after string conversion is:", str_data)

Step 4: Convert Message to Binary (for Steganography)

We use a function to convert the input message into an 8-bit binary format, suitable for embedding in the least significant bits of the image pixels.

```
def message2binary(message):
    if type(message) == str:
        result = ''.join([format(ord(i), "08b") for i in message])
    elif type(message) == bytes or type(message) == np.ndarray:
        result = [format(i, "08b") for i in message]
    elif type(message) == int or type(message) == np.uint8:
        result = format(message, "08b")
    else:
        raise TypeError("Input type is not supported")
    return result
```

Step 5: Load and Display Image

Before encoding the text, we load the image into which we will embed the data. Using OpenCV, we read the image and display it.

```
image = cv2.imread("/content/drive/MyDrive/Colab Notebooks/Steganography/images/download.jpg")
```

Step 6: Encode Data into Image

The `encode_data` function embeds the binary data of the message into the Least Significant Bit (LSB) of each color channel (Red, Green, Blue) in the image.

```
def encode_data(img):
    data = input("Enter the data to be Encoded:")
```

```
    if len(data) == 0:
        raise ValueError('Data is empty')

    filename = input("Enter the name of the New Image after Encoding(with
extension):")
    no_bytes = (img.shape[0] * img.shape[1] * 3) // 8
    print("Maximum bytes to encode:", no_bytes)

    if len(data) > no_bytes:
        raise ValueError("Insufficient bytes, Need Bigger Image or give Less
Data!")

    data += '*****'  # Delimiter to indicate the end of the message
    data_binary = message2binary(data)
    data_len = len(data_binary)

    data_index = 0

    for i in img:
        for pixel in i:
            r, g, b = message2binary(pixel)

            if data_index < data_len:
                pixel[0] = int(r[:-1] + data_binary[data_index], 2)
                data_index += 1
            if data_index < data_len:
                pixel[1] = int(g[:-1] + data_binary[data_index], 2)
                data_index += 1
            if data_index < data_len:
                pixel[2] = int(b[:-1] + data_binary[data_index], 2)
                data_index += 1
```

```
        if data_index >= data_len:
            break

    cv2.imwrite(filename, img)
    print("Encoded the data successfully and saved the image as", filename)
```

This function:

- Takes input for the message to encode.
- Converts the message to binary.
- Iterates over each pixel, replacing the LSB of each color channel with bits from the binary data.
- Saves the modified image to a file.

Step 7: Decode the Hidden Message

To retrieve the hidden message, we need to extract the LSB from each pixel and reconstruct the binary data. The `decode_data` function does this by reading the LSBs and converting them back into the original message.

```
def decode_data(img):
    binary_data = ""
    for i in img:
        for pixel in i:
            r, g, b = message2binary(pixel)
            binary_data += r[-1]  # Extract LSB of Red Pixel
            binary_data += g[-1]  # Extract LSB of Green Pixel
            binary_data += b[-1]  # Extract LSB of Blue Pixel

    all_bytes = [binary_data[i: i+8] for i in range(0, len(binary_data), 8)]
    decoded_data = ""
    for byte in all_bytes:
        decoded_data += chr(int(byte, 2))
```

```
    if decoded_data[-5:] == "*****":
        break

print("The Encoded data was :--", decoded_data[:-5])
```

This function:

- Extracts the LSBs from the Red, Green, and Blue channels.
- Reconstructs the binary string.
- Converts the binary data back to text, stopping at the delimiter.

Step 8: Test the Encoding and Decoding

1. Encode Data:

```
encode_data(image)
```

This will prompt you to enter the data and output the encoded image.

2. Decode Data:

```
image1 = cv2.imread("/content/drive/MyDrive/Colab Notebooks/Steganography/images/stegano_final.png")
decode_data(image1)
```

After encoding and decoding, the image remains visually unchanged, as the modifications are minimal and hidden in the least significant bits.

- ✓ **Text Steganography** allows embedding text within an image without noticeable changes.
- ✓ The text is converted to binary and embedded in the LSB of each pixel in the image.
- ✓ The **encode_data** function is used to embed the data, and the **decode_data** function extracts the hidden message from the image.

This technique ensures that the text remains hidden while the image appears unchanged to the human eye.

4.2 Image Steganography

Here's an explanation of Image Steganography using Python, with each step outlined with small code snippets.

1. Loading and Resizing Images

We begin by loading two images using `PIL` (Python Imaging Library) and resizing the first image to fit into the second image.

```
from PIL import Image


image1 = Image.open('/path/to/image1.png')
image2 = Image.open('/path/to/image2.png')


image1 = image1.resize((300, 200))
```

This step ensures that the image we want to embed (`image1`) is resized to fit within the target image (`image2`).

2. Converting RGB Values to Binary

We need a method to convert the RGB values of pixels from integer format to binary for manipulation.

```
def int2bin(rgb):
    r, g, b = rgb
    return ('{0:08b}'.format(r), '{0:08b}'.format(g), '{0:08b}'.format(b))
```

This function converts RGB values (each ranging from 0 to 255) into their 8-bit binary equivalents.

3. Converting Binary Back to Integer

Once we've manipulated the binary values, we need to convert them back into integers to reconstruct the image.

```
def bin2int(rgb):
    r, g, b = rgb
    return (int(r, 2), int(g, 2), int(b, 2))
```

This function converts binary representations of RGB values back into integers.

4. Merging Two RGB Values

The core idea of image steganography is to combine the pixel information of two images. We can merge the least significant bits (LSBs) of the RGB values.

```
def merge2rgb(rgb1, rgb2):
    r1, g1, b1 = rgb1
    r2, g2, b2 = rgb2
    return (r1[:4] + r2[:4], g1[:4] + g2[:4], b1[:4] + b2[:4])
```

Here, we merge the first 4 bits of the first image's RGB and the first 4 bits of the second image's RGB.

5. Merging Two Images

Using the merging function for RGB values, we apply it to every pixel in both images. We also ensure that the first image fits inside the second image, pixel by pixel.

```
def merge2img(img1, img2):
    if img1.size[0] > img2.size[0] or img1.size[1] > img2.size[1]:
        print("Cannot merge. Image 1 is larger than Image 2.")
        return
    pixel_tuple1 = img1.load()
    pixel_tuple2 = img2.load()

    new_image = Image.new(img2.mode, img2.size)
    pixels_new = new_image.load()
```

```
    for row in range(img2.size[0]):
        for col in range(img2.size[1]):
            rgb1 = int2bin(pixel_tuple2[row, col])
            rgb2 = int2bin((0, 0, 0))

            if row < img1.size[0] and col < img1.size[1]:
                rgb2 = int2bin(pixel_tuple1[row, col])

            merged_rgb = merge2rgb(rgb1, rgb2)
            pixels_new[row, col] = bin2int(merged_rgb)

    new_image.save('merged_image.jpg')
    return new_image
```

This function merges the two images and saves the resulting image.

6. Unmerging the Image

To extract the hidden image, we reverse the merging process. We isolate the part of the RGB values that were used from the second image.

```
def unmerge(path):
    img = Image.open(path)
    pixel_map = img.load()

    new_image = Image.new(img.mode, img.size)
    pixels_new = new_image.load()

    for row in range(img.size[0]):
        for col in range(img.size[1]):
            r, g, b = int2bin(pixel_map[row, col])
```

```
        rgb = (r[4:] + "0000", g[4:] + "0000", b[4:] + "0000")
        pixels_new[row, col] = bin2int(rgb)

    new_image.save('unmerged_image.png')
    return new_image
```

This function retrieves the hidden image by extracting the embedded pixel information from the merged image.

7. Adjusting the Merging Pattern

To improve the quality of the unmerged image, we use a different approach by adjusting the number of bits taken from each image.

```
def merge2rgb2(rgb1, rgb2):
    r1, g1, b1 = rgb1
    r2, g2, b2 = rgb2
    return (r1[:6] + r2[:2], g1[:6] + g2[:2], b1[:6] + b2[:2])
```

In this case, we take 6 bits from the second image and 2 bits from the first image. This allows for better-quality results when unmerging the images.

8. Final Merging and Unmerging

With the new merging pattern, we can merge and unmerge the images while preserving quality.

```
merged_image2 = merge2img2(image1, image2)
unmerged_image2 = unmerge2(merged_image2)
```

Now, the image merged with the new pattern will maintain the integrity of the hidden image, and the unmerged image will have better quality without noise.

Through these steps, you can encode one image inside another using steganography by manipulating the least significant bits of pixel values. The result is an image where the hidden information is not easily detectable, providing a secure way of embedding data.

5. RESULTS AND SCREENSHOTS

5.1 Text Steganography Results

Text Steganography.ipynb ☆

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[ ] import numpy as np
    import pandas as pd

    import os
    for dirname, _, filenames in os.walk('/kaggle/input'):
        for filename in filenames:
            print(os.path.join(dirname, filename))
```

```
[ ] import cv2
```

```
[ ] from google.colab import drive
    drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Checking Text to Binary-->

```
[ ] name="MynameisToran"
```

```
[ ] print("The Original String is:- "+name)
```

```
The Original String is:- MynameisToran
```

```
[ ] res=''.join(format(ord(i),'b')for i in name)
```

```
[ ] print(res)
```

```
1001101111100111011101100001110110111001011101001111001110101001101111111001011000011101110
```

Checking Binary to Text-->

```
def BinaryTointeger(binary):

    binary1 = binary
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = binary % 10
        decimal = decimal + dec * pow(2, i)
        binary = binary//10
        i += 1
    return (decimal)

bin_data =res

print("The binary value is:", bin_data)


str_data =' '


for i in range(0, len(bin_data), 7):
    temp_data = int(bin_data[i:i + 7])
    decimal_data = BinaryTointeger(temp_data)
    str_data = str_data + chr(decimal_data)


print("The Binary value after string conversion is:",str_data)
```

```
The binary value is: 1001101111100111011101100001110110111001011101001111001110101001101111111001011000011101110
The Binary value after string conversion is:  MynameisToran
```

Function to convert the input message to Binary..

```
[ ] def message2binary(message):
      if type(message) == str:
        result= ''.join([ format(ord(i), "08b") for i in message ])

      elif type(message) == bytes or type(message) == np.ndarray:
        result= [ format(i, "08b") for i in message ]

      elif type(message) == int or type(message) == np.uint8:
        result=format(message, "08b")

      else:
        raise TypeError("Input type is not supported")

      return result
```

Here we are using 08b as we require 8 bit representation of binary digits.

If we will be using only b then it will not add 0 to convert it into 8 bits and returns the binary converted value..

```
[ ] # message2binary("helloouserthisisoran")
```

```
[ ] # message2binary("mynameistoran")
```

```
[ ] # r,g,b=message2binary([50,35,155])
```

Importing Image-->

```
from IPython.display import Image
import os
```

```
Image('/content/drive/MyDrive/Colab Notebooks/Steganography/images/download.jpg')
```

```
image=cv2.imread("/content/drive/MyDrive/Colab Notebooks/Steganography/images/download.jpg")
```

```
#image
```

```
# for i in image:
#     for pixel in i:
#         print(pixel)
```

How to overwrite the LSB bit of a binary number and converting it to decimal..-->

```
h='1000110'
int(h[:-1]+'1',2)
```

```
71
```

[:-1] neglects the LSB bit then we can add a bit and by int(value,2) we will change it to new decimal value..

```
list1=[ ]
```

ENCODER FUNCTION

```
[ ] def encode_data(img):
      data=input("Enter the data to be Encoded:")
      if (len(data) == 0):
        raise ValueError('Data is empty')

      filename = input("Enter the name of the New Image after Encoding(with extension):")

      no_bytes=(img.shape[0] * img.shape[1] * 3) // 8

      print("Maximum bytes to encode:", no_bytes)

      if(len(data)>no_bytes):
          raise ValueError("Error encountered Insufficient bytes, Need Bigger Image or give Less Data !!")

      # Using the below as delimeter
      data +='*****'

      data_binary=message2binary(data)
      print(data_binary)
      data_len=len(data_binary)

      print("The Length of Binary data",data_len)

      data_index = 0

      for i in img:
          for pixel in i:

            r, g, b = message2binary(pixel)
           # print(r)
           # print(g)
           # print(b)
          #   print(pixel)
            if data_index < data_len:
                # hiding the data into LSB(Least Significant Bit) of Red Pixel
#                 print("Original Binary",r)
                # print("The old pixel",pixel[0])
                pixel[0] = int(r[:-1] + data_binary[data_index], 2) #changing to binary after overwrriting the LSB bit of Red Pixel
#                 print("Changed binary",r[:-1] + data_binary[data_index])

                data_index += 1
                list1.append(pixel[0])

            if data_index < data_len:
               # hiding the data into LSB of Green Pixel
                pixel[1] = int(g[:-1] + data_binary[data_index], 2) #changing to binary after overwrriting the LSB bit of Green Pixel
                data_index += 1
                list1.append(pixel[1])

            if data_index < data_len:
                # hiding the data into LSB of  Blue Pixel
                pixel[2] = int(b[:-1] + data_binary[data_index], 2) #changing to binary after overwrriting the LSB bit of Blue pixel
                data_index += 1
                list1.append(pixel[2])

                # if data is encoded, just breaking out of the Loop
            if data_index >= data_len:
                break


      cv2.imwrite(filename,img)

      print("Encoded the data successfully and the image is successfully saved as ",filename)
```

ENCODING THE DATA-->

```
[ ] encode_data(image)
```

```
Enter the data to be Encoded:MynameisToran
Enter the name of the New Image after Encoding(with extension):stegano_final.png
Maximum bytes to encode: 94357
010011010111100101101110011000010110110101100101011010010111001101010100011011110111001001100001011011100010101000101010001010100010101000101010
The Length of Binary data 144
Encoded the data successfully and the image is successfully saved as  stegano_final.png
```

DECODER FUNCTION-->

```
[ ] def decode_data(img):

    binary_data = ""
    for i in img:
        for pixel in i:

          #   print(pixel)
            r, g, b = message2binary(pixel)
            binary_data += r[-1]  #Extracting Encoded data from the LSB bit of Red Pixel as we have stored in LSB bit of every pixel.
            binary_data += g[-1]  #Extracting Encoded data from the LSB bit of Green Pixel
            binary_data += b[-1]  #Extracting Encoded data from LSB bit of Blue Pixel

    # splitting by 8-bits
    all_bytes = [ binary_data[i: i+8] for i in range(0, len(binary_data), 8) ]

    # Converting the bits to Characters
    decoded_data = ""
    for byte in all_bytes:
        decoded_data += chr(int(byte, 2))
        if decoded_data[-5:] == "*****": #Checking if we have reached the delimeter which is "*****"
            break


    print("The Encoded data was :--",decoded_data[:-5])
```

DECODING THE DATA-->

```
[ ] image1=cv2.imread("/content/drive/MyDrive/Colab Notebooks/Steganography/images/stegano_final.png")
```

```
[ ] decode_data(image1)
```


```
DÑ□□n³A□öV£®@¶ñ□Íu,luFãû□õ□□Æà□ô·□m"□□²±6¸□ÿ9o□□Åï□¿x÷%] =□$¿□Wk□□s□LÚfÿ¼□o¦eKý□q°:ÝoO□□ôy□è°□Ã³µ£æ□·4MÜÝÔ□²ü=¾□Â¨«jç9#<·□¤□□□³[
□S□(
ó#·     dLâ~°-□□s1`?□úFãfc¸Ç·¸Â*3C&-□3o,□/ÆGMÅ□□_oA6□XÌÊº□□□·íÊ7ö□□□ø1□õ□□□□Û}   □□S¯Á`BH□Öº/□}□d□ä>□x□□I7¬□U¢ð¹E□ý×>¤□÷ç²å$Íâ□ô
õ□¡Öîâ¶1□ÆË£□¸44□□ÚLV-s□ø×<□□Ü·çÏa×g9]□0□ô:□ú×"MÔK!□9ç□□Â" å£ÌÞ
□ßJ□3ã²§1ìá&□M N□Ì ?Gg□±□X¿×MaºÙþèÉ§!ûi□I,□ÀÄþ□NÙ□vÎ□Âu¶M□à□kË}Ø□□çñ□□Uºî□JúFT□c
SEk□Ý□±ø£□□Ç□Ö□ë4Çæ     □ã□□Ù-□Öî»□□³□¨□□]L<O¼öþQ~å □□Ï;□Ý¿õ¢)□_V□□□´Ç¥ßëævÂ¶1□□³«..□µ□½□2,□±ú
T%Ü□□`ÆxA□ÚZ{zÄ□□¼¾«y@□×.hÞÓ@=□§Ü×Ø□□□t2□□Ú¤ÉñÐCx·□u$□□□â        ²%ç□DM¯¦mEâ□æ2□□²=R□Vp□ÒÏ=Y□îh=ê□□ÆK¿/ß□^□/□□À$K□
ó|□L!Û;□ìë□□□ÔÀ□Ì□□÷{{□□□à0ï);`□¿□□□£Ù□ÖÇO□Ì□□ÛÃ□□¸□Eúúkä38|¼!ËÏ□5Ýá□áEsxôK□□□ÕÂÎìð7H□)sd
□éº□6@Ü!□o¥&k
Lõ+
□□□□·' Uô]9åö□<□Þ)·PR1¯}¿□□Vº□  7Lü¨Ç□wáT+PD□□Îx□ôêÕ!□Øj□□ÀÚ²ì`□vVÑ9Ë□H
```

The Original Image

```
Image('/content/drive/MyDrive/Colab Notebooks/Steganography/images/download.jpg')
```

The Image after Encoding Data

```
Image('/content/drive/MyDrive/Colab Notebooks/Steganography/images/stegano_final.png')
```

Here we are seeing that after encoding the data also there is not much change in the image.Really we cannot find any difference between the original and the Steganographed image..

5.2 Image Steganography Results

Image Steganography.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[ ] from PIL import Image
```

```
[ ] from google.colab import drive
    drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
[ ] image1 = Image.open('/content/drive/MyDrive/Colab Notebooks/Steganography/images/nature.png')
    image2=Image.open('/content/drive/MyDrive/Colab Notebooks/Steganography/images/nature1.png')
```

```
[ ] image1
```

```
[ ] image2
```

```
[ ] image1.size
```

```
(455, 463)
```

Resizing Image 1

```
[ ] image1= image1.resize((300,200))
```

```
[ ] image1
```

```
[ ] image1.size
```

```
(300, 200)
```

```
[ ] image2.size
```

```
(525, 355)
```

We will be merging image 1 on image 2

Integer to binary Coversion->

```
[ ] def int2bin(rgb):

        #Will convert RGB pixel values from integer to binary
        #INPUT: An integer tuple (e.g. (220, 110, 96))
        #OUTPUT: A string tuple (e.g. ("00101010", "11101011", "00010110"))

        r, g, b = rgb
        return ('{0:08b}'.format(r),
                '{0:08b}'.format(g),
                '{0:08b}'.format(b))
        #Return converted r,g,b binary values separately..
```

Binary to Integer Conversion-->

```
[ ] def bin2int(rgb):

        #Will convert RGB pixel values from binary to integer.
        #Reverse of the first part.

        r, g, b = rgb
        return (int(r, 2),
                int(g, 2),
                int(b, 2))
        #return converted r,g,b integer values separately
```

```
[ ] r,g,b=int2bin((225,6,7))
    r
```

```
'11100001'
```

```
[ ] r[:4]
```

```
'1110'
```

[:4] will be taking the first 4 digits.

And [:-4] will be taking the digits ignoring the last 4 digits..

```
[ ] print(g)
    print(b)
```

```
00000110
00000111
```

```
[ ] bin2int(('11100001', '00000110', '00000111'))
```

```
(225, 6, 7)
```

Our convertion functions are working perfectly.

```
[ ] def merge2rgb(rgb1,rgb2):

        #Will merge two RGB pixels using 4 least significant bits.
        #INPUT: A string tuple ( ("00101010", "11101011", "00010110")),another string tuple (e.g. ("00101010", "11101011", "00010110"))
        #OUTPUT: An integer tuple with the two RGB values merged
        #Will be merging the first four digits of first image and first four digits of 2nd image(i.e to be merged) as last four digits..


        r1,g1,b1=rgb1
        r2,g2,b2=rgb2

        return (r1[:4]+r2[:4],
                 g1[:4]+g2[:4],
                 b1[:4]+b2[:4]
              )
```

Function to merge two Images-->

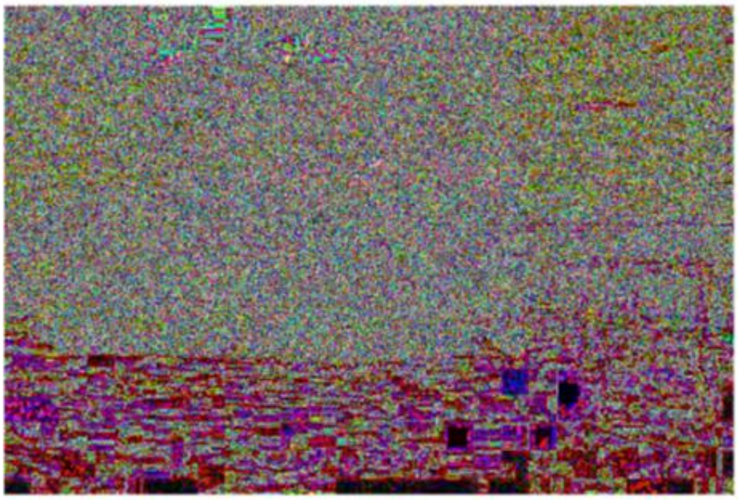
```
[ ] def merge2img(img1,img2):
      # The First image will be merged into the second image.


      image1=img1
      image2=img2
      #print('toran')

      # Condition for merging
      if(image1.size[0]>image2.size[0] or image1.size[1]>image2.size[1]):
        print("Cannot merge as the size of 1st Image is greater than size of 2nd Image")
        return
      # Getting the pixel map of the two images

      pixel_tuple1 = image1.load()
      pixel_tuple2 = image2.load()

      #print(pixel_tuple1)
      #print(pixel_tuple2)

      # The new image that will be created.
      new_image = Image.new(image2.mode, image2.size) # Setting the size of Image 2 as Image 1 will be merged to Image 2.
      pixels_new = new_image.load()

      for row in range(image2.size[0]):
        for col in range(image2.size[1]):

            rgb1 = int2bin(pixel_tuple2[row, col])

            # Using a black pixel as default
            rgb2 = int2bin((0, 0, 0))

            # Converting the pixels of image 1 if condition is satisfied

            if(row <image1.size[0] and col< image1.size[1]):
              rgb2= int2bin(pixel_tuple1[row,col])


            merge_rgb= merge2rgb(rgb1,rgb2)

            pixels_new[row,col] = bin2int(merge_rgb)

      #print('toran')
      new_image.convert('RGB').save('/content/drive/MyDrive/Colab Notebooks/Steganography/images/merged1.jpg')

      return new_image
```

```
[ ] !pip install Pillow

    from PIL import Image

    def int2bin(rgb):
        """
        Convert an integer tuple representing an RGB color to its binary representation.

        INPUT: An integer tuple (e.g. (64, 255, 10))
        OUTPUT: A string tuple (e.g. ("00101010", "11101011", "00010110"))
        """
        # Check if rgb has 3 elements (R, G, B)
        if len(rgb) == 3:
            r, g, b = rgb
        # If rgb has 4 elements (R, G, B, A), ignore A
        elif len(rgb) == 4:
            r, g, b, _ = rgb  # Ignore the alpha channel
        else:
            raise ValueError(f"Invalid rgb tuple: {rgb}. Expected 3 or 4 elements.")

        return ('{0:08b}'.format(r),
                '{0:08b}'.format(g),
                '{0:08b}'.format(b))


    def merge2rgb(rgb1, rgb2):
        """
        Merge two RGB pixels using 4 least significant bits.

        INPUT: A string tuple (e.g. ("00101010", "11101011", "00010110")),
               another string tuple (e.g. ("00101010", "11101011", "00010110"))
        OUTPUT: An integer tuple with the two RGB values merged
        """
        r1, g1, b1 = rgb1
        r2, g2, b2 = rgb2

        return (r1[:4] + r2[:4],
                g1[:4] + g2[:4],
                b1[:4] + b2[:4])


    def merge2img(img1, img2):
        """
        Merge two images. The First image will be merged into the second image.
        """
        image1 = img1
        image2 = img2

        # Condition for merging
        if image1.size[0] > image2.size[0] or image1.size[1] > image2.size[1]:
            print("Cannot merge as the size of 1st Image is greater than size of 2nd Image")
            return

        # Getting the pixel map of the two images
        pixel_tuple1 = image1.load()
        pixel_tuple2 = image2.load()

        # The new image that will be created.
        new_image = Image.new(image2.mode, image2.size)
        pixels_new = new_image.load()

        for row in range(image2.size[0]):
            for col in range(image2.size[1]):
                rgb1 = int2bin(pixel_tuple2[row, col])

                # Using a black pixel as default
                rgb2 = int2bin((0, 0, 0))

                # Converting the pixels of image 1 if condition is satisfied
                if row < image1.size[0] and col < image1.size[1]:
                    rgb2 = int2bin(pixel_tuple1[row, col])

                merge_rgb = merge2rgb(rgb1, rgb2)

                pixels_new[row, col] = bin2int(merge_rgb)

        # Save the merged image
        new_image.convert

    Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages (10.4.0)
```

See now our image 1 is merged inside image 2.But still image 2 is looking as it was earlier..

```
[ ] def unmerge(path):

        img=Image.open(path)

        # Loading the pixel map
        pixel_map = img.load()

        new_image = Image.new(img.mode, img.size)
        pixels_new = new_image.load()

        # Tuple used to store the image original size
        original_size = img.size

        for row in range(img.size[0]):
            for col in range(img.size[1]):
                # Get the RGB (as a string tuple) from the current pixel
                r, g, b = int2bin(pixel_map[row, col])

                # Extract the last 4 bits (corresponding to the hidden image)
                # Concatenate 4 zero bits because we are working with 8 bit values
                rgb = (r[4:] + "0000",
                       g[4:] + "0000",
                       b[4:] + "0000")

                # Convert it to an integer tuple
                pixels_new[row, col] = bin2int(rgb)

                # If this is a 'valid' position, store it
                # as the last valid position
                if pixels_new[row, col] != (0, 0, 0):
                    original_size = (row + 1, col + 1)

        # Crop the image based on the 'valid' pixels
        new_image = new_image.crop((0, 0, original_size[0], original_size[1]))

        new_image.save('/content/drive/MyDrive/Colab Notebooks/Steganography/images/unmerged1.png')

        return new_image
```

```
unmerge('/content/drive/MyDrive/Colab Notebooks/Steganography/images/merged1.jpg')
```



Double-click (or enter) to edit

Here we are seeing that the unmerged image is not clear at all

now we should change our merging pattern.

We can now take 2 MSBs from image 1 and add 6 MSBs of image2 while merging..

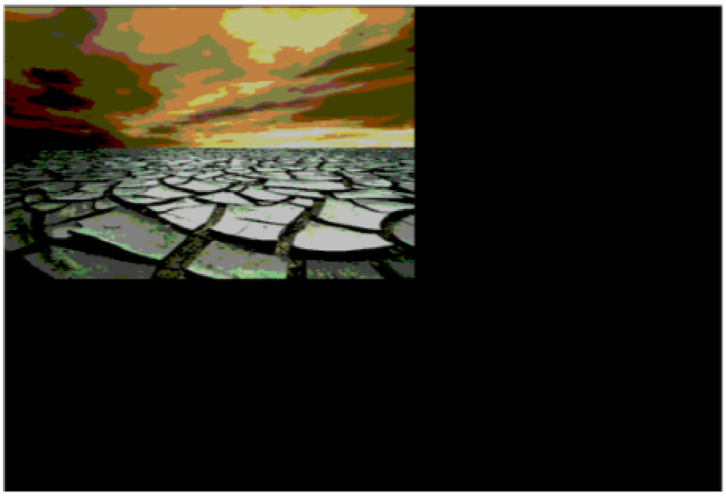
```
[ ] def merge2rgb2(rgb1, rgb2):

        r1, g1, b1 = rgb1
        r2, g2, b2 = rgb2
        rgb = (r1[:6] + r2[:2],
               g1[:6] + g2[:2],
               b1[:6] + b2[:2])
        return rgb
```

```
[ ]  def merge2img2(img1, img2):

        image1=img1
        image2=img2
        #print('toran')

        # Condition for merging
        if(image1.size[0]>image2.size[0] or image1.size[1]>image2.size[1]):
           print("Cannot merge as the size of 1st Image is greater than size of 2nd Image")
           return

      # Getting the pixel map of the two images
        pixel_tuple1 = image1.load()
        pixel_tuple2 = image2.load()

        #print(pixel_tuple1)
        #print(pixel_tuple2)

        # The new image that will be created.
        new_image = Image.new(image2.mode, image2.size) # Setting the size of Image 2 as Image 1 will be merged to Image 2.
        pixels_new = new_image.load()

        for row in range(image2.size[0]):
          for col in range(image2.size[1]):

            rgb1 = int2bin(pixel_tuple2[row, col])

            # Using a black pixel as default
            rgb2 = int2bin((0, 0, 0))

            # Converting the pixels of image 1 if condition is satisfied

            if(row <image1.size[0] and col< image1.size[1]):
              rgb2= int2bin(pixel_tuple1[row,col])


            merge_rgb= merge2rgb2(rgb1,rgb2)

            pixels_new[row,col] = bin2int(merge_rgb)

        #print('toran')
        new_image.convert('RGB').save('/content/drive/MyDrive/Colab Notebooks/Steganography/images/merged2.jpg')

        return new_image
```

```python
def unmerge2(img):

    pixel_map = img.load()

    new_image = Image.new(img.mode, img.size)
    pixels_new = new_image.load()


    original_size = img.size

    for row in range(img.size[0]):
        for col in range(img.size[1]):
            r, g, b = int2bin(pixel_map[row, col])

            # Extracting the last 6 bits (corresponding to the hidden image) and adding zeroes to increase the brightness.

            rgb = (r[6:] + "000000",
                   g[6:] + "000000",
                   b[6:] + "000000")

            # Convert it to an integer tuple
            pixels_new[row, col] = bin2int(rgb)

           #If this is a 'valid' position, store it as a last valid option
            if pixels_new[row, col] != (0, 0, 0):
                original_size = (row + 1, col + 1)

    # Crop the image based on the 'valid' pixels
    new_image = new_image.crop((0, 0, original_size[0], original_size[1]))

    return new_image
```

```python
merged_image2 = merge2img2(image1,image2)
merged_image2
```

```
unmerged_image2 = unmerge2(merged_image2)
unmerged_image2
```



So we have concluded that the 2 MSBs of Image 1 were more important than 4 MSBs of image 1, they were adding noise to the image. So merging 6 MSBs of Image2 and 2 MSBs of Image 1 was fruitful.

5.3 Final Conclusion Results (Text & Image Steganography)

Advantage Over Cryptography

The advantage of steganography over cryptography alone is that the intended secret message does not attract attention to itself as an object of scrutiny.

Plainly visible encrypted messages, no matter how unbreakable they are, arouse interest and may in themselves be incriminating in countries in which encryption is illegal.

To Run

Run the respective ipynb file and in the place of image choose the image you want to choose..

Incase of Text Steganography

Then run the encode_data function to encode the desired data.

After encoding the data run the decode_data function to decode the encoded data.

Incase of Image Steganography

Run the merge2img2 function by selecting the images.

And then run the unmerge2 function to get the initial image after merging.

ENCODING THE DATA-->

```
[47] encode_data(image)
```

```
Enter the data to be Encoded:MynameisToran
Enter the name of the New Image after Encoding(with extension):stegano_final.png
Maximum bytes to encode: 94357
010011010111100101101110011000010110110101100101011010010111001101010100011011110111001001100001011011100010101000101010001010100010101000101010
The Length of Binary data 144
Encoded the data successfully and the image is successfully saved as  stegano_final.png
```

DECODING THE DATA-->

```
[49] image1=cv2.imread("/content/drive/MyDrive/Colab Notebooks/Steganography/images/stegano_final.png")
```

```
[50] decode_data(image1)
```

```
The Encoded data was :--
```

[illegible]

These were the Images before and after encoding the data in Text Steganography:--

Original Image	Image After Encoding
	

These were the images before and after encoding images in Image Steganography..

Image 1 is merged on Image 2..

Cover Image	Image to be Merged
	

Merged Image	Unmerged Image
	

6. RECENT RESEARCH AND DEVELOPMENTS IN STEGANOGRAPHY

6.1 Advances in Steganography Techniques

Recent developments in steganography have shifted to techniques that use artificial intelligence (especially deep learning) to increase the effectiveness of information hiding. AI-based techniques such as generative adversarial networks (GANs) are used to optimize the balance between data availability, imperceptibility, and search robustness.

In modified steganography, researchers use image features to change the placement strategy, improve the flexibility of the method, and make hidden information less visible. Many new methods have also been introduced that take steganography beyond simple image-based encryption for various media such as video and audio.

- **Deep Learning-Based Techniques:** Leveraging neural networks, especially GANs, allows for improved data hiding with minimal impact on the carrier media, enhancing both imperceptibility and robustness.
- **Adaptive Steganography:** Adaptive methods adjust hiding strategies based on the carrier media's features, making hidden data harder to detect and improving resistance to steganalysis.
- **Expanding to Multimedia:** New techniques are extending beyond images to audio, video, and even 3D models, broadening the scope of steganography in digital communications.

6.2 Challenges in Steganography

The main problems with steganography are maintaining the size balance, image quality, and the stability of steganlysis (the technique used to identify hidden objects). Larger data can cause changes in the upload environment, so it is important to maintain image quality when bulk uploading. In addition, the advancement of advanced seganalysis techniques means that medical professionals must develop additional skills to avoid detection.

Ensuring that steganography algorithms are resilient to attacks by steganalysis software continues to be a major challenge as attackers continue to improve their techniques for identifying, extracting, and even exploiting hidden information.

- **Balancing Payload and Quality:** Higher data capacities often compromise image or media quality, making it challenging to hide large payloads without visible alterations.
- **Steganalysis Resistance:** As steganalysis methods improve, it becomes increasingly difficult to evade detection, requiring continuous innovation in hiding strategies.
- **Algorithm Security:** Protecting algorithms from attackers who aim to detect, extract, or alter hidden data is a major challenge, especially as counter-steganographic techniques evolve.

6.3 Steganalysis and Counter-Steganography

In response to advances in steganography, researchers in the field of steganalysis (the search and analysis of hidden information) have also made rapid progress. Steganalysis focuses on identifying patterns or inconsistencies in media that reveal hidden information. Recent techniques include machine learning algorithms that can identify steganographic patterns with high accuracy, even when changes have been made to evade detection.

Anti-steganography also involves removing artifacts without compromising the integrity of the original content. The ongoing battle between steganography and steganalysis has led to the development of more sophisticated encryption techniques and more efficient detection techniques, causing the field to constantly change.

- **Machine Learning for Detection:** Advanced steganalysis employs machine learning models to identify patterns that signal hidden data, improving detection accuracy and making it harder for steganography to go undetected.
- **Statistical and Structural Analysis:** By analyzing statistical anomalies or structural inconsistencies in digital media, steganalysis can detect alterations made by steganography techniques, especially in images, audio, and video files.
- **Payload Extraction and Data Sanitization:** Counter-steganography tools now focus on extracting hidden data without damaging the original media, as well as sanitizing files by removing potential payloads, enhancing security in sensitive environments.

7. PROJECT REPOSITORY AND DEMO LINKS

7.1 Project Repository Link :

GitHub Repository: https://github.com/toranvathani/ProjectonSteganography

Google Collab Links:

Text Steganography :
https://colab.research.google.com/drive/1BwrJDI4ZxJIs0U8p_VPsaWub7iBI1f1G#scrollTo=e4FoMdlRMWRR

Image Steganography :
https://colab.research.google.com/drive/1olDBDtKcBvtfTCyU7Dv6TtDu1BTPwtHE#scrollTo=ad-JuDlXVzEf

7.2 Video Demo Link :

Video Demo: https://www.loom.com/share/13fd473ee97d455394510e17e2d34722

8. BIBLIOGRAPHY

1. Anderson, R. J., & Petitcolas, F. A. P. (1998). On the limits of steganography. IEEE Journal on Selected Areas in Communications, 16(4), 474–481.
   - Discusses fundamental limits and theoretical approaches in steganography, emphasizing data capacity and security concerns.

2. Fridrich, J. (2009). Steganography in Digital Media: Principles, Algorithms, and Applications. Cambridge University Press.
   - Comprehensive coverage on various steganography algorithms and applications, focusing on digital media like images and audio.

3. Johnson, N. F., & Jajodia, S. (1998). Exploring steganography: Seeing the unseen. IEEE Computer, 31(2), 26–34.
   - Introduces core concepts in steganography and common techniques used in digital environments, along with a historical perspective.

4. Boehm, B., et al. (2018). A Survey on Recent Advances in Steganography. ACM Computing Surveys, 51(3), 56.
   - A survey of recent developments in steganographic techniques, with emphasis on advancements in capacity, security, and algorithmic innovations.

5. Qazanfari, K., & Safabakhsh, R. (2015). Adaptive steganography based on integer wavelet transform and chaotic mapping. Information Sciences, 345, 276–294.
   - Examines adaptive steganography methods, focusing on techniques that adjust hiding methods based on the characteristics of the media.

6. Ker, A. D. (2004). A General Framework for Structural Steganalysis of LSB Replacement. Proceedings of the International Workshop on Information Hiding, 295–311.
   - Presents methods in steganalysis for detecting least significant bit (LSB) replacement, a commonly used steganographic technique.

7. Zhou, Z.-H., & Shi, Y.-Q. (2018). Image Steganalysis Based on Convolutional Neural Networks. IEEE Transactions on Information Forensics and Security, 13(9), 2340–2354.
   - Explores the use of convolutional neural networks in detecting steganographic content within images, marking advancements in automated steganalysis.

8. Provos, N., & Honeyman, P. (2003). Hide and seek: An introduction to steganography. IEEE Security & Privacy Magazine, 1(3), 32–44.
   - Provides a foundational introduction to steganography, including techniques, applications, and the challenges involved in countering detection.

9. Baluja, S. (2017). Hiding images in plain sight: Deep steganography. Advances in Neural Information Processing Systems (NeurIPS), 30, 2064–2072.
   - Investigates deep learning-based steganography, demonstrating how neural networks can be used to hide images within images, enhancing imperceptibility and robustness.

10. Holub, V., Fridrich, J., & Denemark, T. (2014). Universal distortion function for

steganography in an arbitrary domain. EURASIP Journal on Information Security, 2014(1), 1–13.
- Proposes a universal distortion function that allows for improved adaptability in steganographic applications, reducing detectable artifacts across various domains.

11. Li, C., et al. (2019). Steganography and steganalysis of JPEG images: A review. Signal Processing: Image Communication, 75, 1–16.
- Reviews JPEG-based steganography and steganalysis methods, focusing on methods that maintain image quality while increasing resistance to detection.

12. Tiwari, A. K., Sharma, V. D., & Dhawan, S. K. (2021). Recent advances in deep learning-based steganography and steganalysis: A survey. Artificial Intelligence Review, 54(3), 1799–1832.
- Surveys recent developments in deep learning applications for both steganography and steganalysis, highlighting the potential and challenges of neural network-based methods for hiding and detecting information.