Tunis Business
University of Tunis

PixHide
Image Steganography System

Secrets between Pixels

Information Assurance and Security project

Submitted by : Amal Jawahdou
Major: IT Minor: BA

Submitted to : Prof. Manel Abdelkader

Declaration of Academic Ethics

I, Amal Jawahdou, declare that this written report, submitted for the Information Assurance and Security final project, is my own work that reflects my own ideas in my own way, and that any external ideas or words included in this paper have been properly cited and acknowledged respectively.

I affirm that I have adhered to respect all the academic ethics of honesty and integrity. And, I have ensured that all the mentioned information in this report is free from misguidance, fallacies, and misrepresentation.

I am fully cognizant that any violation of these ethical principles may lead to serious disciplinary actions, and I am committed to accept the consequence of any breaches.

Amal Jawahdou

Abstract

Hiding secrets has been always a serious challenge for the human being to protect information from going to the wrong hands. That's why they invented various methods to make it almost impossible for unauthorized persons to access this information.

One of those methods was to hide information into something else that's not a secret, also called steganography, which seems at first glance illogical, but when someone learns about it, they will discover how effective it is.

PixHide is a system that helps to hide secret in images or reveal the ones already hidden in it. It gives the client the opportunity to select the steganography method they want from various choices, then to enter the message to encrypt or an image to decrypt, then get the result.

This report will explore the steganography field, the key concepts and the main existing solutions also, an overview of the proposed solution "PixHide" and its principal features.

TABLE OF CONTENTS

1. INTRODUCTION

In this report, I delved into the conceptualization and implementation of an innovative platform PixHide an image steganography system to hide messages in images or extract secret information from a given image.

1.1. Background

Nowadays, digital communication is all around us, that why keeping private and confidential data safe has become super important. Old ways of coding data are strong but can make people notice secret messages. Steganography is a sneaky way to hide info in normal stuff like pics, sound files, or text.

1.2. Motivation

The proliferation of digital images across various online platforms has made image steganography an increasingly relevant and intriguing field of study. This kind of hiding lets you pass notes in images. It's hard to spot by both people and machines. So, it's a good pick for lots of uses, like secret talks, marking pictures, and keeping rights safe.

1.3. Objectives

The primary objective of this project is to develop an Image Steganography System capable of concealing sensitive information within digital images using advanced steganography algorithms. Key objectives include:

- Implementation of multiple steganography algorithms to provide users with a choice of concealment techniques.
- Development of a user-friendly web interface for encoding and decoding secret messages within images.

1.4. Problem statement

Modern secure communication methods often include encryption, which can arouse suspicion and suspicion. Conversely, steganography provides a means of hiding secret information into non secret things, but its success depends on good techniques and methods. Therefore, the challenge is to develop a unified steganography that balances steganography performance and anonymity to enable secret communication without any room for suspicion.

1.5. Scope of Project

This project will focus on implementing an Image Steganography System using Python-based open-source libraries, with a particular emphasis on providing users with a choice of steganography algorithms. The system will be designed to accommodate both encoding and decoding functionalities through a user-friendly web interface.

2. Steganography Exploration

2.1. What's Steganography ?

Steganography is the practice of representing information within another message or physical object, in such a manner that the presence of the information is not evident to human inspection. In computing/electronic contexts, a computer file, message, image, or video is concealed within another file, message, image, or video. [6]

2.2. The Difference Between Steganography, Cryptography, and Obfuscation

Steganography, cryptography, and obfuscation are three related terms that appear similar but have distinct definitions

- **Cryptography** attempts to encode a message, making it difficult or impossible for anyone except the intended recipient to decrypt it. The encoding and decoding process is accomplished using cryptographic keys that translate back and forth between the true message and its encrypted version.
- **Steganography** attempts to hide a message within another object. Not only does steganography seek to make this information harder to understand, but it also seeks to conceal that a message is being sent in the first place.
- **Obfuscation** is any technique that prevents third parties from understanding a message. For example, a program's source code may be obfuscated by removing the whitespace, making the message difficult for humans to read. [2]

2.3. Types of Steganography

1. **Text steganography**

   This method hide a secret message inside a text file. Text steganography techniques might include the use the first letter in each sentence to form the hidden message , adding meaningful typos or encoding information through punctuation.

2. **Image steganography**

   In image steganography, secret information are encoded within a digital image. This technique relies on the fact that minuscule changes in image are very difficult to detect with the human eye. As example, one image can be concealed within another by using the least significant bits of each pixel in the image to represent the hidden image instead.

3. **Video steganography**

   Video steganography involves concealing information within video sequences, like sneaking a message into a movie clip. As digital videos are represented as a sequence of consecutive images, each video frame can encode a separate image, hiding a coherent video in plain sight.

4. **Audio steganography**

   The practice of hiding information in an audio file, such as a song, speech, or sound effect. One simple form of audio steganography is "backmasking" in which secret messages are played backwards on a track . More sophisticated techniques might involve the least significant bits of each byte in the audio file.

5. **Network steganography**

   This technique is basically a clever digital steganography technique that hides information inside network traffic. Data can be concealed within the TCP/IP headers or payloads of network packets.

2.4. Main Components of Any Image Steganography System

Any image steganography system is based on some essential components that are as follows

- **Cover Image:**
  The original image in which the secret message will be hidden.
- **Secret Message:**
  The confidential information that needs to be concealed within the cover image.
- **Stego Key:**
  A secret key used in some steganography techniques for encryption or decryption.
- **Steganography Algorithm:**
  The method used to embed the secret message into the cover image and extract it back.

- **Stego Image:**
  The resulting image after embedding the secret message.
- **Decoder:**
  The component responsible for extracting the hidden message from the stego image.

2.5. Functional Flow of the Image Steganography

2.5.1. Encoding Process

- **Step 1:** Select a cover image where to hide the message.
- **Step 2:** Enter the secret message .
- **Step 3:** Selecting a steganography key .
- **Step 4:** Apply a steganography algorithm to embed the secret message into the cover image using the key.

2.5.2. Decoding Process:

- **Step 1:** Selecting the stego image containing the hidden message.
- **Step 2:** Selecting the key .
- **Step 3:** Apply the reverse process of the steganography algorithm to extract the secret message using the key.

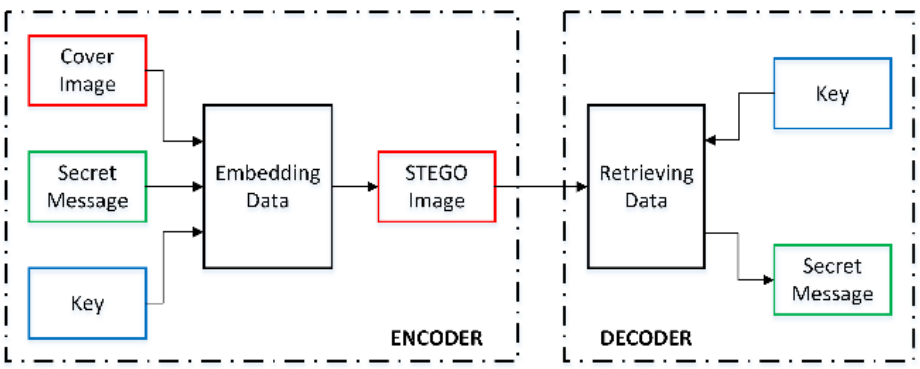


Figure 2.1: Basic Steganography system

source: researchgate.net/figure/Basic-steganography-system_fig1_321183782

2.6. Uses of steganography

Steganography uses include:

- **Avoiding censorship:**
  sending message without being censored, which means the intractability to their sender.
- **Digital watermarking:**
  Steganography is employed to create invisible watermarks within digital content which do not distort the original content but enable tracking to detect unauthorized use.
- **Securing information:**
  By embedding data within seemingly innocuous files, they can share critical data without arousing suspicion.Usually used by law enforcement and government agencies to transmit highly sensitive information.

3. Existing Solutions

The art of Steganography , has many tricks made over time . These ways are not the same in how hard they are , how much they can hide , how well they can stay hidden , and how good they are at not being found out . By knowing these things about each way, we can see when and where they are best to use .

3.1. Least Significant Bit (LSB) Embedding

The Least Significant Bit (LSB) method is a message hiding technique by inserting messages in the low or rightmost bit of the cover work file as a medium for hiding messages.[4]

To facilitate comprehension, the following steps outline the encoding process with the LSB algorithm:

- **Step 1:** Preparing the Cover Image:
  Analyzing The cover image as factors like image size and complexity .

- **Step 2:** Enter the secret message:
  Converting to binary the secret message, which can be text, another image, or a file.
- **Step 3:** Embedding the Secret Message (LSB) Modification:

  – Selecting how many least significant bits (LSBs) of each pixel will be modified to hold the message data.
  – Going through each pixel in the cover image one by one.
  – For each pixel, a chosen number of LSBs (based on embedding depth) are replaced with the corresponding bits from the binary representation of the secret message.

Here's an illustrative example (assuming 1 LSB embedding):

- **Original pixel value (8 bits):** 10010110 (binary)
- **Secret message bit to embed:** 1 (binary)
- **Modified pixel value (LSB replaced):** 10010111 (binary)

And here are some screenshots exemplifying the process of steganography, which I conducted using an online steganography tool [5]:

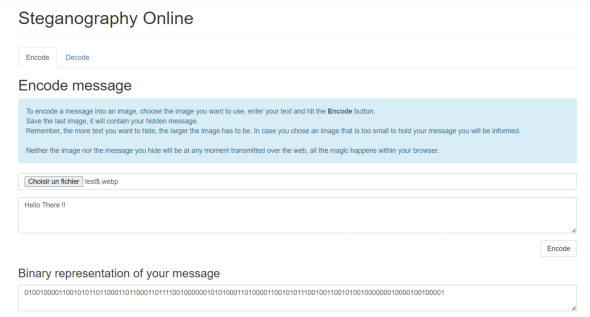
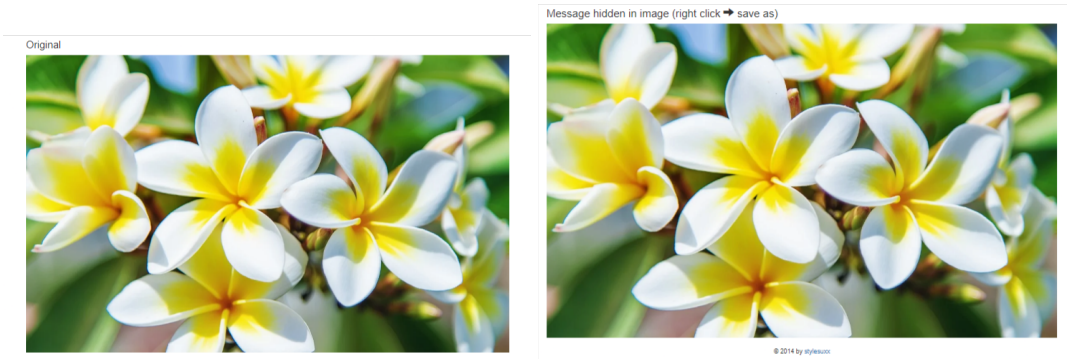


Figure 3.1: Enter the cover image and the secret message



(a) Original Image (b) Output Image

To demonstrate the LSB embedding technique in action, we provide a Python implementation utilizing the Stegano library. This implementation showcases how to encode a secret message into an image using the LSB method and subsequently decode the hidden message from the encoded image.
The following Python code snippet illustrates how the LSB embedding technique can be implemented using the Stegano library.

Code Snippet :

```
from stegano import lsb

def encode_lsb(carrier_image, secret_message):
    try:
        # Embed message and save the encoded image
        encoded_image = lsb.hide(carrier_image, secret_message)
        encoded_image.save("encoded_image.png")
        print("Image with hidden message saved successfully!")
    except (OSError, ValueError) as e:
        # Handle potential errors like file not found or message size
   exceeding capacity
        print(f"Error: {e}")
        return None
    return encoded_image  # Return the encoded image

def decode_lsb(encoded_image):
    try:
        # Extract the message
        decoded_message = lsb.reveal(encoded_image)
```

```
    except (IOError, ValueError) as e:
        # Handle potential errors like file not found or invalid image format
        print(f"Error: {e}")
        return "None"

# Example usage
secret_message = "This is a confidential message."
encoded_image = encode_lsb("C:/Users/amal1/OneDrive/Desktop/Fleur.jpg", secret_message)

if encoded_image:
    decoded_message = decode_lsb("encoded_image.png")

if decoded_message:
    print(f"Decoded Message: {decoded_message}")  # Print the decoded message
else:
    print("No message found in the stego-image.")

# Result
''' Image with hidden message saved successfully!
Decoded Message: This is a confidential message.'''
```

Strengths and Weaknesses

The LSB embedding technique offers several strengths, making it a popular choice for steganographic applications. However, it also has certain weaknesses that need to be considered.

Strengths

- **Simplicity:** The LSB embedding method is relatively simple to implement, requiring minimal computational resources.
- **Capacity:** It provides a high payload capacity for hiding data, especially in images with high bit-depth.
- **Robustness:** LSB embedding tends to maintain the visual quality of the cover image, making it less susceptible to visual detection by humans.

Weaknesses

- **Susceptibility to Attacks:** LSB embedding is vulnerable to statistical analysis and various steganalysis techniques, especially when the embedding depth is shallow.
- **Limited Security:** While LSB embedding can hide data effectively, it may not provide strong security against determined adversaries who employ advanced steganalysis methods.
- **Data Integrity:** Altering the LSBs of pixels can affect the integrity of the cover image, potentially causing unintended visual artifacts or corruption.

Despite its weaknesses, LSB embedding remains a popular choice for steganography due to its simplicity and high capacity. However, it's essential to consider these limitations when determining its suitability for specific applications.

Possible Enhancements

While the LSB embedding algorithm offers a simple and effective method for hiding data within images, there are several potential enhancements that could further improve its performance and security:

- **Embedding Depth Variation:** Implementing variable embedding depths across different regions of the cover image can enhance security by making it more challenging for adversaries to detect and extract hidden data. Adaptive embedding techniques can adjust the embedding depth based on the characteristics of the image and the importance of the data being concealed.
- **Error Correction Coding:** Incorporating error correction coding schemes, such as Reed-Solomon codes or Hamming codes, can improve the resilience of the steganographic system against data loss or corruption during transmission or storage. By adding redundancy to the encoded data, it becomes possible to recover the original message even if some bits are altered or lost.
- **Randomized Embedding Locations:** Randomizing the selection of embedding locations within the cover image can increase the security of the steganographic system by reducing patterns that may arise from deterministic embedding strategies. By spreading the hidden data across the image in a random manner, it becomes more challenging for adversaries to identify and extract the concealed information.
- **Enhanced Steganalysis Resistance:** Developing countermeasures against advanced steganalysis techniques can improve the robustness of the LSB embedding algorithm

against detection. This may involve incorporating additional obfuscation or distortion techniques to mask the presence of hidden data and confuse steganalyzers.

- **Integration with Encryption:** Combining steganography with encryption techniques can provide an additional layer of security by encrypting the data before embedding it into the cover image. This ensures that even if the hidden message is detected, it remains unintelligible without the corresponding decryption key.

By exploring these enhancements, it's possible to augment the capabilities and security of the LSB embedding algorithm, making it more suitable for a wider range of applications and scenarios.

3.2. Chaos-based Spread Spectrum Image Steganography (CSSIS)

Chaos-based Spread Spectrum Image Steganography (CSSIS) is a technique designed to elevate the security and imperceptibility of concealed messages within images. It employs chaotic encryption and modulation methodologies within spread spectrum technology to embed messages seamlessly into images.[3]
The fundamental goal of this method is to heighten the confidentiality and resilience of hidden information by leveraging the chaotic characteristics inherent in the encryption and modulation procedures.

To aid comprehension, these are the steps of the Chaos-based steganography process:

- **Step 1: Cover Image**
  - This is the original image that will be used to hide the secret message.
- **Step 2: Message**
  - The information you want to hide (the secret message) within the cover image.
- **Step 3: Chaotic Encryption**
  - The message undergoes chaotic encryption. Chaotic systems are used to scramble the message, making it harder to detect.
  - Key 1 influences the chaotic encryption process.

- **Step 4: ECC Encoder**
  - The encrypted message is encoded using an Error Correction Code (ECC) encoder. ECC helps ensure data integrity during transmission.
  - This step adds redundancy to the message, making it more robust against noise or errors.
- **Step 5: Modulation**
  - The encoded message is modulated. Modulation changes the signal's characteristics to embed it more effectively.
  - Key 2 is involved in the chaotic-shift keying process connected to modulation.
- **Step 6: Interleaving**
  - The interleaving process rearranges the bits of the message.
  - Key 3 plays a role in this step.
- **Step 7: Quantizer**
  - The interleaved message is quantized.
  - Quantization maps the continuous values to discrete levels suitable for embedding.
  - The result is the final stego-image, which contains the hidden message.

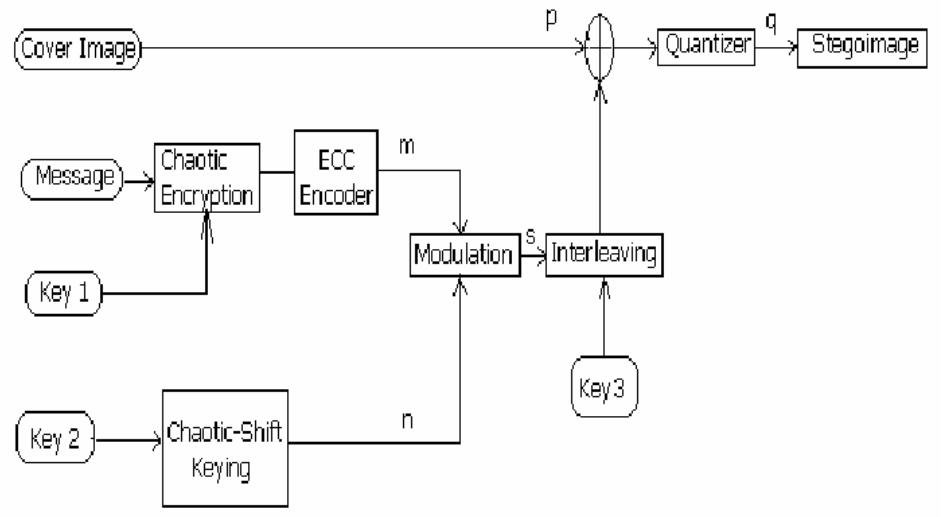


Fig. 1. CSSIS encoder.

Figure 3.3: CSSIS Process

Code Snippet :

```
import numpy as np
from PIL import Image

def encode(image_path, message, key):
    # Load the image
    image = Image.open(image_path)
    # Convert the image to a numpy array
    image_array = np.array(image)
    # Generate a pseudo-random sequence using chaos theory
    sequence = generate_sequence(key, len(message))
    # Convert the message to binary
    binary_message = ''.join(format(ord(i), '08b') for i in message)
    # Embed the message in the image using spread spectrum technique
    index = 0
    for i in range(len(image_array)):
        for j in range(len(image_array[0])):
            for k in range(len(image_array[0][0])):
                if index < len(binary_message):
                    image_array[i][j][k] = int('{0:b}'.format(
    image_array[i][j][k]).zfill(8)[:-2] +
                            binary_message[index] + binary_message[
    index + 1], 2) ^ int(sequence[index // 2])
                    index += 2
    # Save the modified image
    print("Image encoded successfully")
    modified_image = Image.fromarray(image_array)
    modified_image.save('modified_' + image_path)

def decode(image_path, key):
    # Load the image
    image = Image.open(image_path)
    # Convert the image to a numpy array
    image_array = np.array(image)
    # Generate the same pseudo-random sequence used for encoding
    sequence = generate_sequence(key, len(image_array.flat) * 4)
```

```
    # assuming 1 byte per channel
    # Extract the hidden message from the image using spread spectrum technique
    binary_message = ''
    index = 0
    for i in range(len(image_array)):
        for j in range(len(image_array[0])):
            for k in range(len(image_array[0][0])):
                if index < len(sequence):
                    binary_message += str((int('{0:b}'.format(image_array[i][j][k]).zfill(8)[-2:], 2) ^
                                           sequence[index]) & 1)
                    index += 1
    # Convert the binary message to text
    message = ''.join([chr(int(binary_message[i:i + 8], 2)) for i in range(0, len(binary_message), 8)])
    return message

def generate_sequence(key, length):
    # Use logistic map equation for generating pseudo-random sequence
    sequence = [key]
    for i in range(length - 1):
        sequence.append(4 * key * (1 - key))
        key = sequence[-1]
    return sequence

# Example usage
image_path = "Desktop/fleur.jpg"
key = 0.5
message = "Your hidden message"
encode(image_path, message, key)
decoded_message = decode('modified_' + image_path, key)
print('Hidden message:', decoded_message)

#Result
''' Image encoded successfully
Hidden message: Your hidden message '''
```

Strengths and Weaknesses

Strengths

- **Security:** Chaos-based spread spectrum image steganography offers high security due to its complex encryption techniques, making it resistant to traditional steganalysis methods.
- **Capacity:** This method provides a significant payload capacity for hiding data within images, allowing for the concealment of large amounts of information.
- **Robustness:** Chaos-based spread spectrum steganography tends to maintain the visual quality of the cover image, making it difficult for unauthorized parties to detect the presence of hidden data.
- **Resistance to Statistical Analysis:** By utilizing chaotic functions for embedding, this technique is less susceptible to statistical analysis compared to conventional steganographic methods, enhancing its security against detection.

Weaknesses

- **Complexity:** Implementing chaos-based spread spectrum steganography requires a deeper understanding of chaos theory and signal processing techniques, making it more complex to implement compared to simpler methods like LSB embedding.
- **Computational Overhead:** The encryption and embedding processes involved in chaos-based spread spectrum steganography may require significant computational resources, leading to higher processing times.
- **Key Management:** Managing encryption keys securely is crucial for the effectiveness of chaos-based spread spectrum steganography, and any compromise in key management could lead to the exposure of hidden data.
- **Sensitivity to Parameters:** Chaotic systems are sensitive to initial conditions and parameter settings, which could affect the robustness and security of the steganographic system if not properly configured.

Despite its strengths in security and capacity, chaos-based spread spectrum image steganography faces challenges related to complexity, computational overhead, key management, and sensitivity to parameters. However, with proper implementation and management, it can offer robust concealment of data within images, particularly in scenarios where high security is paramount.

Possible Enhancements

To further enhance the performance and security of chaos-based spread spectrum image steganography, several enhancements can be considered:

- **Dynamic Parameter Adaptation:** Implementing algorithms that dynamically adjust chaotic system parameters based on image characteristics and security requirements can improve robustness and resistance against attacks.
- **Key Management Protocols:** Developing robust key management protocols, including key generation, distribution, and revocation mechanisms, can enhance the security of chaos-based spread spectrum steganography systems.
- **Adaptive Embedding Techniques:** Introducing adaptive embedding techniques that adjust the embedding strength and locations based on the content of the cover image and the importance of the hidden data can improve security and minimize visual distortion.
- **Multilayered Encryption:** Integrating multiple layers of encryption, including both chaotic encryption and conventional cryptographic algorithms, can enhance the security of hidden data and provide defense against various attacks.
- **Machine Learning Defenses:** Leveraging machine learning algorithms to develop defensive mechanisms against advanced steganalysis techniques can further strengthen the resilience of chaos-based spread spectrum steganography.

By incorporating these enhancements, chaos-based spread spectrum image steganography can become even more robust and secure, expanding its applicability in various sensitive communication and data protection scenarios.

3.3. Bit-Plane Complexity Segmentation (BPCS)

Bit-Plane Complexity Segmentation is also a way to hide secret informations that is based on dividing the image into the informative region and the noise-like region.

BPCS steganography, initially proposed by Kawaguchi and Eason, introduces a distinct approach. The fundamental concept involves dividing the cover image into ”informative regions” and ”noise-like regions.” Subsequently, secret information is concealed within the noise-like blocks of the cover image. In contrast to the LSB technique, where data is embedded solely in the lowest bit-plane, BPCS entails embedding data across pixel blocks spanning

all planes, from the highest (most significant bit, MSB plane) to the lowest (LSB plane), each exhibiting noisy patterns. In BPCS, a gray image comprising n-bit pixels can be decomposed into n-binary planes.

To aid comprehension, these are the detailed steps of this algorithm :

Embedding:

- **Message Conversion:** The secret message is converted into a binary string. Each character in the message is represented by its ASCII code, which is then converted to an 8-bit binary string.
- **Space Check:** The code checks if there's enough space in the image to embed the message. BPCS modifies the Least Significant Bit (LSB) of pixels, so the message size shouldn't exceed a significant portion (around 50%) of the image's total bits.
- **Bit Splitting:** The binary message is split into individual bits.
- **Image Preprocessing:** The image is converted from RGB (Red, Green, Blue) format to a NumPy array for easier manipulation. The array is then flattened into a 1D list for sequential processing of pixels.
- **LSB Modification:** The code iterates through each element (pixel) in the flattened image array. It retrieves the LSB of the current pixel value.
  - If the LSB needs to be modified (based on the corresponding secret message bit), the code:
    - Increments the pixel value by 1 if the secret message bit is 1 and the LSB is 0 (to set the LSB to 1).
    - Decrements the pixel value by 1 if the secret message bit is 0 and the LSB is 1 (to set the LSB to 0).
  - Essentially, the LSB of the pixel is overwritten with the corresponding bit from the secret message.
- **Image Reshaping:** After all the message bits are embedded, the modified flattened array is reshaped back to its original 2D (or 3D for RGB) form to represent the stego-image (image with the secret message hidden).

Extracting:

- **Image Preprocessing:** Similar to embedding, the stego-image is converted to a NumPy array and flattened.
- **LSB Reading:** The code iterates through each element in the flattened array, retrieving the LSB of the current pixel value.
- **Bit Collection:** The LSBs of each pixel are collected and stored in a list, essentially reconstructing the hidden binary message.
- **Binary to String Conversion:** The collected LSBs (now representing the hidden message bits) are joined to form a binary string.
- **Character Conversion:** The binary string is split into groups of 8 bits (representing the ASCII code of a character) and converted back to characters, revealing the secret message.

Code Snippet :

```
1
2  import numpy as np
3  from PIL import Image
4
5  def embed_message_in_bpc(original_image, secret_message):
6      # Convert the secret message to a binary string.
7      binary_message = ''.join(format(ord(char), '08b') for char in
       secret_message)
8
9      # Get image dimensions and message length.
10     image_height, image_width, channels = original_image.shape
11     message_length = len(binary_message)
12
13     # Check if there's enough space in the image for the message.
14     if message_length > image_height * image_width * 0.5:
15       raise ValueError("Insufficient space in image to embed message.")
16
17     # Split the message into bits and convert them to integers.
18     message_bits = [int(bit) for bit in binary_message]
```

```
19
20      # Flatten the image into a 1D array for easier processing.
21      flat_image = original_image.flatten()
22
23      # Embed the secret message one bit at a time using BPCS.
24      for i in range(message_length):
25          # Get the LSB (Least Significant Bit) of the current pixel.
26          lsb = flat_image[i] & 1
27
28          # Modify the LSB to match the secret message bit.
29          if message_bits[i] == 0 and lsb == 1:
30              flat_image[i] -= 1
31          elif message_bits[i] == 1 and lsb == 0:
32              flat_image[i] += 1
33
34      # Reshape the modified image data back to its original form.
35      stego_image = flat_image.reshape(image_height, image_width,
    channels)
36
37      return stego_image
38
39  def extract_message_from_bpc(stego_image):
40      # Flatten the stego-image into a 1D array for easier processing.
41      flat_image = stego_image.flatten()
42
43      # Extract the secret message bits one at a time using BPCS.
44      extracted_bits = []
45      for pixel in flat_image:
46          # Get the LSB (Least Significant Bit) of the current pixel.
47          lsb = pixel & 1
48          extracted_bits.append(lsb)
49
50      # Convert the extracted bits back to a binary string.
51      binary_message = ''.join(str(bit) for bit in extracted_bits)
52
53      # Convert the binary string to a secret message string.
54      secret_message = ''.join(chr(int(binary_message[i:i+8], 2)) for i
    in range(0, len(binary_message), 8))
```

```
    return secret_message

# Example usage:

# Load the original image using Pillow
original_image = Image.open("BPCS.jpg").convert("RGB")
original_image_array = np.array(original_image)
# Convert to a NumPy array

# Define the secret message
secret_message = "This is a secret message"

# Embed the message in the image using BPCS
stego_image_array = embed_message_in_bpc(original_image_array,
    secret_message)

# Convert the stego image back to a Pillow image
stego_image = Image.fromarray(stego_image_array.astype("uint8"), "RGB")

# Save the stego image using Pillow
stego_image.save("stego_image.png")

# Extract the message from the stego image
extracted_message = extract_message_from_bpc(stego_image_array)

# Print the extracted message
print(extracted_message)  # Should print "This is a secret message"
```

Strengths and Weaknesses

Strengths

- **Simplicity:** BPCS is comparatively easy to implement. It requires little alteration of the original image and does not involve complicated mathematical operations.
- **Low Computational Cost:** Because of its simplicity, BPCS has a low computation cost. Embedding messages and extracting them using BPCS are relatively fast compared with more sophisticated steganographic approaches.

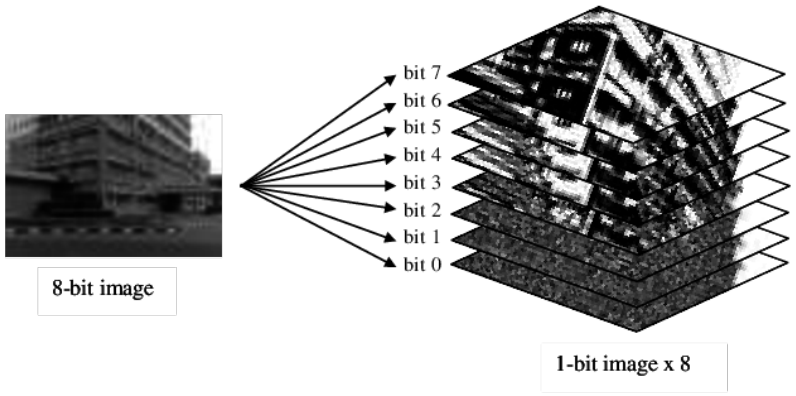


Figure 3.4: BPCS Decomposition Procss

- **Some Detectability Resistance:** In some situations, especially in poor quality or high noise images, BPCS can possibly carry out hiding of messages havingsome resistance to basic types of steganalysis.

Weaknesses

- **Low embedding capacity:** This image hiding program only changes the least significant bit of each pixel thereby limiting the data that can be hidden. Hence, it cannot conceal large messages.
- **High detectability:** Changes in the least significant bits are statistically identifiable, particularly by techniques such as steganalysis which statistically examine properties manifested in the image.
- **Low security:** BPCS has weak security for confidential communication because of its low embedding capacity and high detectability. If anyone suspects that there is a steganography going on and uses a tool for detecting this kind of activities, he/she will find your text without any difficulties.
- **Susceptible to noise:** The BPCS systems are susceptible to operations like noise manipulation. Consequently, if there is any alteration of the LSBs during compression or other related processes with regard to stego-image, then the embedded message can be distorted or lost altogether.

To sum up, BPCS steganography provides a low grade platform for hiding confidential information. While it can be helpful for simple applications where safety is not a major concern, it is not advised for circumstances that need tight security or large quantities of data to be concealed.

3.4. Batch Steganography

Batch steganography studies how to distribute payload across a group of images based on the distortion definition and STC embedding of single image steganography. [1]

These are the steps of the Batch image steganography process :
Step 1: Preprocessing:

- **Secret Message Preparation:** Similar to single image steganography, the secret message needs to be converted into a format suitable for embedding, likely binary (a string of 0s and 1s).
- **Image Selection:** A set of digital images (batch) is chosen to act as carriers for the hidden message. Factors like image size, quality, and total capacity will be considered.

Step 2: Message Distribution and Embedding:

- **Secret Sharing (Optional):** The message might be fragmented into multiple parts for added security. Each image in the batch would then receive a specific fragment.
- **Distortion Budget Allocation:** An algorithm determines how much distortion (changes introduced while hiding data) is acceptable for each image in the batch. This ensures a balance between hiding capacity and maintaining image quality.
- **Stego Image Creation:** Each image in the batch undergoes an embedding process based on a chosen steganographic method (like STC) while adhering to the allocated distortion budget. This results in a stego image containing its assigned portion of the secret message.

Step 3: Post-Processing:

- **Batch Validation (Optional):** The stego images might be checked to ensure successful embedding and compliance with distortion limits.
- **Delivery/Storage:** The stego image collection is prepared for secure delivery or storage.

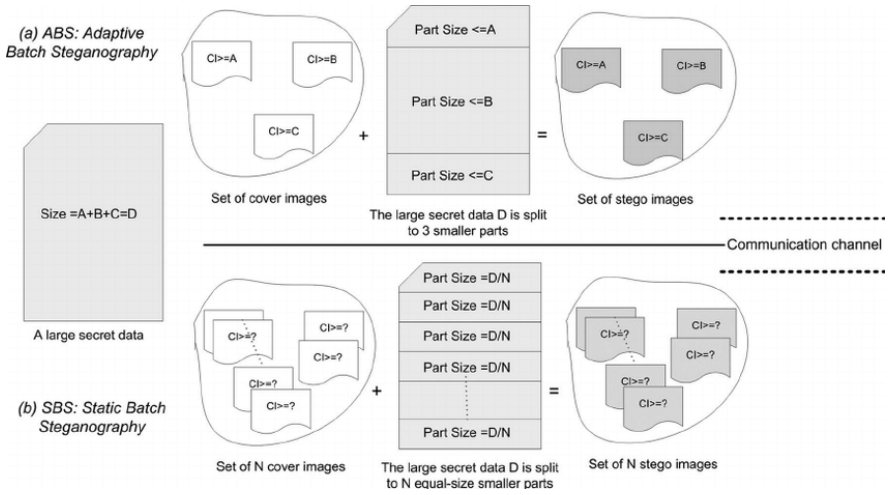


Figure 3.5: Batch Steganography process
source: `https://www.sciencedirect.com/`

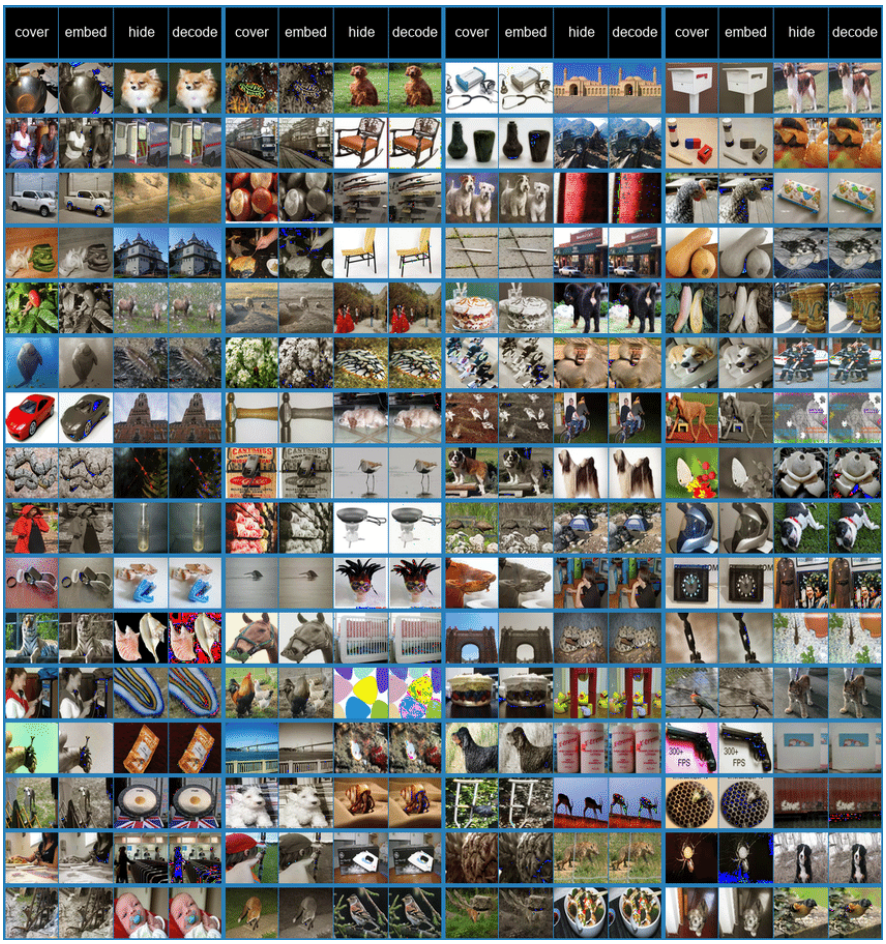


Figure 3.6: Example
source: `https://www.researchgate.net//`

Code Snippet:

```
import numpy as np
from PIL import Image
def embed_message_in_bpc(original_image, secret_message):
  # Convert the secret message to a binary string.
  binary_message = ''.join(format(ord(char), '08b') for char in
   secret_message)
  # Get image dimensions and message length.
  image_height, image_width, channels = original_image.shape
  message_length = len(binary_message)
  # Check if there's enough space in the image for the message.
  if message_length > image_height * image_width * 0.5:
    raise ValueError("Insufficient space in image to embed message.")
  # Split the message into bits and convert them to integers.
  message_bits = [int(bit) for bit in binary_message]
  # Flatten the image into a 1D array for easier processing.
  flat_image = original_image.flatten()
  # Embed the secret message one bit at a time using BPCS.
  for i in range(message_length):
    # Get the LSB (Least Significant Bit) of the current pixel.
    lsb = flat_image[i] & 1
    # Modify the LSB to match the secret message bit.
    if message_bits[i] == 0 and lsb == 1:
      flat_image[i] -= 1
    elif message_bits[i] == 1 and lsb == 0:
      flat_image[i] += 1
  # Reshape the modified image data back to its original form.
  stego_image = flat_image.reshape(image_height, image_width, channels)
  return stego_image


def extract_message_from_bpc(stego_image):
  # Flatten the stego-image into a 1D array for easier processing.
  flat_image = stego_image.flatten()
  # Extract the secret message bits one at a time using BPCS.
  extracted_bits = []
  for pixel in flat_image:
```

```
    # Get the LSB (Least Significant Bit) of the current pixel.
    lsb = pixel & 1
    extracted_bits.append(lsb)
  # Convert the extracted bits back to a binary string.
  binary_message = ''.join(str(bit) for bit in extracted_bits)
  # Convert the binary string to a secret message string.
  secret_message = ''.join(chr(int(binary_message[i:i+8], 2)) for i in
   range(0, len(binary_message), 8))
  return secret_message

def embed_messages_in_batch(image_paths, messages):
  stego_images = []
  for i,(image_path, message) in enumerate(zip(image_paths, messages)):
    # Load the image
    original_image = Image.open(image_path).convert("RGB")
    original_image_array = np.array(original_image)
    # Embed the message
    stego_image_array = embed_message_in_bpc(original_image_array,
   message)
    # Append the stego image to the list
    stego_images.append(stego_image_array)
  return stego_images


# Example usage (replace with your actual image paths and messages)
image_paths = ["path/to/image1.jpg", "path/to/image2.png", ...]
messages = ["This is a secret message for image 1", "Another secret
   message for image 2", ...]

# Embed messages in batch (key argument not implemented for this
   example)
stego_image_arrays = embed_messages_in_batch(image_paths, messages)

# Save the stego images (replace with your saving logic)
for i, stego_image_array in enumerate(stego_image_arrays):
  stego_image=Image.fromarray(stego_image_array.astype("uint8"), "RGB")
  stego_image.save(f"stego_image_{i}.png")
```

Strengths and Weaknesses

Strengths

- **More Space:** Using many pictures makes the hiding space much bigger. You can hide a lot more data than using just one picture.
- **Stronger:** Spreading the secret message over many images makes the hidden data tougher. Even if some images are lost, broken, or found by steganalysis, there's a chance to get the message back from the others (if they're not all affected). This extra layer makes it more secure.
- **Faster:** With batch steganography, embedding tasks can be done at the same time on different pictures, which might make it quicker, depending on how it's set up.

Weaknesses

- **Complexity:** Compared to single image steganography, batch methods involve more complex algorithms to manage message distribution, distortion allocation across images, and potentially secret sharing.
- **Distortion Management:** aintaining acceptable visual quality across all images in the batch becomes more critical. The embedding process needs to be carefully controlled to ensure the overall distortion introduced remains within acceptable limits. This can be more challenging compared to a single image.
- **Increased Detection Risk:** Using multiple images might increase the chances of steganalysis techniques identifying anomalies, especially if the embedding method introduces similar patterns across the batch. Careful selection of steganographic methods and message distribution strategies is crucial.
- **Storage and Management:**Storing and managing a collection of stego images can be more complex compared to a single image, especially if secret sharing is involved.

Overall, Group hiding data, called batch steganography, is a strong way to hide more data and be more strong. But, it needs careful thought about the added difficulty, picture changes, and finding risks. The pick between hiding in one picture or in groups depends on what is needed, thinking about the data size, security needs, and picture quality rules.

3.5. Permutation Steganography

Image permutation steganography is a technique used to hide secret data within an image file by rearranging its pixels in a specific way. It's a twist on traditional steganography methods that modify the pixel data itself.

You can see as follows, the steps of this unique technique :

Hiding the Secret Message:

- **Step 1: Secret Message Preparation:** Convert the secret message into a binary format (a string of 0s and 1s). This allows the message to be embedded within the image data.
- **Step 2: Cover Image Selection:** Choose a digital image to act as the carrier for the hidden message. This is called the cover image. The quality and size of the image will impact the amount of data you can hide and the potential for distortion.
- **Step 3: Pixel Blocking:** Divide the cover image into smaller blocks. These blocks will be the units where the pixel rearrangement happens. The size of these blocks can affect the amount of data hidden and the potential for noticeable changes in the image.
- **Step 4: Permutation Algorithm:** Define a method for rearranging the pixels within each block. This algorithm will depend on the secret key and the message bits.
  - The secret key: This is a crucial piece of information needed to reverse the permutation later. It should be complex and kept confidential.
  - Message bits: The binary representation of the secret message can be used to influence the specific rearrangement pattern within each block.
- **Step 6: Pixel Shuffling:** Implement the permutation algorithm on each block. Based on the secret key and the message bits, rearrange the pixels within each block according to the defined pattern.
- **Step 7: Stego Image Creation:** After all the blocks are processed, create a new image with the rearranged pixels. This new image, containing the hidden message, is called the stego image.

Retrieving the Secret Message:

- **Step 1: Stego Image and Secret Key:** The recipient needs the original stego image and the same secret key used for hiding the message.

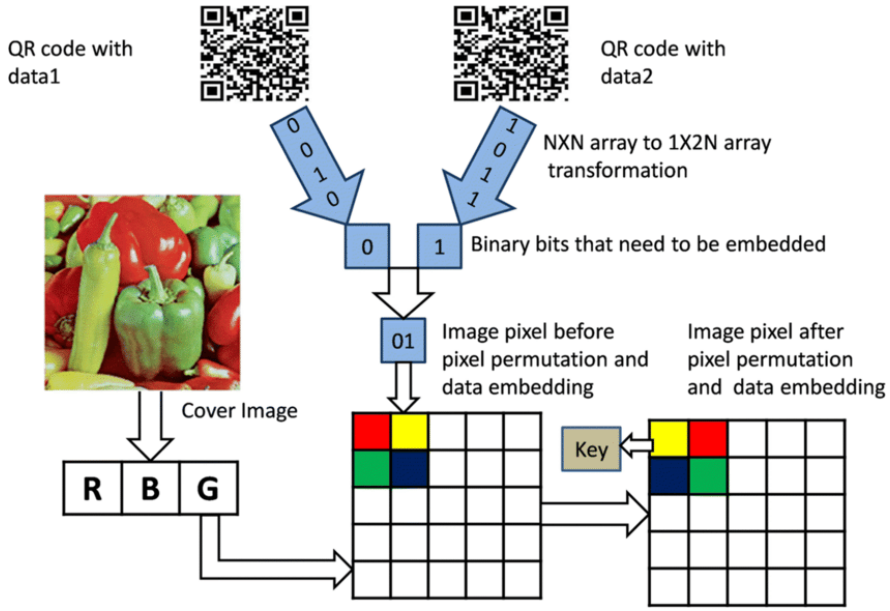


Figure 3.7: Permutation based Steganography
source: www.researchgate.net

- **Step 2: Reverse Permutation:** With the secret key, apply the reverse of the permutation algorithm used earlier on each block in the stego image. This will reorder the pixels back to their original positions, revealing the hidden message bits.
- **Step 3: Binary to Message Conversion:** Once the message bits are retrieved, convert them back from binary format to the original secret message.

Code Snippet :

```
from PIL import Image

def encode_message(image_path, message, secret_key):
    # Open the image and convert to RGB mode
    image = Image.open(image_path).convert("RGB")
    width, height = image.size

    # Define block size(adjust for message size and quality preference)
    block_size = 8

    # Split message into bits
    message_bits = ''.join(format(ord(char),'08b') for char in message)

```

```
    # Prepare empty list for stego image data
    stego_data = []

    # Loop through image blocks
    for y in range(0, height, block_size):
        for x in range(0, width, block_size):
            # Get current block
            block = image.crop((x, y, x + block_size, y + block_size))
            block_data = list(block.getdata())

            # Check if there are still bits left in the message_bits string
            if len(message_bits) > 0:
                # Permute block based on secret key and message bits (replace with your permutation logic)
                # Here's a simple example using key modulo block size
                permutation = [i for i in range(block_size)]
                for i in range(block_size):
                    permutation[i], permutation[(i + int(message_bits[0])) % block_size] = permutation[(i + int(message_bits[0])) % block_size], permutation[i]
                message_bits = message_bits[1:]

                #Remove used message bit

                # Apply permutation to block data
                stego_block_data = [block_data[i] for i in permutation]

                # Add permuted block data to stego image data
                stego_data.extend(stego_block_data)
            else:
                # If there are no more message bits, simply add the block data without permutation
                stego_data.extend(block_data)

    # Create a new image with stego data
    stego_image = Image.new(image.mode, image.size)
    stego_image.putdata(stego_data)
```

```
    print("Image encoded successfully")
    return stego_image

def decode_message(stego_image_path, secret_key):
    # Open the stego image and convert to RGB mode
    stego_image = Image.open(stego_image_path).convert("RGB")
    width, height = stego_image.size
    block_size = 8  # Same block size as used during encoding

    # Prepare empty string for message
    message = ""

    # Loop through stego image blocks
    for y in range(0, height, block_size):
        for x in range(0, width, block_size):
            # Get current block
            block = stego_image.crop((x, y, x + block_size, y +
    block_size))
            block_data = list(block.getdata())

            # Reverse permutation based on secret key (replace with
        your reverse logic)
            permutation = [i for i in range(block_size)]
            for i in range(block_size - 1, 0, -1):
                permutation[i], permutation[permutation[i]] =
    permutation[permutation[i]], i

            # Apply reverse permutation to block data
            message_bits = ''.join([str(int(block_data[permutation[i
    ]][0] % 2)) for i in range(block_size)])
            message += chr(int(message_bits, 2))

    return message

# Example usage (replace with your image paths and message)
secret_message = "This is a secret message"
secret_key = "your_secret_key"

```

```
encoded_image = encode_message("image path",secret_message, secret_key)
      # No .encode()
encoded_image.save("stego_image.jpg")

decoded_message = decode_message("stego_image.jpg", secret_key)
print("Decoded Message:", decoded_message)

# Result
''' Image encoded successfully
Decoded Message: This is a secret message '''
```

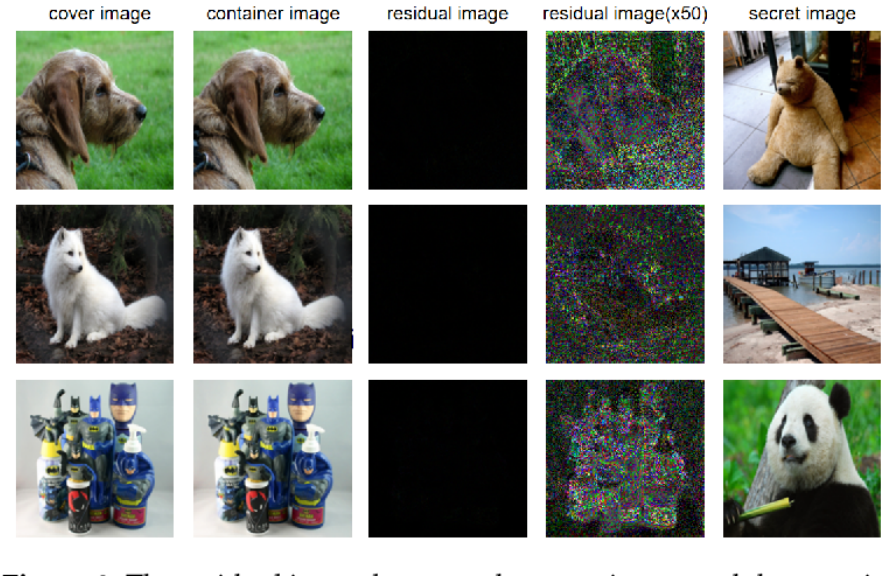


Figure 3.8: Example
source: ww.semanticscholar.org

Strengths and Weaknesses

Strengths

- **Security:** Permutation can offer enhanced security compared to some simpler steganographic methods. The complex pixel rearrangement makes it more challenging to detect the hidden data using traditional steganalysis techniques.
- **Capacity:** This technique has the potential to hide a reasonable amount of data within the image by strategically shuffling pixel blocks. However, the capacity depends on factors like block size and the cover image's characteristics.
- **Reduced Distortion:** By manipulating pixel order rather than values, permutation can

potentially maintain a good visual quality of the stego-image (image containing hidden data) compared to modification-based methods.

Weaknesses

- **Potential Detectability:** Extensive pixel shuffling might introduce subtle visual artifacts, especially in areas with low complexity. Careful selection of permutation algorithms and block sizes is crucial to minimize these distortions.
- **Reliance on Secret Key:** The security of permutation steganography hinges on the secrecy of the key used for rearranging pixels. If the key is compromised, anyone can easily extract the hidden message by reversing the permutation.
- **Limited Error Correction:** Permutation itself doesn't offer inherent error correction capabilities. Corrupted pixels during transmission or storage can potentially damage the hidden message.

Permutation-based steganography presents an alternative approach for data hiding within images. It offers advantages in terms of security and potentially better visual quality compared to some traditional methods. However, it's essential to consider the trade-offs between data capacity, potential for distortion, and the robustness of the steganographic system.

4. System Overview and Functionality

In this section, I will provide a detailed examination of PixHide System, outlining its design, constituent components, operational processes, functionalities, and implementation approaches. The discussion will encompass various system aspects, including design principles, component overviews, functional operation, data transmission, user interface, applied algorithms, and utilized techniques.

4.1. System Design and Architecture

PixHide is built on a layered architecture prioritizing modularity, scalability, and security. The system's core is the image processing module, which manipulates image data for embedding and extracting hidden information based on the needs of the clients. A variety of

steganography algorithms can be integrated for data concealment. The user interface is web-based, enabling easy access to encoding and decoding features. The system also includes encryption mechanisms for data protection and rigorous testing for reliability.

4.2. Components Overview

This section delves into the individual elements that constitute the Image Steganography System, providing insight into their functionalities and interactions within the system architecture.

1. **Image Processing Module**: - The image processing module serves as the backbone of the system, responsible for handling image data and facilitating the embedding and extraction of hidden information. This component employs various image manipulation techniques to ensure seamless integration of steganographic data while preserving the visual integrity of the carrier image.

2. **Steganography Algorithms**: - The steganography algorithms component encompasses a diverse range of techniques mentioned earlier that are employed to conceal data within images. These algorithms are designed to exploit imperceptible modifications in pixel values or image attributes, ensuring covert transmission of information. The system offers flexibility in selecting and configuring different steganographic methods to suit user preferences and security requirements.

3. **User Interface**: - The user interface component provides a user-friendly platform for interacting with the Image Steganography System. Through a web-based interface, users can access encoding and decoding functionalities, upload images, customize steganographic parameters, and visualize the results in real-time. The interface prioritizes simplicity, intuitiveness, and accessibility to accommodate users with varying levels of technical expertise.

4. **Data Management Module**: - The data management module handles the storage, retrieval, and manipulation of encoded messages within the system. This component ensures efficient organization and retrieval of steganographic data, enabling seamless integration with other system functionalities. Additionally, the data management module incorporates encryption mechanisms to safeguard sensitive information and mitigate risks associated with unauthorized access.

5. **Security Features**: - Security features are integrated throughout the system to safeguard against potential threats and vulnerabilities. Encryption techniques are employed to protect data integrity and confidentiality during transmission and storage. Furthermore, authentica-

tion mechanisms such as Google OAuth2.0 and access controls are implemented to restrict unauthorized access to system resources and functionalities.

By leveraging these components in concert, the Image Steganography System offers a comprehensive and robust platform for concealing and retrieving confidential information within digital images, ensuring privacy and security in sensitive communication scenarios.

4.3. Functionality and Workflow

This section elucidates the operational aspects of PixHide Image Steganography System, delineating the processes involved in encoding and decoding hidden information within digital images. Additionally, it outlines the user interactions and system behaviors that govern the overall workflow of the application.

1. **Algorithm Selection Process**: - The algorithm selection process is a critical aspect of the PixHide Image Steganography System, designed to tailor the steganographic approach to the specific needs and preferences of the user. As PixHide operates as a web-based steganography platform, it employs an intelligent algorithm selection mechanism to determine the most suitable technique based on various factors such as security requirements, resistance to statistical analysis, simplicity, robustness, computational cost, distortion, efficiency, and available space for embedding data.

PixHide utilizes a repository of steganography algorithms, each optimized for specific use cases and scenarios. These algorithms may include Least Significant Bit (LSB) embedding, Spread Spectrum Image Steganography (SSIS), Batch Steganography, and others. Each algorithm is characterized by its strengths and weaknesses, making it suitable for different applications and requirements.

Based on the user's input and system analysis, PixHide employs a decision-making algorithm or heuristic to select the most appropriate steganographic technique for the given scenario. This decision considers factors such as the desired level of security, the available computational resources, the tolerance for image distortion, and the efficiency of data embedding and extraction.

The algorithm selection process in PixHide aims to optimize the balance between security, efficiency, and usability, ensuring that the hidden information remains secure while minimizing the impact on the visual quality of the carrier image. By dynamically adapting the steganographic technique to the specific requirements of each encoding session, PixHide

enhances the effectiveness and versatility of its image concealment capabilities.

2. **Encoding Process**: - The encoding process entails embedding hidden data within a digital image using steganographic techniques. Users initiate the process by selecting an image as the carrier and specifying the message or data to be concealed. The system then applies the chosen steganography algorithm to adjust specific image properties or pixel values, effectively hiding the information within the image. Upon completion, users launch the encoded image containing the concealed data.

3. **Decoding Process**: - The decoding process involves extracting hidden information from an encoded image without altering its visual appearance. Users initiate the process by uploading the encoded image to the system, which then applies reverse steganographic techniques to recover the concealed data. The system analyzes the image properties or pixel values to identify and extract the embedded information, presenting it to the user in a human-readable format.

4. **User Interactions**: - User interactions with the Image Steganography System are facilitated through a user-friendly web interface. Users can seamlessly navigate between encoding and decoding functionalities, upload images, customize steganographic parameters, and visualize the results in real-time. The interface prioritizes simplicity and intuitiveness to enhance user experience and facilitate efficient communication of hidden information.

5. **System Behaviors**: - The Image Steganography System exhibits specific behaviors and functionalities to ensure seamless operation and reliable performance. These include error handling mechanisms to address input validation errors or system failures, feedback mechanisms to provide users with informative messages and notifications, and performance optimizations to enhance the efficiency of encoding and decoding processes.

By elucidating the functionality and workflow of the system, I hope this section has provided users with a comprehensive understanding of the processes involved in concealing and retrieving hidden information within digital images. Highlighting the user interactions and system behaviors that contribute to the overall usability and effectiveness of the application.

Here I have included a use case diagram that illustrates the primary actors, use cases, and their interactions within the PixHide system. This diagram provides a visual representation of the system's functionality and user interactions, enhancing understanding and clarity.

Figure 4.1: Pixhide Use Case Diagram

4.4. Data Exchange and Security Measures

The Data Exchange and Security Measures section delves into the mechanisms employed by the Image Steganography System to ensure secure transmission and storage of concealed information. This includes encryption techniques, authentication protocols, and access controls implemented to safeguard sensitive data from unauthorized access or interception during transmission.

1. **Encryption Techniques**:

   - To protect the confidentiality and integrity of concealed information during transmission and storage, PixHide employs robust encryption techniques. Data is encrypted using industry-standard cryptographic algorithms such as AES (Advanced Encryption Standard) or RSA (Rivest-Shamir-Adleman) before embedding it into the carrier image. This ensures that even if the encoded image is intercepted, the hidden data remains secure and inaccessible without the appropriate decryption key.

2. **Authentication Protocols**:

    - PixHide integrates authentication protocols to verify the identity of users and ensure that only authorized individuals have access to the system's functionalities. Authentication mechanisms such as Google OAuth2.0 or username/password authentication may be employed to authenticate users before granting them access to encoding and decoding features. This helps prevent unauthorized access to sensitive data and system resources.

3. **Access Controls**:

    - Access controls are implemented within the PixHide system to regulate user permissions and restrict unauthorized access to sensitive functionalities or data. This ensures that only authorized users can perform actions such as uploading images, encoding data, or accessing encoded messages.

4. **Secure Data Transmission**:

    - PixHide prioritizes secure data transmission protocols to prevent interception or tampering of information during communication between the user's device and the server. HTTPS (Hypertext Transfer Protocol Secure) is employed to encrypt data exchanged between the client's web browser and the PixHide server, ensuring that sensitive information, including uploaded images and encoded messages, remains confidential and protected from eavesdropping attacks.

5. **Data Integrity Checks**:

    - To ensure the integrity of concealed information, PixHide incorporates data integrity checks into its encoding and decoding processes. Hash functions or digital signatures may be utilized to generate checksums or signatures of the encoded data before and after transmission. These checksums or signatures are compared at the receiving end to verify the integrity of the transmitted information and detect any unauthorized alterations or tampering attempts.

By implementing these data exchange and security measures, PixHide enhances the confidentiality, integrity, and availability of concealed information, ensuring that sensitive data remains protected throughout the encoding, transmission, and decoding processes.

4.5. Implementation Details

In this section I will delve into the technical aspects of developing and deploying the Image Steganography System. This encompasses the programming languages, frameworks, libraries, and tools utilized in the system's implementation, as well as any notable design patterns or architectural considerations guiding the development process.

Programming Languages and Technologies

PixHide is primarily developed using a combination of frontend and backend technologies to create a robust and user-friendly steganography platform.Furthermore,to ensure scalability and data storage requirements, PixHide utilize relational databases for persistent storage of user data, encoded messages, and system configurations.

Architectural Considerations

The architecture of PixHide is designed to be modular, scalable, and maintainable, facilitating future enhancements and updates.
PixHide follows a layered architecture pattern, separating concerns between presentation, business logic, and data access layers. This promotes modularity and encapsulation, allowing for independent development and testing of each component.Also, as scalability and flexibility are paramount, PixHide will adopt a microservices architecture, decomposing the application into smaller, independent services that communicate via lightweight protocols. This enables better resource utilization, fault isolation, and deployment agility.
Containerization technologies will be also utilized to package the application and its dependencies into lightweight, portable containers, ensuring consistency and reproducibility across different environments.
By adhering to these implementation details and architectural considerations, PixHide aims to deliver a robust, scalable, and secure steganography platform that meets the needs of its users while remaining adaptable to evolving technological trends and requirements.

4.6. User Interface and User Experience (UX)

The User Interface and User Experience (UX) is a crucial part of the design principles and user-centric features incorporated into the Image Steganography System's interface. This includes considerations such as layout design, navigation structure, visual aesthetics, and interactive elements aimed at enhancing user satisfaction, engagement, and usability.

A prototype of the PixHide user interface was developed to visualize the design concepts and user interactions. The prototype features a simplified version of the encoding and decoding workflows, showcasing the layout, navigation structure, and interactive elements envisioned for the final system.

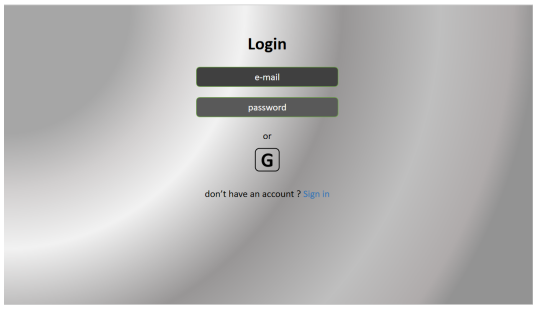


Figure 4.2: Login Page

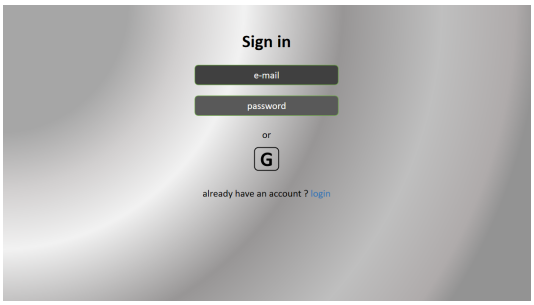


Figure 4.3: Sign in Page

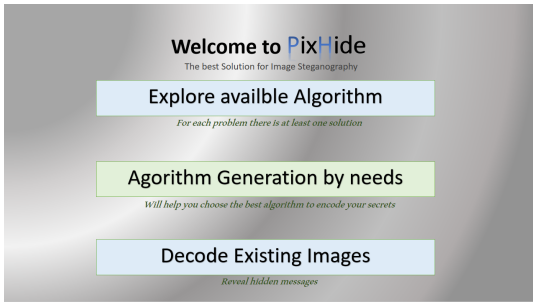


Figure 4.4: Welcome Page

Figure 4.5: Available Solutions Page

Figure 4.6: Needs Selection Page

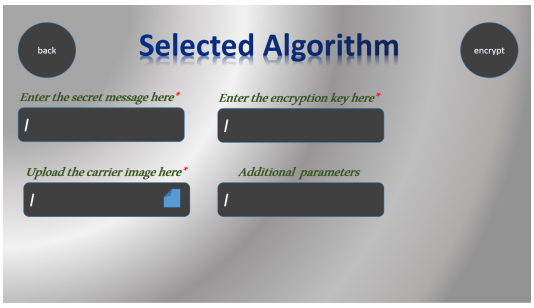


Figure 4.7: Encryption Page

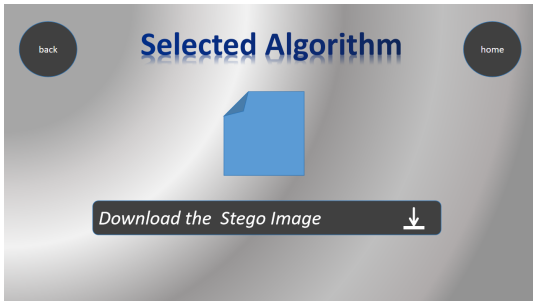


Figure 4.8: Stego Image Downloading Page

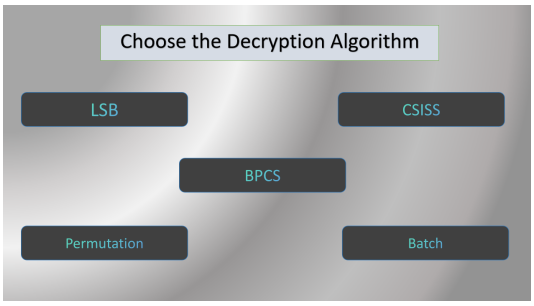


Figure 4.9: Decryption Algorithm Selection Page

Figure 4.10: Decryption Page

5. Tools and Development Phases

5.1. Phase 1: Planning and Requirements Gathering

1. **Identifying Clients Requirements**: Trying to understand the needs for data concealment and retrieval, as well as the preferences for user experience.
2. **Research on Steganography Techniques**: Explore various steganography techniques provided by the `stegano` library to understand their strengths and weaknesses in different scenarios.
3. **Selection of Encryption Libraries**: Assess the suitability of encryption techniques provided by libraries such as `numpy` and `PIL` for securing the hidden data.

5.2. Phase 2: Backend Development

1. **API Development with FastAPI**: Utilize FastAPI to create robust APIs for encoding and decoding images, facilitating seamless communication between the frontend and backend.

2. **Database Schema Design using MySQL**: Design a database schema to store encoded images, along with metadata such as encryption keys and steganography techniques used.

3. **Integration of Steganography and Encryption Libraries**: Implement the selected steganography and encryption techniques from the `stegano`, `numpy`, and `PIL` libraries into the backend for image encoding and decoding processes.

5.3. Phase 3: Frontend Development

1. **User Interface Design**: Develop an intuitive user interface using HTML, CSS, and JavaScript to enable users to interact with the PixHide platform effectively.

2. **Input Form for Encoding**: Create input forms for users to specify carrier images, messages to be concealed, encryption keys, and other relevant parameters during the encoding process.

3. **Input Form for Decoding**: Design input forms for users to input encoded images and select the steganography solution used during the decoding process.

5.4. Phase 4: Testing and Quality Assurance

1. **Unit Testing**: Conduct thorough unit tests for each component of the backend, ensuring that the encoding and decoding processes function correctly.

2. **Integration Testing**: Perform integration tests to verify the seamless integration of frontend and backend components, including API endpoints and database operations.

3. **User Acceptance Testing (UAT)**: Engage stakeholders in UAT to validate that the PixHide platform meets their requirements and expectations for data concealment and retrieval.

5.5. Phase 5: Deployment and Version Control

1. **Deployment on Render**: Deploy the PixHide platform on the Render platform to make it accessible to users over the internet.
2. **Version Control with Git and Github**: Utilize Git and Github for version control, enabling collaborative development and tracking of changes throughout the project lifecycle.

By following these development phases and leveraging the specified tools and technologies, PixHide platform will be efficiently developed, offering users a versatile solution for secure data concealment and retrieval.

6. CONCLUSION

Finally, Image Steganography is a captivating technique that conceals secret information within digital images. To implement effective steganographic systems, it's essential to understand the key components, functional flow, and commonly used algorithms.
By mastering these concepts, one can develop secure and reliable image steganography solutions for various applications.
And that is what I have tried to do for the creation of PixHide platform that integrates advanced steganography and encryption techniques with a user-friendly interface, providing users with a seamless experience for concealing and retrieving hidden data. By leveraging technologies such as `stegano`, `numpy`, and `PIL` libraries for steganography and encryption, along with FastAPI for backend development and HTML/CSS/JS for frontend development, PixHide offers a versatile solution for protecting sensitive information.
Through planning, rigorous development phases, and comprehensive testing, PixHide ensures reliability, security, and ease of use. By deploying the platform on Render and utilizing version control with Git and Github, PixHide can be continuously improved and maintained.
Overall, PixHide empowers users to safeguard their data while providing a robust and intuitive platform for data concealment and retrieval, meeting the diverse needs of the users.

References

[1] *Batch Stega*. Adversarial batch image steganography against CNN-based pooled steganalysis. Li Li, Weiming Zhang, Chuan Qin, Kejiang Chen, Wenbo Zhou, Nenghai Yu.

[2] *Crypto , Stega and Obfuscation Difference*. URL: https://www.eccouncil.org/cybersecurity-exchange/ethical-hacking/what-is-steganography-guide-meaning-types-tools/.

[3] *CSSIS*. Implementation of Spread Spectrum Image Steganography ,Frederick S. Brundick and Lisa M. Marvel ,Computational and Information Sciences Directorate, ARL.

[4] *LSB*. Naveen K Nishchal 2019, Digital techniques of data and image encryption. IOP Publishing Ltd.

[5] *LSB Tool*. URL: https://stylesuxx.github.io/steganography/.

[6] *Steganography*. URL: https://en.wikipedia.org/wiki/Steganography.